|  |  |
| --- | --- |
|  |  |
| {- |  |
|  | Introduction to Functional Programming CS3016 Exam Haskell Code |
|  | Geoffrey Natin 10/5/17 17:38 |
|  |  |
|  | This file contains attempts at the questions |
|  | from the CS3016 Exams from 2013-2016. |
|  |  |
|  | These are 'attempts'. Who knows how many marks they would go for... |
|  | (yes, I know: Dr Andrew Butterfield probably!) |
|  | -} |
|  | import Test.HUnit |
|  |  |
|  | --------------------------------2016----------------------------------------- |
|  |  |
|  | ------------Q1-------------- |
|  |  |
|  | --Q1a) |
|  | head :: [a] -> a |
|  | head (x:xs) = x |
|  |  |
|  | --Q1b) |
|  | init :: [a] -> [a] |
|  | init [x] = [] |
|  | init (x:xs) = x : init xs |
|  |  |
|  | --Q1c) |
|  | last :: [a] -> a |
|  | last [x] = x |
|  | last (x:xs) = last xs |
|  |  |
|  | --1d) |
|  | span :: (a -> Bool) -> [a] -> ([a],[a]) |
|  | span p [] = ([],[]) |
|  | span p (x:xs) | p x = (x:a,b) |
|  | | otherwise = ([],x:xs) |
|  | where (a,b) = span p xs |
|  |  |
|  | --1e) |
|  | (!!) :: [a] -> Int -> a |
|  | (!!) (x:xs) 0 = x |
|  | (!!) (x:xs) n = (!!) xs $ n - 1 |
|  |  |
|  | --1f) |
|  | fold1 :: (a -> a -> a) -> [a] -> a |
|  | fold1 \_ [x] = x |
|  | fold1 op (x:xs) = op x (fold1 op xs) |
|  |  |
|  |  |
|  | ------------Q2-------------- |
|  |  |
|  | --2a) |
|  | hof :: a -> (a->a->a) -> [a] -> a |
|  | hof x fx [] = x |
|  | hof x fx (y:ys) = fx y (hof fx x ys) |
|  |  |
|  | --2b) |
|  | f1 = hof 1 (\*) |
|  | f2 = hof 0 f2x |
|  | where f2x x y = (1+y) |
|  | f3 = hof 0 (+) |
|  | f4 = hof [] (++) |
|  | f5 = hof 0 f5x |
|  | where f5x x y = (x\*x)+y |
|  |  |
|  | --2c) |
|  | {- |
|  | There is no pattern for a 'Many' tree with 'x < i'. So if you are |
|  | searching a node that isn't a leaf, there will be a runtime error |
|  | if the node's integer is greater than the integer you are searching |
|  | for. This means node's left branches are never traversed. |
|  |  |
|  | There is no pattern for searching an empty tree, so there will be a |
|  | runtime error if this is attempted. |
|  |  |
|  | The only pattern provided for searching a leaf node is for the case |
|  | that the integer of the leaf node is equal to 'x'. Any searching of |
|  | a leaf node where this is not the case will result in a runtime |
|  | error. |
|  | -} |
|  |  |
|  |  |
|  | ------------Q3-------------- |
|  |  |
|  | --3a) |
|  | eval :: Dict -> Expr -> Maybe Int |
|  | eval \_ (K i) = Just i |
|  | eval d (V s) = lkp s d |
|  | eval d (Add e1 e2) = case (eval d e1,eval e2) of |
|  | (Nothing,\_) -> Nothing |
|  | (\_,Nothing) -> Nothing |
|  | (Just x, Just y) -> Just (x+y) |
|  | eval d (Dvd e1 e2) = case(eval d e1,eval d e2) of |
|  | (Nothing,\_) -> Nothing |
|  | (\_,Nothing) -> Nothing |
|  | (Just x, Just y) |
|  | | y == 0 -> Nothing |
|  | | otherwise -> Just (div x y) |
|  | eval d (Let v e1 e2) = case (eval d e1) of |
|  | | Nothing -> Nothing |
|  | | Just i -> eval (ins v i d) e2 |
|  |  |
|  | --3b) |
|  | {- |
|  | Won't attempt to draw a tree in a .hs file. |
|  | -} |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | --------------------------------2015----------------------------------------- |
|  |  |
|  | ------------Q1-------------- |
|  |  |
|  | --1a) |
|  | repeat :: a -> [a] |
|  | repeat x = x : repeat x |
|  |  |
|  | --1b) |
|  | replicate :: Int -> a -> [a] |
|  | replicate 0 \_ = [] |
|  | replicate n x = x : replicate (n-1) x |
|  |  |
|  | --1c) |
|  | concat :: [[a]] -> [a] |
|  | concat [] = [] |
|  | concat (x:xs) = x ++ (concat xs) |
|  |  |
|  | --1d) |
|  | zip :: [a] -> [b] -> [(a,b)] |
|  | zip [] [] = [] |
|  | zip (x:xs) (y:ys) = (x,y) : (zip xs ys) |
|  |  |
|  | --1e) |
|  | unzip :: [(a,b)] -> ([a],[b]) |
|  | unzip [] = ([],[]) |
|  | unzip ((x,y):xs) = (x:a,y:b) |
|  | where (a,b) = unzip xs |
|  | --1f) |
|  | minimum :: (Ord a) => [a] -> a |
|  | minimum [x] = x |
|  | minimum (x:y:xs) | x < y = minimum (x:xs) |
