Национальный исследовательский университет ИТМО

Факультет Программной Инженерии и Компьютерной Техники

Вариант №8,23

Лабораторная работа №1

По дисциплине:

«Функциональное программирование»

Работу выполнила:

Студентка группы P34102

Никонова Наталья Игоревна

Преподаватель:

Пенской Александр Владимирович

Санкт-Петербург

2023

Цель

Освоить базовые приёмы и абстракции функционального программирования: функции, поток управления и поток данных, сопоставление с образцом, рекурсия, свёртка, отображение, работа с функциями как с данными, списки.

Задание

В рамках лабораторной работы вам предлагается решить несколько задач проекта Эйлер. Список задач -- ваш вариант.

Для каждой проблемы должно быть представлено несколько решений:

1. монолитные реализации с использованием:
    a. хвостовой рекурсии;
    b. рекурсии (вариант с хвостовой рекурсией не является примером рекурсии);
2. модульной реализации, где явно разделена генерация последовательности, фильтрация и свёртка (должны использоваться функции reduce/fold, filter и аналогичные);
3. генерация последовательности при помощи отображения (map);
4. работа со спец. синтаксисом для циклов (где применимо);
5. работа с бесконечными списками для языков, поддерживающих ленивые коллекции или итераторы как часть языка (к примеру Haskell, Clojure);
6. реализация на любом удобном для вас традиционном языке программирования для сравнения.

Требуется использовать идиоматичный для технологии стиль программирования.

Вариант

Задача 8. Максимальное произведение

Наибольшее произведение четырех последовательных цифр в нижеприведенном 1000-значном числе равно 9 × 9 × 8 × 9 = 5832.

73167176531330624919225119674426574742355349194934969835203127745063262395783180169848018694788518438586156078911294949545950173795833195285320880551112540698747158523863050715693290963295227443043557668966489504452445231617318564030987111217223831136222989342338030813533627661428280644448664523874930358907296290491560440772390713810515859307960866701724271218839987979087922749219016997208880937766572733300105336788122023542180975125454059475224352584907711670556013604839586446706324415722155397536978179778461740649551492908625693219784686224828397224137565705605749026140797296865241453510047482166370484403199890008895243450658541227588666881164271714799244429282308634656748139191231628245861786645835912456652947654568284891288314260769004224219022671055626321111109370544217506941658960408071984038509624554443629812309878799272442849091888458015616609791913387549920052406368991256071760605886116467109405077541002256983155200055935729725716362695618826704282524836008232575304207529634 50

Найдите наибольшее произведение тринадцати последовательных цифр в данном числе.

Задача 23. Сумма избыточных чисел

Идеальным числом называется число, у которого сумма его делителей равна самому числу. Например, сумма делителей числа 28 равна 1 + 2 + 4 + 7 + 14 = 28, что означает, что число 28 является идеальным числом.

Число n называется недостаточным, если сумма его делителей меньше n, и называется избыточным, если сумма его делителей больше n.

Так как число 12 является наименьшим избыточным числом (1 + 2 + 3 + 4 + 6 = 16), наименьшее число, которое может быть записано как сумма двух избыточных чисел, равно 24. Используя математический анализ, можно показать, что все целые числа больше 28123 могут быть записаны как сумма двух избыточных чисел. Эта граница не может быть уменьшена дальнейшим анализом, даже несмотря на то, что наибольшее число, которое не может быть записано как сумма двух избыточных чисел, меньше этой границы.

Найдите сумму всех положительных чисел, которые не могут быть записаны как сумма двух избыточных чисел.

Выполнение

Репозиторий с кодом: https://github.com/nanikon/functional-programming

Задание 8.

