プログラム言語特論 レポート1

author : M1 丹野雄太
number : 4555235023
date : 2023-11-30

目次

- 問題1
- 問題2
- 問題3
- 問題4
- 問題5
- 問題6
- 問題7

問題1

myTake

```
myTake :: Int -> [a] -> [a]
myTake 0 _ = []
myTake n [] = []
myTake n (x:xs) = x : myTake (n-1) xs
----------
ghci> myTake 3 [1,2,3,4,5]
[1,2,3]
ghci> myTake 0 [1,2,3,4,5]
[]
ghci> myTake 9 [1,2,3,4,5]
[1,2,3,4,5]
ghci> myTake 9 []
[]
```

- 強力なパターンマッチがあるため、ワイルドカード無いかなと思って探した所、あったのでつかってみた
- myTake n [] = []を忘れていたが、実行時エラーで気づいた
    - さすがにマッチ漏れまで検出はできないか
    - 関数自体をパターンマッチさせるから、コーナーケースを検査するの難しいのかなと、直感的に思った

myTakeWhile

```
myTakeWhile :: (a -> Bool) -> [a] -> [a]
myTakeWhile put []     = []
myTakeWhile put (x:xs) | put x = x : myTakeWhile put xs
                       | otherwise = []
-----------
ghci> myTakeWhile (<3) [1,2,3,4,5]
[1,2]
ghci> myTakeWhile (<9) [1,2,3,4,5]
[1,2,3,4,5]
ghci> myTakeWhile (<9) []
[]
```

- 定義をmyTakeWhile :: (Int -> Bool) -> [a] -> [a]と間違えていた
    - コンパイラがIntとaの型不一致を指摘したため気づけた
- この時点で、今まで学んだ再帰関数の書き方と同じ考え方ができることに気づく
    - 再帰の停止地点（基底）と、内部処理の2つを書けばよいことを理解した

myTakeDrop

```
myTakeDrop :: (a -> Bool) -> [a] -> [a]
myTakeDrop drop [] = []
myTakeDrop drop (x:xs) | drop x = myTakeDrop drop xs
                       | otherwise = x : myTakeDrop drop xs
---------------
ghci> myTakeDrop (<3) [1,2,3,4,5,6]
```

```
[3,4,5,6]
ghci> myTakeDrop (<0) [1,2,3,4,5,6]
[1,2,3,4,5,6]
ghci> myTakeDrop (<9) [1,2,3,4,5,6]
[]
ghci> myTakeDrop (<9) []
[]
```

問題2

再帰でpoly

```
recursive_poly :: [Int] -> Int -> Int
recursive_poly [] _ = 0
recursive_poly (x:xs) v = (x * v ^ (length xs)) + recursive_poly xs v
----------
ghci> recursive_poly [2,3,4] 2
18
ghci> recursive_poly [] 2
0
```

- 当たれられたリストの初項の値を計算し、後ろの項は再帰呼び出しすることで足していく
- 空リストが与えられたときは、0にすると仕様定義した
    - recursive_poly [] _ = 0が無くて、網羅的でないパターンだと怒られた

問題3

merge3

- 2つのリストをくっつけるmeger2を作るという案で行く
    - merge2はx,yの大小で分岐し、x,yが同じ値の時は、2度同じ値を入れないように注意
- さらにそのmerge2を、merge3から2回呼ぶことで動作を実現する

```
merge2 :: Ord a => [a] -> [a] -> [a]
merge2 x [] = x
merge2 [] y = y
merge2 (x:xs) (y:ys) | x < y = x : merge2 xs (y:ys)
                     | x == y = x : merge2 xs ys
                     | otherwise = y : merge2 (x:xs) ys

merge3 :: Ord a => [a] -> [a] -> [a] -> [a]
merge3 x y z = merge2 x (merge2 y z)
-----------
ghci> merge3 [1,4,7] [2,5,8] [3,5,9]
[1,2,3,4,5,7,8,9]
ghci> merge3 [1,8,9] [1,2,3,4,7] [2,5,6,10]
[1,2,3,4,5,6,7,8,9,10]
```