|  | | otherwise = minimum (y:xs) |
|  |  |
|  |  |
|  | ------------Q2-------------- |
|  |  |
|  | --2a) |
|  | hof :: (a -> a -> b) -> [a] -> [a] -> [b] |
|  | hof \_ [] \_ = [] |
|  | hof \_ \_ [] = [] |
|  | hof fx (x:xs) (y:ys) = (fx x y) : (hof fx xs ys) |
|  |  |
|  | --2b) |
|  | {- |
|  | hof :: (a -> b -> c) -> [a] -> [b] -> [c] |
|  | -} |
|  |  |
|  | --2c) |
|  | f1 = hof (\*) |
|  | f2 = hof (+) |
|  | f3 = hof f3b |
|  | where f3b x y = (x y) |
|  | f4 = hof f4b |
|  | where f4b x y = (y,x) |
|  | f5 = hof f5b |
|  | where f5b x y = x |
|  |  |
|  | --2d) |
|  | {- |
|  | Yes, hof is provided in the prelude under the name 'zipWith'. |
|  | -} |
|  |  |
|  |  |
|  | ------------Q3-------------- |
|  |  |
|  | --3a) |
|  | {- |
|  | The 'lookup' function creates a hash of the string being searched |
|  | for. |
|  | It then finds the bucket in the table who's key is the hash |
|  | of the string being searched for. |
|  | It then finds the key-value pair in bucket who's key is the string |
|  | that is being searched for. |
|  | The value of this key-value pair is what is then returned. |
|  |  |
|  | The function 'lookup' will produce an error if there is no bucket |
|  | in 'table' who's key is the hash of the string being searched for. |
|  |  |
|  | If a bucket who's key is the hash of the string being searched for |
|  | is found in the table, then there will be an error if that bucket |
|  | does not contain a key-value pair who's key is string being |
|  | searched for. |
|  | -} |
|  |  |
|  | --3b) |
|  | lookup :: String -> (HashTable String String) -> Maybe String |
|  | lookup str table = |
|  | let hashValue = hash str |
|  | bucketTuple = find (==hashValue) . fst) table |
|  | valueTuple = case bucketTuple of |
|  | Nothing -> Nothing |
|  | Just (\_,bucket) -> find (==str . fst) bucket |
|  | value = case valueTuple of |
|  | Nothing -> Nothing |
|  | Just (\_,v) -> Just v |
|  | in value |
|  |  |
|  | --3c) |
|  | type Err = String |
|  | lookup2 :: String -> (HashTable String String) -> Either Err String |
|  | lookup2 str table = |
|  | let hashValue = hash str |
|  | bucketTuple = find (==hashValue) . fst) table |
|  | valueTuple = case bucketTuple of |
|  | Nothing -> Left "No Buckets present in hashtable for hash of string." |
|  | Just (\_,bucket) -> Right (find (==str . fst) bucket) |
|  | value = case valueTuple of |
|  | Left err -> err |
|  | Right Nothing -> Left "Key not found in bucket for hash of string." |
|  | Right (Just (\_,v) -> Right v |
|  | in value |
|  |  |
|  |  |
|  | ------------Q4-------------- |
|  |  |
|  | --4a) |
|  | {- |
|  | Won't attempt to draw a tree in a .hs file. |
|  | -} |
|  |  |
|  | --4b) |
|  | {- |
|  |  |
|  | Strict: |
|  | take 2 (zig 20) |
|  | take 2 (20:zag(19)) |
|  | take 2 (20:19:zig 18) |
|  | take 2 (20:19:18:zag 17) |
|  | take 2 (20:19:18:17:zig 16) |
|  | take 2 (20:19:18:17:16:zag 15) |
|  | take 2 (20:19:18:17:16:15:zig 14)... |
|  | This continues on infinitely list (it won't stop at zig 0 or zag 0) |
|  |  |
|  | Lazy: |
|  | take 2 (zig 20) |
|  | 20 : take 1 (zig 20) |
|  | 20 : 19 : take 0 (zig 20) |
|  | 20 : 19 : [] |
|  | [20,19] |
|  |  |
|  | As you can see, only lazy evaluation will give the proper result. |
|  | -} |
|  |  |
|  | --4ci) take 0 [] |
|  |  |
|  | --4cii) There is no expression that can be evaluated strictly but not lazily. |
|  |  |
|  | --4ciii) take 2 (zig 20) |
|  |  |
|  | --4civ) zig 0 |
|  |  |
|  | --4d) |
|  | hashFileToAnother = |
|  | do putStr "What file do you want hashed?" |
|  | from <- getLine |
|  | putStr "Where do you want the hash?" |
|  | to <- getLine |
|  | text <- readFile (from++".txt") |
|  | writeFile (to++".txt") (hash text) |
|  | putStr "Done." |
|  |  |
|  |  |
|  |  |
|  |  |
|  | --------------------------------2014----------------------------------------- |
|  |  |
|  | ------------Q1-------------- |
|  |  |
|  | --1a) |
|  | tail :: [a] -> [a] |