Решение через рекурсию

```haskell
task8Recursion :: [Int] -> Int
task8Recursion [] = error "List must have 13 elements"
task8Recursion [_] = error "List must have 13 elements"
task8Recursion [_, _] = error "List must have 13 elements"
task8Recursion [_, _, _] = error "List must have 13 elements"
task8Recursion [_, _, _, _] = error "List must have 13 elements"
task8Recursion [_, _, _, _, _] = error "List must have 13 elements"
task8Recursion [_, _, _, _, _, _] = error "List must have 13 elements"
task8Recursion [_, _, _, _, _, _, _] = error "List must have 13 elements"
task8Recursion [_, _, _, _, _, _, _, _] = error "List must have 13 elements"
task8Recursion [_, _, _, _, _, _, _, _, _] = error "List must have 13 elements"
task8Recursion [_, _, _, _, _, _, _, _, _, _] = error "List must have 13 elements"
task8Recursion [_, _, _, _, _, _, _, _, _, _, _] = error "List must have 13 elements"
task8Recursion [_, _, _, _, _, _, _, _, _, _, _, _] = error "List must have 13 elements"
task8Recursion [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13] = product [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13]
task8Recursion (x1 : x2 : x3 : x4 : x5 : x6 : x7 : x8 : x9 : x10 : x11 : x12 : x13 : xs) =
  max (product [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13]) (task8Recursion (x2 : x3 : x4 : x5 : x6 : x7 : x8 : x9 : x10 : x11 : x12 : x13 : xs))
```

Просто перебираем все произведения 13-ти элементных подсписков, и на каждом шаге рекурсии выбираем какое из них максимальное – текущее или следующее?

Также здесь было использовано сопоставление с образцом чтобы отсечь слишком короткие списки.

При хвостовой рекурсии решение аналогично, только добавляется ещё один параметр – текущее максимальное произведение, которое обновляется на каждом шаге. А также попробовала вложенные функции.

```haskell
task8_TailRec :: [Int] -> Int
task8_TailRec = maxMult 0
  where
    maxMult _ [] = error "List must have 13 elements"
    maxMult _ [_] = error "List must have 13 elements"
    maxMult _ [_, _] = error "List must have 13 elements"
    maxMult _ [_, _, _] = error "List must have 13 elements"
    maxMult _ [_, _, _, _] = error "List must have 13 elements"
    maxMult _ [_, _, _, _, _] = error "List must have 13 elements"
    maxMult _ [_, _, _, _, _, _] = error "List must have 13 elements"
    maxMult _ [_, _, _, _, _, _, _] = error "List must have 13 elements"
    maxMult _ [_, _, _, _, _, _, _, _] = error "List must have 13 elements"
    maxMult _ [_, _, _, _, _, _, _, _, _] = error "List must have 13 elements"
    maxMult _ [_, _, _, _, _, _, _, _, _, _] = error "List must have 13 elements"
    maxMult _ [_, _, _, _, _, _, _, _, _, _, _] = error "List must have 13 elements"
    maxMult _ [_, _, _, _, _, _, _, _, _, _, _, _] = error "List must have 13 elements"
    maxMult n [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13] = max n $ product [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13]
    maxMult n (x1 : x2 : x3 : x4 : x5 : x6 : x7 : x8 : x9 : x10 : x11 : x12 : x13 : xs) =
        maxMult (max (product [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13]) n) (x2 : x3 : x4 : x5 : x6 : x7 : x8 : x9 : x10 : x11 : x12 : x13 : xs)
```

При использовании сверток и фильтров было интересно понять, что аккумулятор в свертке можно использовать для построения нового массива. Также с помощью фильтра добавила проверку на наличие нуля в множителях, чтобы не перемножать лишнее.

```haskell
groupByNFold :: Int -> [Int] -> [[Int]]
groupByNFold n =
    foldr
        ( \el acc ->
            case acc of
                (_ : _) ->
                    foldr
                        ( \curList newList ->
                            if length curList < n then (el : curList) : newList else curList : newList
                        )
                        [[el]]
                        acc
                _ -> [[el]]
        )
        []

filterShortElem :: Int -> [[Int]] -> [[Int]]
filterShortElem n = filter (\x -> length x >= n)

filterWithout0 :: [[Int]] -> [[Int]]
filterWithout0 = filter (notElem 0)
```