- 3並列のときは、2並列を2回やるとよい、という教訓を活かせた
- merge2を思いついた時、とても気持ちよかった

問題4

全く分からなかったので、後輩の渡邉こうきくんに完全に教えてもらいました ### takeXiyjzk - 本体の定義は以下の通り - dropWhileが使われているのは、kの値が大きくなると、オーバーフローして負の値が答えの配列の最初に入ってしまうため

```
takeXiyjzk :: Int -> Int -> Int -> Int -> [Int]
takeXiyjzk 0 x y z = []
takeXiyjzk k x y z = take k (dropWhile (< 1) (takeAs k x y z [1] [1] [1]))
```

---
ghci> takeXiyjzk 10 2 3 5
[1,2,3,4,5,6,8,9,10,12]

- takeAsは以下の通り
- 今のxyzと、再帰呼び出ししたtakeAsをマージする。ここで重複が消える
    - 再帰呼び出ししたtakeAsは、今のxyzにそれぞれa,b,c乗したものを計算してマージする

```
takeAs :: Int -> Int -> Int -> Int -> [Int] -> [Int] -> [Int] -> [Int]
takeAs 0 _ _ _ _ _ _ = [1]
takeAs k a b c x y z = merge2 (merge3 x y z) (takeAs (k-1) a b c (map (* a) (merge3 x y z))
                                            (map (* b) (merge3 x y z))
                                            (map (* c) (merge3 x y z)))
```

本当にこんなのよく思いついたな、と思いました 渡邉こうきくん曰く、渾身の出来だそうです ### between 時間ないので諦めます… ## 問題5 ### 最大部分列の計算量

- 全探索すると、整数リストのi番目からj番目(i < j) までの和を順に求めるため、O(N^2)
- Kadane's Algorithmを使うとO(N)で解けるもよう
    - 再帰で綺麗に実装できそうな予感 ### 最大部分積
- kadaneは適用できない気がするため、全探索で実装する

問題6

(1)片側だけでもいい二分木を定義

- 片側しかない場合のBENode1を定義することで実現した
    - 後輩と、Node1の時、この実装だと左右の区別できないのではという議論になった
    - 僕は、二分木自体が左右の区別をしないため、このような実装にした
- 両側ある場合はBENode2とした

```
data BETree a = BELeaf a
          | BENode1 a (BETree a)
          | BENode2 a (BETree a) (BETree a)
          deriving(Show)
```

(2)3つの関数を作成

depthBETree

- 関数の目的は、引数の木t以下の深さを計算して返すこと
- Leafのときは、深さはカウントしない。
- Nodeの時は、深さカウントを1増やす
    - 今回は深さの最大値を求めたいため、Node2の時は大きい方の値を使用する

```
depthBETree :: BETree t -> Int
depthBETree (BELeaf a) = 0
depthBETree (BENode1 a t) = (depthBETree t) + 1
depthBETree (BENode2 a tl tr) = max (depthBETree tl + 1) (depthBETree tr + 1)
---------------
ghci> depthBETree (BENode2 5 (BENode2 8 (BELeaf 3) (BENode1 1 (BELeaf 7))) (BENode2 6 (BENode1 2
(BELeaf 9)) (BELeaf 4)))
3
```

sumBETree

- 関数の目的は、引数の木t以下の和を返すこと
- Leafの時は、足すものもないので、その値を返す
- Nodeの時は、自分の値を足しつつ、子のノードを再帰呼び出しし、その返り値を足す

```
sumBETree :: Num t => BETree t -> t
sumBETree (BELeaf a) = a
```

```
sumBETree (BENode1 a t) = a + (sumBETree t)
sumBETree (BENode2 a tl tr) = a + sumBETree tl + sumBETree tr
-------------
ghci> sumBETree (BENode2 5 (BENode2 8 (BELeaf 3) (BENode1 1 (BELeaf 7))) (BENode2 6 (BENode1 2 (BELeaf
9)) (BELeaf 4)))
45
```