|  | tail (x:xs) = xs |
|  |  |
|  | --1b) |
|  | (++) :: [a] -> [a] -> [a] |
|  | (++) [] xs = xs |
|  | (++) (x:xs) ys = x : ((++) xs ys) |
|  |  |
|  | --1c) |
|  | init2 :: [a] -> [a] |
|  | init2 [x] = [] |
|  | init2 (x:xs) = x : init2 xs |
|  |  |
|  | --1d) |
|  | break :: (a -> Bool) -> [a] -> ([a],[a]) |
|  | break p (x:xs) | p x = ([],xs) |
|  | | otherwise = (x:a,b) |
|  | where (a,b) = break p xs |
|  |  |
|  | --1e) |
|  | reverse :: [a] -> [a] |
|  | reverse [] = [] |
|  | reverse (x:xs) = reverse xs ++ [x] |
|  |  |
|  | --1f) |
|  | maximum :: Ord a => [a] -> a |
|  | maximum [x] = x |
|  | maximum (x:y:xs) | x > y = maximum (x:xs) |
|  | | otherwise = maximum (y:xs) |
|  |  |
|  |  |
|  | ------------Q2-------------- |
|  |  |
|  | --2a) |
|  | hof :: (a -> b -> a) -> a -> [b] -> a |
|  | hof \_ e [] = e |
|  | hof fx e (x:xs) = hof fx (fx e x) xs |
|  |  |
|  | --2b) |
|  | f1 = hof (\*) |
|  | f2 = hof f2b |
|  | where f2b x y = (x+1) |
|  | f3 = hof (+) |
|  | f4 = hof (++) |
|  | f5 = hof f5b |
|  | where f5b x y = (x+y\*y) |
|  |  |
|  | --2c) |
|  | {- |
|  | Yes, hof is provided in the prelude under the name 'foldl'. |
|  | -} |
|  |  |
|  |  |
|  | ------------Q3-------------- |
|  |  |
|  | --3a) |
|  | {- |
|  | In the first pattern, there is no case for when 'x < i', so |
|  | searching a non-leaf node with a value bigger than x will |
|  | cause a run-time error. |
|  | The searches of non-leaf nodes have no case that leads to |
|  | the traversal of the left branch. The searching of a tree |
|  | only ever deepens on the right hand side of the tree. |
|  |  |
|  | There is no pattern for searching an Empty tree. Any attempt |
|  | to search an Empty tree will result in a runtime error. |
|  |  |
|  | The only case given for the pattern searching a leaf node is |
|  | the case where 'x == i'. Any attempt to search a node where |
|  | its value is not equal to x will result in a runtime error. |
|  | -} |
|  |  |
|  | --3b) |
|  | search :: Int -> Tree -> Maybe String |
|  | search \_ Empty = Nothing |
|  | search x (Many left i s right) |
|  | | x == i = Just s |
|  | | x > i = search x right |
|  | | x < i = search x left |
|  | search x (Single i s) |
|  | | x == i = Just s |
|  | | otherwise = Nothing |
|  |  |
|  | --3c) |
|  | search :: Int -> Tree -> Either Err String |
|  | search \_ Empty = Left "The tree you are searching is empty." |
|  | search x (Many left i s right) |
|  | | x == i = Right s |
|  | | x > i = search x right |
|  | | x < i = search x left |
|  | search x (Single i s) |
|  | | x == i = Right s |
|  | | otherwise = Left "Key not found in tree." |
|  |  |
|  |  |
|  | ------------Q4-------------- |
|  |  |
|  | --4a) |
|  | {- |
|  | Won't attempt to create a tree in a .hs file. |
|  | -} |
|  |  |
|  | --4b) |
|  | {- |
|  | take 2 (evenup 2) |
|  |  |
|  | Strict: |
|  | take 2 (evenup 2) |
|  | take 2 (2:evenup 3) |
|  | take 2 (2:3:evenup 4) |
|  | take 2 (2:3:4:evenup 5) |
|  | This will continue on infinitely and therefore not give a result. |
|  |  |
|  | Lazy: |
|  | take 2 (evenup 2) |
|  | take 2 (2:evenup 4) |
|  | 2: take 1 (evenup 4) |
|  | 2: take 1 (4:evenup 6) |
|  | 2: 4: take 0 (evenup 6) |
|  | 2: 4: [] |
|  | [2,4] |
|  |  |
|  | As you can see, only lazy evaluation will give the proper result. |
|  | -} |
|  |  |
|  | --4ci) evenup 0 |
|  |  |
|  | --4cii) There is no expression that terminates when evaluated strictly but |
|  | -- not when evaluated lazily. |
|  |  |
|  | --4ciii) take 2 (evenup 2) |
|  |  |
|  | --4civ) take 0 [] |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | --------------------------------2013----------------------------------------- |
|  |  |
|  | ------------Q1-------------- |
|  |  |
|  | --1a) |
|  | null :: [a] -> Bool |
|  | null [] = True |
|  | null \_ = False |
|  |  |
|  | {- Commented as is duplicate in file |
|  | --1b) |
|  | (+) :: [a] -> [a] -> [a] |
|  | (++) [] xs = xs |
