Лабораторная работа #1

Дисциплина: "Функциональное программирование"

Дата: 2021/10

Выполнил: Федоров Сергей, P34113

Название: "Решение задач Project Euler"

Цель работы: Решить две задачи project euler, различными способами используя выбранный ранее функциональный ЯП. 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.

Вариант:

ЯП	**Clojure**
Задания	10, 21
ЯП по выбору	Scala

Требования:

1. монолитные реализации с использованием:
    - хвостовой рекурсии
    - 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 программировании для сравнения.

Выполнение

[Ссылка на репозиторий]

Условия задач

1. Задача 10:

```
- The sum of the primes below 10 is 2 + 3 + 5 + 7 = 17
- Find the sum of all the primes below two million.
```

2. Задача 21:

```
Let d(n) be defined as the sum of proper divisors of n
(numbers less than n which divide evenly into n).
If d(a) = b and d(b) = a, where a ≠ b,
then a and b are an amicable pair and each of a and b
are called amicable numbers.

For example, the proper divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 5
Therefore d(220) = 284. The proper divisors of 284 are 1, 2, 4, 71 and 142
So d(284) = 220.

Evaluate the sum of all the amicable numbers under 10000.
```

Программная реализация

Структура проекта:

```
.
├── project.clj
├── resources
│   └── scala // moved to resources in order not to interfere with lein
│       └── functional_programming_itmo_2021
│           └── lab_1
│               ├── Main.scala
│               ├── Task10.scala
│               └── Task21.scala
└── src
    └── functional_programming_itmo_2021
        ├── core
        │   └── sequences.clj
        └── lab_1
```

```
            ├── main.clj
            ├── task_10.clj
            └── task_21.clj
```

Настройка Clojure-окружения с помощью Leiningen:

`project.clj`

```
(defproject functional-programming-itmo-2021 "0.0.1"
  :dependencies [[org.clojure/clojure "1.10.1"]
                 [nrepl/lein-nrepl "0.3.2"]]
  :profiles {
     :lab_1 {
         :repl-options {
                :init-ns functional-programming-itmo-2021.lab-1.main
                :package functional-programming-itmo-2021.lab-1
             }
         :main functional-programming-itmo-2021.lab-1.main
         }
     }
  )
```

Выполнения задания №10:

1. Реализованное решето Эратосфена, для бесконечного поиска простых чисел. Более примитивные алгоритмы загибались на числах близких к *1E+6*:

`sequences.clj`

```
(ns functional-programming-itmo-2021.core.sequences)

; Implementation of Sieve of Eratosthenes

(defn ^:private enqueue [sieve candidate step]
   (let [m (+ candidate step)]
    (if (sieve m)
      (recur sieve m step)
      (assoc sieve m step)))
  )

(defn ^:private next-sieve [sieve candidate]
   (if-let [step (sieve candidate)]
    (-> (dissoc sieve candidate)                          ; Macro magic O.o
        (enqueue candidate step))
    (enqueue sieve candidate (* 2 candidate))
```

```
    )
  )

(defn ^:private next-primes [sieve candidate]
   (if (sieve candidate)
    (recur (next-sieve sieve candidate) (+ 2 candidate))
    (cons candidate
          (lazy-seq (next-primes (next-sieve sieve candidate) (+ 2 candidate)))
    )
  )

(def primes (concat [2] (next-primes {} 3)))                ; Lazy seq of prime
```

2. Решенное задание, с использованием операций преобразования коллекций, рекурсии, циклов и монолитная:

task_10.clj

```
(ns functional-programming-itmo-2021.lab-1.task-10
  (:require [functional-programming-itmo-2021.core.sequences :refer [primes]])
  )

(def task-threshold (int 2E+6))                              ;; Task answer: 142

(defn sum-of-primes-below-n-reducing
  "Finds the sum of all the primes bellow N
   Using the reduce operation
  "
  [n]
  (reduce + 0N (take-while #(< % n) primes)))

(defn sum-of-primes-below-n-loopy
  "Finds the sum of all the primes bellow N
   Using the loop/recur operators
  "
  [n]
  (loop [accumulator 0N
         next-prime (first primes)
         others (rest primes)]
    (if
      (< next-prime n)
      (recur (+ accumulator next-prime) (first others) (rest others))
      accumulator
      )
    )
  )
```

```
(defn sum-of-primes-below-n-recursive
  "Finds the sum of all the primes bellow N
   Using the standard- and tail- recursion
  "
  ([n] (sum-of-primes-below-n-recursive n 0N (first primes) (rest primes)))
  ([n acc next-prime others]
   (if
     (< next-prime n)
     (recur n (+ acc next-prime) (first others) (rest others))
     acc
     )
   )
  )

(defn sum-of-primes-below-n-non-modular
  "Finds the sum of all the primes bellow N
   Bulk implementation
  "
  [n]
  (let [
        enqueue (fn [sieve candidate step]
                  (let [m (+ candidate step)]
                    (if (sieve m)
                      (recur sieve m step)
                      (assoc sieve m step)))
                  )
        next-sieve (fn [sieve candidate]
                     (if-let [step (sieve candidate)]
                       (-> (dissoc sieve candidate)
                           (enqueue candidate step))
                       (enqueue sieve candidate (* 2 candidate))
                       )
                     )
        next-primes (fn n-p [sieve candidate]
                      (if (sieve candidate)
                        (recur (next-sieve sieve candidate) (+ 2 candidate))
                        [candidate [(next-sieve sieve candidate) (+ 2 candidate
                        )
                      )]
    (loop [accumulator 0N
           next-prime 2
           params [{} 3]]
      (if
        (< next-prime n)
        (let [[n-p params] (apply next-primes params)]
            (recur (+ accumulator next-prime) n-p params)
          )
```

```
        accumulator
        )
      )
    )
  )
```