При использовании мапы я также попробовала использовать условные конструкции.

```haskell
groupByNMap :: Int -> [Int] -> [[Int]]
groupByNMap n list =
    if n > length list
        then error "List must have more or equal n elements"
        else map (\x -> take n (drop x list)) [0 .. length list - n]

task8Map :: [Int] -> Int
task8Map = maximum . map product . groupByNMap 13
```

А при использовании ленивых вычислений – генераторы списков

```haskell
task8Iterate :: [Int] -> Int
task8Iterate list = maximum (take (length list - 13) [product (take 13 (drop x list)) | x <- [0 ..]
```

Решение привычном языке – python

```python
def task8(input):
    if (len(input)) < 13:
        raise ValueException("List must have 13 elements minimum")
    max_product = 0
    for i in range(0,len(input)-12):
        cur_product = input[i]
        for j in range(1,13):
            cur_product *= input[i+j]
        if cur_product > max_product:
            max_product = cur_product
    return max_product
```

Задача 23.

Эта задача на мой взгляд слишком сложная чтобы решать без встроенных функций, так что я начала сразу со свертки.

```haskell
divisionSum :: Int -> Int
divisionSum n =
    foldr
        ( \x acc ->
            let t = n `div` x
            in if n `mod` x == 0
                    then
                        if x == t
                            then acc + x
                            else acc + x + t
                    else acc
        )
        1
        [2 .. intSqrt n]

isAbundant :: Int -> Bool
isAbundant = memo (\n -> divisionSum n > n)

isSumAb :: Int -> Bool
isSumAb n = any (\k -> isAbundant (n - k)) (filter isAbundant [1 .. n - 1])

task23Spec :: Int -> Int
task23Spec limit = sum $ filter (not . isSumAb) [1 .. limit]
```

Вариант с map

```haskell
divisionsByMap :: Int -> [Int]
divisionsByMap n =
    1
        : map
            ( \x ->
                let t = n `div` x
                in if n `mod` x == 0
                        then
                            if x == t
                                then x
                                else x + t
                        else 0
            )
            [2 .. intSqrt n]

divisionSumMap :: Int -> Int
divisionSumMap = sum . divisionsByMap

isAbundantMap :: Int -> Bool
isAbundantMap = memo (\n -> divisionSumMap n > n)

isSumAbMap :: Int -> Bool
isSumAbMap n = any (\k -> isAbundantMap (n - k)) [x | x <- [1 .. n - 1], isAbundantMap x]

task23Map :: Int -> Int
task23Map limit =
    sum $
        map
            ( \x ->
                if not (isSumAbMap x)
                    then x
                    else 0
            )
            [1 .. limit]
```

Вариант с ленивыми вычислениями – частично используются функции с варианта со сверткой.

```haskell
abundantNums :: [Int]
abundantNums = filter isAbundant [1 ..]

isSumTwoAb :: Int -> Bool
isSumTwoAb n = any (\k -> isAbundant (n - k)) (takeWhile (< n) abundantNums)

task23Lazy :: Int -> Int
task23Lazy limit = sum $ filter (not . isSumTwoAb) (takeWhile (<= limit) [1 ..])
```

Решение на Python

```python
def task23(input):
    limit = input + 1
    divisors_sum = [0] * limit
    for i in range(1, limit):
        for j in range(i * 2, limit, i):
            divisors_sum[j] += i
    ab_nums = [i for (i, x) in enumerate(divisors_sum) if x > i]

    sums = [False] * limit
    for i in ab_nums:
        for j in ab_nums:
            if i + j < limit:
                sums[i + j] = True
            else:
                break
    ans = sum(i for (i, x) in enumerate(sums) if not x)
    return ans
```

Вывод

В ходе выполнения лабораторной работы я познакомилась с парадигмой функционального программирования. Самым сложным для меня в ней оказалась неизменяемость данных, так как для уменьшения времени выполнения я привыкла в одном цикле делать несколько задач параллельно (решение 23 задачи на python например). Также вначале я часто путалась в типах, так что хорошо, что мой язык имеет статическую типизацию и все несоответствия показывал на этапе компиляции.