|  | (++) xs [] = xs |
|  | (++) (x:xs) ys = x : (++) xs ys |
|  | -} |
|  | # |
|  | --1c) |
|  | last2 :: [a] -> a |
|  | last2 [x] = x |
|  | last2 (x:xs) = last2 xs |
|  |  |
|  | --1d) |
|  | dropWhile :: (a -> Bool) -> [a] -> [a] |
|  | dropWhile \_ [] = [] |
|  | dropeWhile p (x:xs) | p x = dropWhile p xs |
|  | | otherwise = xs |
|  |  |
|  | --1e) |
|  | filter :: (a -> Bool) -> [a] -> [a] |
|  | filter \_ [] = [] |
|  | filter p (x:xs) | p x = (x : (filter p xs)) |
|  | | otherwise = (filter p xs) |
|  |  |
|  | --1f) |
|  | foldr1 :: (a -> a -> a) -> [a] -> a |
|  | foldr1 \_ [x] = x |
|  | foldr1 op (x:xs) = op x (foldr1 op xs) |
|  |  |
|  |  |
|  | ------------Q2-------------- |
|  |  |
|  | --2a) |
|  | hof :: (a -> b -> b) -> b -> [a] -> b |
|  | hof \_ e [] = e |
|  | hof fx e (x:xs) = fx x (hof fx e xs) |
|  |  |
|  | --2b) |
|  | f1 = hof (\*) 1 |
|  | f2 = hof f2b 0 |
|  | where f2b x y = (1+y) |
|  | f3 = hof (+) 0 |
|  | f4 = hof (++) [] |
|  | f5 = hof f5b 0 |
|  | where f5b x y = (x\*x+y) |
|  |  |
|  | --2c) |
|  | {- |
|  | Yes, hof is provided by the prelude under the name 'foldr'. |
|  | -} |
|  |  |
|  |  |
|  | ------------Q3-------------- |
|  |  |
|  | --3a) |
|  | {- |
|  | The function will fail if the 'lkp' does not return a 'Just' in the |
|  | second pattern. |
|  |  |
|  | In the fourth pattern, if 'eval d e2' evaluates to zero there will |
|  | be an runtime error because of an attempt to divide by zero. |
|  | -} |
|  |  |
|  | --3b) |
|  | eval :: Dict -> Expr -> Maybe Int |
|  | eval \_ (K i) = Just i |
|  | eval d (V s) = lkp s d |
|  | eval d (Add e1 e2) = case (eval d e1,eval d e2) of |
|  | (Nothing,\_) -> Nothing |
|  | (\_,Nothing) -> Nothing |
|  | (Just a, Just b) -> Just (a+b) |
|  | eval d (Dvd e1 e2) = case (eval d e1,eval d e2) of |
|  | (Nothing,\_) -> Nothing |
|  | (\_,Nothing) -> Nothing |
|  | (Just a, Just b) |
|  | | b == 0 -> Nothing |
|  | | otherwise -> Just (div a b) |
|  | eval d (Let v e1 e2) = case eval d e1 of |
|  | Nothing -> Nothing |
|  | Just i -> eval (ins v i d) e2 |
|  |  |
|  | --3c) |
|  | eval :: Dict -> Expr -> Either Err Int |
|  | eval \_ (K i) = Right i |
|  | eval d (V s) = case (lkp s d) of |
|  | Nothing -> Left "Function 'lkp' returned Nothing." |
|  | Just i -> Right i |
|  | eval d (Add e1 e2) = case (eval d e1,eval d e2) of |
|  | (Left msg,\_) -> Left msg |
|  | (\_,Left msg) -> Left msg |
|  | (Right a, Right b) -> Right (a+b) |
|  | eval d (Dvd e1 e2) = case (eval d e1,eval d e2) of |
|  | (Left msg,\_) -> Left msg |
|  | (\_,Left msg) -> Left msg |
|  | (Right a, Right b) |
|  | | b == 0 -> Left "Won't divide by zero- sorry! x" |
|  | | otherwise -> Right (div a b) |
|  | eval d (Let v e1 e2) = case eval d e1 of |
|  | Left msg -> Left msg |
|  | Right i -> eval (ins v i d) e2 |
|  |  |
|  |  |
|  | ------------Q4-------------- |
|  |  |
|  | --4a) |
|  | {- |
|  | Won't attempt to draw a tree in a .hs file. |
|  | -} |
|  |  |
|  | --4b) |
|  | {- |
|  | take 2 (down 42) |
|  |  |
|  | Strict: |
|  | take 2 (down 42) |
|  | take 2 (42:down 41) |
|  | take 2 (42:41:down 40) |
|  | take 2 (42:41:40:down 39) |
|  | This continues on infitely (does not stop at down 0) and therefore a result will never emerge. |
|  |  |
|  | Lazy: |
|  | take 2 (down 42) |
|  | take 2 (42:down 41) |
|  | 42: take 1 (down 41) |
|  | 42: take 1 (41:down 40) |
|  | 42: 41: take 0 (down 40) |
|  | 42: 41: [] |
|  | [42,41] |
|  |  |
|  | As you can see, only lazy evaluation will give the proper result. |
|  | -} |
|  |  |
|  | --4ci) take 0 [] |
|  |  |
|  | --4cii) There is no such expression |
|  |  |
|  | --4ciii) take 2 (down 42) |
|  |  |
|  | --4civ) down 42 |
|  |  |
|  | --4d) |
|  | fileToUpperCase = |
|  | do putStr "What file do you want to uppercase?" |
|  | from <- getLine |
|  | text <- readFile (from++".in") |
|  | writeFile (from++".ing") (toUpper text) |
|  | putStr "Done." |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