upAccBETree

- 関数の目的は、引数の木t以下に所定の計算をすること
- Leafのときは、そのままLeafを返す
- Nodeの時は、引数の木以下の和に自分の値を足したものを自分の値にする。
- 更に、下にupAccを伝播させるために再帰呼び出しする
    - これを忘れていて、後輩の渡邉くんと野村くんに教えてもらった

```
upAccBETree :: Num t => BETree t -> BETree t
upAccBETree (BELeaf a) = (BELeaf a)
upAccBETree (BENode1 a t) = (BENode1 (a + (sumBETree t)) (upAccBETree t))
upAccBETree (BENode2 a tl tr) = (BENode2 (a + (sumBETree tl) + (sumBETree tr)) (upAccBETree tl)
(upAccBETree tr))
-------
ghci> upAccBETree (BENode2 5 (BENode2 8 (BELeaf 3) (BENode1 1 (BELeaf 7))) (BENode2 6 (BENode1 2
(BELeaf 9)) (BELeaf 4)))
BENode2 45 (BENode2 19 (BELeaf 3) (BENode1 8 (BELeaf 7))) (BENode2 21 (BENode1 11 (BELeaf 9))
(BELeaf 4))
```

(3)(4) cataBETreeで実装

```
cataBETree :: (a -> b) -> (a -> b -> b) -> (a -> b -> b -> b) -> BETree a -> b
cataBETree f g h (BELeaf x) = f x
cataBETree f g h (BENode1 x t) = g x (cataBETree f g h t)
cataBETree f g h (BENode2 x tl tr) = h x (cataBETree f g h tl) (cataBETree f g h tr)
```

depthBETree

- fが適用される場合、`cataBETree f g h (BELeaf x) = f x`を見る
    - 今回は、0を返すだけである
- gが適用される場合、`cataBETree f g h (BENode1 x t) = g x (cataBETree f g h t)`を見る
    - 今回は、+1する
- hが適用される場合、`cataBETree f g h (BENode2 x tl tr) = h x (cataBETree f g h tl) (cataBETree f g h tr)`を見る
    - 今回は、yが`(cataBETree f g h tl)`に対応しているため、+1する。
    - その後、max

```
depthBETree :: BETree t -> Int
depthBETree t = cataBETree (\x -> 0) (\x y -> y + 1) (\x y z -> max (y+1) (z+1)) t
----------
ghci> depthBETree (BENode2 5 (BENode2 8 (BELeaf 3) (BENode1 1 (BELeaf 7))) (BENode2 6 (BENode1 2
(BELeaf 9)) (BELeaf 4)))
3
```

sumBETree

- fが適用される場合、`cataBETree f g h (BELeaf x) = f x`を見る
    - 今回はxを返すだけ
- gが適用される場合、`cataBETree f g h (BENode1 x t) = g x (cataBETree f g h t)`を見る
    - 今回は、xにcataBETree(sumBETree)を足していく
- hが適用される場合、`cataBETree f g h (BENode2 x tl tr) = h x (cataBETree f g h tl) (cataBETree f g h tr)`を見る
    - 今回は、xにcataBETree(sumBETree)を足していく

```
sumBETree :: Num t => BETree t -> t
sumBETree t = cataBETree (\x -> x) (\x y -> x + y) (\x y z -> x + y + z) t
```

---
ghci> sumBETree (BENode2 5 (BENode2 8 (BELeaf 3) (BENode1 1 (BELeaf 7))) (BENode2 6 (BENode1 2 (BELeaf 9)) (BELeaf 4)))
45

upAccBETree

- 時間ないので諦めました… ## 問題7
- 時間ないので諦めました…