3. Альтернативное решение одним из способов на выбранном языке (Scala):

`Task10.scala`

```
package scala.functional_programming_itmo_2021.lab_1

import Numeric.Implicits.given
import Ordering.Implicits.given

object Task10:
  def taskThreshold[N: Numeric] = Numeric[N].fromInt(2_000_000)

  def primes[N: Numeric]: LazyList[N] = {
    // Implementation of Sieve of Eratosthenes

    def enqueue(sieve: Map[N, N], candidate: N, step: N): Map[N, N] =
      if (sieve contains (candidate + step))
        enqueue(sieve, candidate + step, step)
      else
        sieve + (candidate + step -> step)

    def nextSieve(sieve: Map[N, N], candidate: N): Map[N, N] =
      sieve get candidate match {
        case Some(step) => enqueue(sieve - candidate, candidate, step)
        case None       => enqueue(sieve, candidate, candidate * Numeric[N].fro
      }

    def nextPrimes(sieve: Map[N, N], candidate: N): LazyList[N] =
      if (sieve contains candidate)
        nextPrimes(nextSieve(sieve, candidate), candidate + Numeric[N].fromInt(
      else
        LazyList(candidate) lazyAppendedAll
          nextPrimes(nextSieve(sieve, candidate), candidate + Numeric[N].fromIn

    Numeric[N].fromInt(2) #:: nextPrimes(Map.empty, Numeric[N].fromInt(3))
  }

  def sumOfPrimesBelowN[N: Numeric: Ordering](n: N): N =
    primes[N].takeWhile(_ < n).foldLeft(Numeric[N].fromInt(0))(_ + _)
```

Выполнения задания №21:

1. Решенное задание, с использованием операций преобразования коллекций, рекурсии, циклов и монолитная:

task_21.clj

```
(ns functional-programming-itmo-2021.lab-1.task-21)

(def task-threshold 1E+4)

(defn ^:private divisors-of-num
  ([num] (divisors-of-num num 0 []))
  ([num last divisors]
   (let [next (inc last)] (if (< next num)
       (if (zero? (rem num next))
         (recur num next (cons next divisors))
         (recur num next divisors))
       divisors
       )))
  )

(defn finding-amicable-numbers-lazy-mapped
  "Finds pairs of amicable numbers, up to threshold
   Using collection operators: map, filter, sort, distinct
  "
  [threshold]
  (let [divisors-sums (fn [coll] (map #(reduce + (divisors-of-num %)) coll))
        numbers (range 1 (inc threshold))
        first-row (divisors-sums numbers)
        second-row (divisors-sums first-row)]
      (distinct (map #(sort [(% 0) (% 1)])
                     (filter #(and (not= (% 0) (% 1)) (== (% 0) (% 2)))
                                  (map vector numbers first-row second-row))))
    )
  )

(defn finding-amicable-numbers-recursive
  "Finds pairs of amicable numbers, up to threshold
   Using standard- and tail- recursion
  "
  ([threshold] (finding-amicable-numbers-recursive #{} threshold 1))
  ([acc threshold current]
    (let [divisors-sum #(reduce + (divisors-of-num %))
          fst-iter (divisors-sum current)
          snd-iter (divisors-sum fst-iter)
          next-acc (if
                     (and (not= current fst-iter) (== current snd-iter))
```

```
                    (conj acc (sort [current fst-iter]))
                    acc
                    )]
        (if (>= current threshold) next-acc (recur next-acc threshold (inc curr
      )
    )
  )

(defn finding-amicable-numbers-loopy
  "Finds pairs of amicable numbers, up to threshold
   Using loop/recur operators
  "
  [threshold]
  (loop [acc #{}
         current 1]
   (let [divisors-sum #(reduce + (divisors-of-num %))
         fst-iter (divisors-sum current)
         snd-iter (divisors-sum fst-iter)
         next-acc (if
                    (and (not= current fst-iter) (== current snd-iter))
                    (conj acc (sort [current fst-iter]))
                    acc
                    )]
     (if (>= current threshold) next-acc (recur next-acc (inc current)))
     )
   )
  )

(defn finding-amicable-numbers-non-modular
  "Finds pairs of amicable numbers, up to threshold
   Bulk implementation
  "
  [threshold]
  (let [divisors (fn [num last divisors]
                   (let [next (inc last)] (if (< next num)
                       (if (zero? (rem num next))
                         (recur num next (cons next divisors))
                         (recur num next divisors))
                       divisors
                       )))
        divisors-sums (fn [coll] (map #(reduce + (divisors % 0 [])) coll))
        numbers (range 1 (inc threshold))
        first-row (divisors-sums numbers)
        second-row (divisors-sums first-row)]
    (distinct (map #(sort [(% 0) (% 1)])
                   (filter #(and (not= (% 0) (% 1)) (== (% 0) (% 2)))
                           (map vector numbers first-row second-row))))
    )
```

```
  )
```