| {- |  |
|  | Introduction to Functional Programming CS3016 Exam Haskell Code |
|  | Geoffrey Natin 10/5/17 17:38 |
|  |  |
|  | This file contains attempts at the questions |
|  | from the CS3016 Exams from 2013-2016. |
|  |  |
|  | These are 'attemps'. Who knows how many marks they would go for... |
|  | (yes, I know: Dr Andrew Butterfield probably!) |
|  | -} |
|  | import Test.HUnit |
|  |  |
|  | --------------------------------2016----------------------------------------- |
|  |  |
|  | ------------Q1-------------- |
|  |  |
|  | --Q1a) |
|  | head :: [a] -> a |
|  | head (x:xs) = x |
|  |  |
|  | --Q1b) |
|  | init :: [a] -> [a] |
|  | init [x] = [] |
|  | init (x:xs) = x : init xs |
|  |  |
|  | --Q1c) |
|  | last :: [a] -> a |
|  | last [x] = x |
|  | last (x:xs) = last xs |
|  |  |
|  | --1d) |
|  | span :: (a -> Bool) -> [a] -> ([a],[a]) |
|  | span p [] = ([],[]) |
|  | span p (x:xs) | p x = (x:a,b) |
|  | | otherwise = ([],x:xs) |
|  | where (a,b) = span p xs |
|  |  |
|  | --1e) |
|  | (!!) :: [a] -> Int -> a |
|  | (!!) (x:xs) 0 = x |
|  | (!!) (x:xs) n = (!!) xs $ n - 1 |
|  |  |
|  | --1f) |
|  | fold1 :: (a -> a -> a) -> [a] -> a |
|  | fold1 \_ [x] = x |
|  | fold1 op (x:xs) = op x (fold1 op xs) |
|  |  |
|  |  |
|  | ------------Q2-------------- |
|  |  |
|  | --2a) |
|  | hof :: a -> (a->a->a) -> [a] -> a |
|  | hof x fx [] = x |
|  | hof x fx (y:ys) = fx y (hof fx x ys) |
|  |  |
|  | --2b) |
|  | f1 = hof 1 (\*) |
|  | f2 = hof 0 f2x |
|  | where f2x x y = (1+y) |
|  | f3 = hof 0 (+) |
|  | f4 = hof [] (++) |
|  | f5 = hof 0 f5x |
|  | where f5x x y = (x\*x)+y |
|  |  |
|  | --2c) |
|  | {- |
|  | There is no pattern for a 'Many' tree with 'x < i'. So if you are |
|  | searching a node that isn't a leaf, there will be a runtime error |
|  | if the node's integer is greater than the integer you are searching |
|  | for. This means node's left branches are never traversed. |
|  |  |
|  | There is no pattern for searching an empty tree, so there will be a |
|  | runtime error if this is attempted. |
|  |  |
|  | The only pattern provided for searching a leaf node is for the case |
|  | that the integer of the leaf node is equal to 'x'. Any searching of |
|  | a leaf node where this is not the case will result in a runtime |
|  | error. |
|  | -} |
|  |  |
|  |  |
|  | ------------Q3-------------- |
|  |  |
|  | --3a) |
|  | eval :: Dict -> Expr -> Maybe Int |
|  | eval \_ (K i) = Just i |
|  | eval d (V s) = lkp s d |
|  | eval d (Add e1 e2) = case (eval d e1,eval e2) of |
|  | (Nothing,\_) -> Nothing |
|  | (\_,Nothing) -> Nothing |
|  | (Just x, Just y) -> Just (x+y) |
|  | eval d (Dvd e1 e2) = case(eval d e1,eval d e2) of |
|  | (Nothing,\_) -> Nothing |
|  | (\_,Nothing) -> Nothing |
|  | (Just x, Just y) |
|  | | y == 0 -> Nothing |
|  | | otherwise -> Just (div x y) |
|  | eval d (Let v e1 e2) = case (eval d e1) of |
|  | | Nothing -> Nothing |
|  | | Just i -> eval (ins v i d) e2 |
|  |  |
|  | --3b) |
|  | {- |
|  | Won't attempt to draw a tree in a .hs file. |
|  | -} |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | --------------------------------2015----------------------------------------- |
|  |  |
|  | ------------Q1-------------- |
|  |  |
|  | --1a) |
|  | repeat :: a -> [a] |
|  | repeat x = x : repeat x |
|  |  |
|  | --1b) |
|  | replicate :: Int -> a -> [a] |
|  | replicate 0 \_ = [] |
|  | replicate n x = x : replicate (n-1) x |
|  |  |
|  | --1c) |
|  | concat :: [[a]] -> [a] |
|  | concat [] = [] |
|  | concat (x:xs) = x ++ (concat xs) |
|  |  |
|  | --1d) |
|  | zip :: [a] -> [b] -> [(a,b)] |
|  | zip [] [] = [] |
|  | zip (x:xs) (y:ys) = (x,y) : (zip xs ys) |
|  |  |
|  | --1e) |
|  | unzip :: [(a,b)] -> ([a],[b]) |
|  | unzip [] = ([],[]) |
|  | unzip ((x,y):xs) = (x:a,y:b) |
|  | where (a,b) = unzip xs |
|  | --1f) |
|  | minimum :: (Ord a) => [a] -> a |
|  | minimum [x] = x |
|  | minimum (x:y:xs) | x < y = minimum (x:xs) |
|  | | otherwise = minimum (y:xs) |
|  |  |
|  |  |
|  | ------------Q2-------------- |
|  |  |
|  | --2a) |
|  | hof :: (a -> a -> b) -> [a] -> [a] -> [b] |
|  | hof \_ [] \_ = [] |
|  | hof \_ \_ [] = [] |
|  | hof fx (x:xs) (y:ys) = (fx x y) : (hof fx xs ys) |
|  |  |
|  | --2b) |
|  | {- |
|  | hof :: (a -> b -> c) -> [a] -> [b] -> [c] |
|  | -} |
|  |  |
|  | --2c) |
|  | f1 = hof (\*) |
|  | f2 = hof (+) |
|  | f3 = hof f3b |
|  | where f3b x y = (x y) |
|  | f4 = hof f4b |
|  | where f4b x y = (y,x) |
|  | f5 = hof f5b |
|  | where f5b x y = x |
|  |  |
|  | --2d) |
|  | {- |
|  | Yes, hof is provided in the prelude under the name 'zipWith'. |
|  | -} |
|  |  |
|  |  |
|  | ------------Q3-------------- |
|  |  |
|  | --3a) |
|  | {- |
|  | The 'lookup' function creates a hash of the string being searched |
|  | for. |
|  | It then finds the bucket in the table who's key is the hash |
|  | of the string being searched for. |
|  | It then finds the key-value pair in bucket who's key is the string |
|  | that is being searched for. |
|  | The value of this key-value pair is what is then returned. |
|  |  |
|  | The function 'lookup' will produce an error if there is no bucket |
|  | in 'table' who's key is the hash of the string being searched for. |
|  |  |
|  | If a bucket who's key is the hash of the string being searched for |
|  | is found in the table, then there will be an error if that bucket |
|  | does not contain a key-value pair who's key is string being |
|  | searched for. |
|  | -} |
|  |  |
|  | --3b) |
|  | lookup :: String -> (HashTable String String) -> Maybe String |
|  | lookup str table = |
|  | let hashValue = hash str |
|  | bucketTuple = find (==hashValue) . fst) table |
|  | valueTuple = case bucketTuple of |
|  | Nothing -> Nothing |
|  | Just (\_,bucket) -> find (==str . fst) bucket |
|  | value = case valueTuple of |
|  | Nothing -> Nothing |
|  | Just (\_,v) -> Just v |
|  | in value |
|  |  |
|  | --3c) |
|  | type Err = String |
|  | lookup2 :: String -> (HashTable String String) -> Either Err String |
|  | lookup2 str table = |
|  | let hashValue = hash str |
|  | bucketTuple = find (==hashValue) . fst) table |
|  | valueTuple = case bucketTuple of |
|  | Nothing -> Left "No Buckets present in hashtable for hash of string." |
|  | Just (\_,bucket) -> Right (find (==str . fst) bucket) |
|  | value = case valueTuple of |
|  | Left err -> err |
|  | Right Nothing -> Left "Key not found in bucket for hash of string." |
|  | Right (Just (\_,v) -> Right v |
|  | in value |
|  |  |
|  |  |
|  | ------------Q4-------------- |
|  |  |
|  | --4a) |
|  | {- |
|  | Won't attempt to draw a tree in a .hs file. |
|  | -} |
|  |  |
|  | --4b) |
|  | {- |
|  |  |
|  | Strict: |
|  | take 2 (zig 20) |
|  | take 2 (20:zag(19)) |
|  | take 2 (20:19:zig 18) |
|  | take 2 (20:19:18:zag 17) |
|  | take 2 (20:19:18:17:zig 16) |
|  | take 2 (20:19:18:17:16:zag 15) |
|  | take 2 (20:19:18:17:16:15:zig 14)... |
|  | This continues on infinitely list (it won't stop at zig 0 or zag 0) |
|  |  |
|  | Lazy: |
|  | take 2 (zig 20) |
|  | 20 : take 1 (zig 20) |
|  | 20 : 19 : take 0 (zig 20) |
|  | 20 : 19 : [] |
|  | [20,19] |
|  |  |
|  | As you can see, only lazy evaluation will give the proper result. |
|  | -} |
|  |  |
|  | --4ci) take 0 [] |
|  |  |
|  | --4cii) There is no expression that can be evaluated strictly but not lazily. |
|  |  |
|  | --4ciii) take 2 (zig 20) |
|  |  |
|  | --4civ) zig 0 |
|  |  |
|  | --4d) |
|  | hashFileToAnother = |
|  | do putStr "What file do you want hashed?" |
|  | from <- getLine |
|  | putStr "Where do you want the hash?" |
|  | to <- getLine |
|  | text <- readFile (from++".txt") |
|  | writeFile (to++".txt") (hash text) |
|  | putStr "Done." |
|  |  |
|  |  |
|  |  |
|  |  |
|  | --------------------------------2014----------------------------------------- |
|  |  |
|  | ------------Q1-------------- |
|  |  |
|  | --1a) |
|  | tail :: [a] -> [a] |