3. Альтернативное решение одним из способов на выбранном языке (Scala):

`Task21.scala`

```
package scala.functional_programming_itmo_2021.lab_1

import Integral.Implicits.given
import Ordering.Implicits.given

object Task21:
  def taskThreshold[N: Integral] = Numeric[N].fromInt(10_000)

  // Recursion would've been faster, but i chose more idiomatic way
  def divisorsSums[N: Integral](coll: Seq[N]): Seq[N] =
    coll.map( n =>
      Iterator.from(1, 1)
      .map(Numeric[N].fromInt)
      .takeWhile(_ < n)
      .filter(d => (n % d) == Numeric[N].fromInt(0))
      .foldLeft(Numeric[N].fromInt(0))(_ + _))


  def findingAmicableNumbers[N: Integral: Ordering](threshold: N): Set[(N, N)] 
    val numbers   = Iterator.from(1, 1).map(Numeric[N].fromInt).takeWhile(_ <= 
    val firstRow  = divisorsSums(numbers)
    val secondRow = divisorsSums(firstRow)

    numbers.lazyZip(firstRow).lazyZip(secondRow).iterator.collect{
      case (num, fst, snd) if num != fst && num == snd =>
        if (num > fst) fst -> num else num -> fst
    }.toSet
  }
```

Точки вхождения и результаты программ:

1. Main'ы для реализации на Clojure и Scala:

`main.clj`

```
(ns functional-programming-itmo-2021.lab-1.main
  (:require [functional-programming-itmo-2021.lab-1.task-10 :refer [task-10-rep
            [functional-programming-itmo-2021.lab-1.task-21 :refer [task-21-rep
```

```
  )

(defn -main [] (do
                 (task-10-report)
                 (task-21-report)
                 ))
```

Main.scala

```
package scala.functional_programming_itmo_2021.lab_1

import Task10.task10Report
import Task21.task21Report

object Main extends App:
  println(task10Report[BigInt])
  println(task21Report[Int])
```

2. Результаты выполнения:
    - Clojure:

```
Task 10 solutions:
              * sum-of-primes-below-n-reducing
              * sum-of-primes-below-n-loopy
              * sum-of-primes-below-n-recursive
              * sum-of-primes-below-n-non-modular
"Elapsed time: 1234.891474 msecs"
"Elapsed time: 20.818945 msecs" <- faster due to memoization
"Elapsed time: 56.900111 msecs" <- faster due to memoization
"Elapsed time: 1173.162714 msecs"
Results are equal: true
Solution: 142913828922N

Task 21 solutions:
              * finding-amicable-numbers-lazy-mapped
              * finding-amicable-numbers-recursive
              * finding-amicable-numbers-loopy
              * finding-amicable-numbers-non-modular
"Elapsed time: 1527.301397 msecs"
"Elapsed time: 1090.769127 msecs"
"Elapsed time: 1061.077565 msecs"
"Elapsed time: 1215.730148 msecs"
Results are equal: true
Solution: ((220 284) (1184 1210) (2620 2924) (5020 5564) (6232 6368))
```

```
Process finished with exit code 0
```

- Scala:

```
Task 10 solution:
 * sumOfPrimesBelowN
Finished in: 1301 ms
Solution: 142913828922

Task 21 solution:
 * findingAmicableNumbers
Finished in: 2047 ms
Solution: HashSet((1184,1210), (5020,5564), (2620,2924), (220,284), (6232,63

Process finished with exit code 0
```