|  | tail (x:xs) = xs |
|  |  |
|  | --1b) |
|  | (++) :: [a] -> [a] -> [a] |
|  | (++) [] xs = xs |
|  | (++) (x:xs) ys = x : ((++) xs ys) |
|  |  |
|  | --1c) |
|  | init2 :: [a] -> [a] |
|  | init2 [x] = [] |
|  | init2 (x:xs) = x : init2 xs |
|  |  |
|  | --1d) |
|  | break :: (a -> Bool) -> [a] -> ([a],[a]) |
|  | break p (x:xs) | p x = ([],xs) |
|  | | otherwise = (x:a,b) |
|  | where (a,b) = break p xs |
|  |  |
|  | --1e) |
|  | reverse :: [a] -> [a] |
|  | reverse [] = [] |
|  | reverse (x:xs) = reverse xs ++ [x] |
|  |  |
|  | --1f) |
|  | maximum :: Ord a => [a] -> a |
|  | maximum [x] = x |
|  | maximum (x:y:xs) | x > y = maximum (x:xs) |
|  | | otherwise = maximum (y:xs) |
|  |  |
|  |  |
|  | ------------Q2-------------- |
|  |  |
|  | --2a) |
|  | hof :: (a -> b -> a) -> a -> [b] -> a |
|  | hof \_ e [] = e |
|  | hof fx e (x:xs) = hof fx (fx e x) xs |
|  |  |
|  | --2b) |
|  | f1 = hof (\*) |
|  | f2 = hof f2b |
|  | where f2b x y = (x+1) |
|  | f3 = hof (+) |
|  | f4 = hof (++) |
|  | f5 = hof f5b |
|  | where f5b x y = (x+y\*y) |
|  |  |
|  | --2c) |
|  | {- |
|  | Yes, hof is provided in the prelude under the name 'foldl'. |
|  | -} |
|  |  |
|  |  |
|  | ------------Q3-------------- |
|  |  |
|  | --3a) |
|  | {- |
|  | In the first pattern, there is no case for when 'x < i', so |
|  | searching a non-leaf node with a value bigger than x will |
|  | cause a run-time error. |
|  | The searches of non-leaf nodes have no case that leads to |
|  | the traversal of the left branch. The searching of a tree |
|  | only ever deepens on the right hand side of the tree. |
|  |  |
|  | There is no pattern for searching an Empty tree. Any attempt |
|  | to search an Empty tree will result in a runtime error. |
|  |  |
|  | The only case given for the pattern searching a leaf node is |
|  | the case where 'x == i'. Any attempt to search a node where |
|  | its value is not equal to x will result in a runtime error. |
|  | -} |
|  |  |
|  | --3b) |
|  | search :: Int -> Tree -> Maybe String |
|  | search \_ Empty = Nothing |
|  | search x (Many left i s right) |
|  | | x == i = Just s |
|  | | x > i = search x right |
|  | | x < i = search x left |
|  | search x (Single i s) |
|  | | x == i = Just s |
|  | | otherwise = Nothing |
|  |  |
|  | --3c) |
|  | search :: Int -> Tree -> Either Err String |
|  | search \_ Empty = Left "The tree you are searching is empty." |
|  | search x (Many left i s right) |
|  | | x == i = Right s |
|  | | x > i = search x right |
|  | | x < i = search x left |
|  | search x (Single i s) |
|  | | x == i = Right s |
|  | | otherwise = Left "Key not found in tree." |
|  |  |
|  |  |
|  | ------------Q4-------------- |
|  |  |
|  | --4a) |
|  | {- |
|  | Won't attempt to create a tree in a .hs file. |
|  | -} |
|  |  |
|  | --4b) |
|  | {- |
|  | take 2 (evenup 2) |
|  |  |
|  | Strict: |
|  | take 2 (evenup 2) |
|  | take 2 (2:evenup 3) |
|  | take 2 (2:3:evenup 4) |
|  | take 2 (2:3:4:evenup 5) |
|  | This will continue on infinitely and therefore not give a result. |
|  |  |
|  | Lazy: |
|  | take 2 (evenup 2) |
|  | take 2 (2:evenup 4) |
|  | 2: take 1 (evenup 4) |
|  | 2: take 1 (4:evenup 6) |
|  | 2: 4: take 0 (evenup 6) |
|  | 2: 4: [] |
|  | [2,4] |
|  |  |
|  | As you can see, only lazy evaluation will give the proper result. |
|  | -} |
|  |  |
|  | --4ci) evenup 0 |
|  |  |
|  | --4cii) There is no expression that terminates when evaluated strictly but |
|  | -- not when evaluated lazily. |
|  |  |
|  | --4ciii) take 2 (evenup 2) |
|  |  |
|  | --4civ) take 0 [] |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | --------------------------------2013----------------------------------------- |
|  |  |
|  | ------------Q1-------------- |
|  |  |
|  | --1a) |
|  | null :: [a] -> Bool |
|  | null [] = True |
|  | null \_ = False |
|  |  |
|  | {- Commented as is duplicate in file |
|  | --1b) |
|  | (+) :: [a] -> [a] -> [a] |
|  | (++) [] xs = xs |
|  | (++) xs [] = xs |
|  | (++) (x:xs) ys = x : (++) xs ys |
|  | -} |
|  |  |
|  | --1c) |
|  | last2 :: [a] -> a |
|  | last2 [x] = x |
|  | last2 (x:xs) = last2 xs |
|  |  |
|  | --1d) |
|  | dropWhile :: (a -> Bool) -> [a] -> [a] |
|  | dropWhile \_ [] = [] |
|  | dropeWhile p (x:xs) | p x = dropWhile p xs |
|  | | otherwise = xs |
|  |  |
|  | --1e) |
|  | filter :: (a -> Bool) -> [a] -> [a] |
|  | filter \_ [] = [] |
|  | filter p (x:xs) | p x = (x : (filter p xs)) |
|  | | otherwise = (filter p xs) |
|  |  |
|  | --1f) |
|  | foldr1 :: (a -> a -> a) -> [a] -> a |
|  | foldr1 \_ [x] = x |
|  | foldr1 op (x:xs) = op x (foldr1 op xs) |
|  |  |
|  |  |
|  | ------------Q2-------------- |
|  |  |
|  | --2a) |
|  | hof :: (a -> b -> b) -> b -> [a] -> b |
|  | hof \_ e [] = e |
|  | hof fx e (x:xs) = fx x (hof fx e xs) |
|  |  |
|  | --2b) |
|  | f1 = hof (\*) 1 |
|  | f2 = hof f2b 0 |
|  | where f2b x y = (1+y) |
|  | f3 = hof (+) 0 |
|  | f4 = hof (++) [] |
|  | f5 = hof f5b 0 |
|  | where f5b x y = (x\*x+y) |
|  |  |
|  | --2c) |
|  | {- |
|  | Yes, hof is provided by the prelude under the name 'foldr'. |
|  | -} |
|  |  |
|  |  |
|  | ------------Q3-------------- |
|  |  |
|  | --3a) |
|  | {- |
|  | The function will fail if the 'lkp' does not return a 'Just' in the |
|  | second pattern. |
|  |  |
|  | In the fourth pattern, if 'eval d e2' evaluates to zero there will |
|  | be an runtime error because of an attempt to divide by zero. |
|  | -} |
|  |  |
|  | --3b) |
|  | eval :: Dict -> Expr -> Maybe Int |
|  | eval \_ (K i) = Just i |
|  | eval d (V s) = lkp s d |
|  | eval d (Add e1 e2) = case (eval d e1,eval d e2) of |
|  | (Nothing,\_) -> Nothing |
|  | (\_,Nothing) -> Nothing |
|  | (Just a, Just b) -> Just (a+b) |
|  | eval d (Dvd e1 e2) = case (eval d e1,eval d e2) of |
|  | (Nothing,\_) -> Nothing |
|  | (\_,Nothing) -> Nothing |
|  | (Just a, Just b) |
|  | | b == 0 -> Nothing |
|  | | otherwise -> Just (div a b) |
|  | eval d (Let v e1 e2) = case eval d e1 of |
|  | Nothing -> Nothing |
|  | Just i -> eval (ins v i d) e2 |
|  |  |
|  | --3c) |
|  | eval :: Dict -> Expr -> Either Err Int |
|  | eval \_ (K i) = Right i |
|  | eval d (V s) = case (lkp s d) of |
|  | Nothing -> Left "Function 'lkp' returned Nothing." |
|  | Just i -> Right i |
|  | eval d (Add e1 e2) = case (eval d e1,eval d e2) of |
|  | (Left msg,\_) -> Left msg |
|  | (\_,Left msg) -> Left msg |
|  | (Right a, Right b) -> Right (a+b) |
|  | eval d (Dvd e1 e2) = case (eval d e1,eval d e2) of |
|  | (Left msg,\_) -> Left msg |
|  | (\_,Left msg) -> Left msg |
|  | (Right a, Right b) |
|  | | b == 0 -> Left "Won't divide by zero- sorry! x" |
|  | | otherwise -> Right (div a b) |
|  | eval d (Let v e1 e2) = case eval d e1 of |
|  | Left msg -> Left msg |
|  | Right i -> eval (ins v i d) e2 |
|  |  |
|  |  |
|  | ------------Q4-------------- |
|  |  |
|  | --4a) |
|  | {- |
|  | Won't attempt to draw a tree in a .hs file. |
|  | -} |
|  |  |
|  | --4b) |
|  | {- |
|  | take 2 (down 42) |
|  |  |
|  | Strict: |
|  | take 2 (down 42) |
|  | take 2 (42:down 41) |
|  | take 2 (42:41:down 40) |
|  | take 2 (42:41:40:down 39) |
|  | This continues on infitely (does not stop at down 0) and therefore a result will never emerge. |
|  |  |
|  | Lazy: |
|  | take 2 (down 42) |
|  | take 2 (42:down 41) |
|  | 42: take 1 (down 41) |
|  | 42: take 1 (41:down 40) |
|  | 42: 41: take 0 (down 40) |
|  | 42: 41: [] |
|  | [42,41] |
|  |  |
|  | As you can see, only lazy evaluation will give the proper result. |
|  | -} |
|  |  |
|  | --4ci) take 0 [] |
|  |  |
|  | --4cii) There is no such expression |
|  |  |
|  | --4ciii) take 2 (down 42) |
|  |  |
|  | --4civ) down 42 |
|  |  |
|  | --4d) |
|  | fileToUpperCase = |
|  | do putStr "What file do you want to uppercase?" |
|  | from <- getLine |
|  | text <- readFile (from++".in") |
|  | writeFile (from++".ing") (toUpper text) |
|  | putStr "Done." |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |