ЗВІТ З ЛАБОРАТОРНОЇ РОБОТИ №2

За курсом «Функціональне та теоретичне програмування»

Студента групи ПА-19-2

Ільяшенко Єгора Віталійовича

Кафедра комп’ютерних технологій, ДНУ

Тема: "Штучні типи в програмування на мові OCAML"

Text

Description automatically generated

Варіант 14

14) На основі списку натуральних чисел сформувати новий список так, що для кожного елементу N заданого списку елемент нового списку це підсписок, елементи якого – прості дільники N.

Опис тестових прикладів

Graphical user interface, text

Description automatically generated

Висновки

Виконуючи лабораторну роботу №2 з дисципліни «Функціональне та теоретичне програмування», я навчився розробляти складні програми на мові програмування Ocaml, вводити дані з файлу, з командного рядка. Розібрався із кастомними типами даних, рекурсивними списками та виконав індивідуальне завдання.

Вихідний текст програми розв'язку задачі

type variable =

| IntAtom of int

| FloatAtom of float

| List of variable list

let rec printList myList =

match myList with

| IntAtom atom ->

print\_int atom;

print\_string " "

| FloatAtom atom ->

print\_float atom;

print\_string " "

| List lst ->

print\_string "[ ";

let rec printIns innerList =

match innerList with

| [] ->

()

| value :: newList ->

printList value;

printIns newList in

printIns lst;

print\_string "] ";;

let isPrime number =

match number with

| IntAtom(atomNumber) ->

if atomNumber <= 1 then

false

else

let rec innerFunction num =

if num <= 1 then

true

else if atomNumber mod num = 0 then

false

else

innerFunction (num - 1) in

innerFunction (atomNumber - 1)

| FloatAtom(atomNumber) ->

if int\_of\_float(atomNumber) <= 1 then

false

else

let rec innerFunction num =

if num <= 1 then

true

else if int\_of\_float(atomNumber) mod num = 0 then

false

else

innerFunction (num - 1) in

innerFunction (int\_of\_float(atomNumber) - 1)

| \_ -> false;;

let primeFactorization number =

match number with

| IntAtom(atomNumber) ->

let rec innerFunction incrementNumber outputList =

if incrementNumber >= atomNumber then

IntAtom(1) :: outputList

else

if atomNumber mod incrementNumber = 0 && isPrime(IntAtom(incrementNumber)) then

innerFunction (incrementNumber + 1) (IntAtom(incrementNumber) :: outputList)

else

innerFunction (incrementNumber + 1) outputList in

innerFunction 2 []

| FloatAtom(atomNumber) ->

let rec innerFunction incrementNumber outputList =

if incrementNumber >= int\_of\_float(atomNumber) then

IntAtom(1) :: outputList

else

if int\_of\_float(atomNumber) mod incrementNumber = 0 && isPrime(IntAtom(incrementNumber)) then

innerFunction (incrementNumber + 1) (IntAtom(incrementNumber) :: outputList)

else

innerFunction (incrementNumber + 1) outputList in

innerFunction 2 []

| \_ -> [];;

let concatList list1 list2 =

match list1, list2 with

| List(h1::t1), List(h2::t2) ->

(h1::t1) @ (h2::t2)

| List[], List(h2::t2) ->

[h2] @ t2

| List(h1::t1), List[] ->

[h1] @ t1

| List[], List[] ->

[]

| \_ ->

[];;

let rec myTask myList =

match myList with

| List[] ->

List []

| List(List h :: t) ->

List(concatList (List[(myTask (List h))]) (myTask (List t)))

| List(h :: t) ->

List(concatList (List[List(primeFactorization h)]) (myTask (List t)))

| \_ ->

List[];;

(\* - - - Test part - - - \*)

let rec printStringByChar sd =

let listOfString = List.of\_seq(String.to\_seq(sd)) in

let rec innerFunc charList =

match charList with

| [] ->

()

| h::t ->

print\_char h;

innerFunc t in

innerFunc listOfString;;

(\* Input \*)

let myIntList =

List [

IntAtom 8;

IntAtom 1;

List [

IntAtom 14;

List [IntAtom 2]];

IntAtom 5];;

let myFloatList =

List [

FloatAtom 8.;

FloatAtom 1.;

List [

FloatAtom 15.;

List [FloatAtom 2.]];

FloatAtom 5.];;

(\* - - - Test part - - - \*)

let stringView1 = "[ 5. 57. [ 2. 333. ] ]";;

let stringView2 = "[ 5 57 [ 2 3 ] ]";;

let stringView3 = "[ 5 57 [ 2 3 ] 1 ]";;

let stringView4 = "[ 5 57[ 2 [ 333 32 ] 32 ] 32 32 ]";;

let stringView5 = "[ 5 57 [ 2 333 ] ]";;

(\*

let listParser stringView =

(\*Char list\*)

let listOfString = List.of\_seq(String.to\_seq(stringView)) in

let rec charIterator charList myList =

match charList with

| [] -> List[]

| h::t ->

match h with

| '[' ->

List[(concatList(myList (List[charIterator t myList])))]

| ']' ->

myList

| ' ' ->

charIterator t (myList)

| ch ->

List[IntAtom(int\_of\_char(ch))]

| \_ ->

List[]

in

charIterator listOfString List[];;

\*)

(\* Какое-то работоспособное состояние

let listParser stringView =

let rec charIterator pos myList =

if pos < String.length stringView then

match stringView.[pos] with

| '[' ->

(\*

let newPos = pos + 1 in

let charIteratorResult = charIterator newPos (List[]) in

(List(concatList myList (List[(fst (charIteratorResult))])), snd charIteratorResult)

\*)

let newPos = pos + 1 in

let charIteratorResult = charIterator newPos (List[]) in

((List[(fst (charIteratorResult))]), snd charIteratorResult)

| ']' ->

let newPos = pos + 1 in

let res = charIterator newPos (List[]) in

(List(concatList myList (fst (res))), snd res)

| ' ' ->

let newPos = pos + 1 in

charIterator newPos myList

| ch ->

let rec digitCollector pos =

let newPos = pos in

match stringView.[pos] with

| ' ' -> ([], newPos)

| ']' -> ([], newPos)

| ch -> (ch :: (fst (digitCollector (newPos + 1))), (snd (digitCollector (newPos + 1))))

in

let digitCollectorResult = digitCollector pos in

let stringNumber = String.of\_seq (List.to\_seq ((fst digitCollectorResult))) in

(List(concatList (List[IntAtom(int\_of\_string stringNumber)]) (fst (charIterator (snd digitCollectorResult) (List[])))), (snd digitCollectorResult))

(\*let newPos = pos + 1 in

List(concatList (List[IntAtom(int\_of\_char(ch) - int\_of\_char('0'))]) (charIterator newPos (List[])))\*)

else (myList, 0)

in

match fst (charIterator 0 (List[])) with

| List(h::t) -> h

| \_ -> List[];;

\*)

let listParser stringView =

let rec charIterator pos myList =

if pos < String.length stringView then

match stringView.[pos] with

| '[' ->

(\*

let newPos = pos + 1 in

let charIteratorResult = charIterator newPos (List[]) in

(List(concatList myList (List[(fst (charIteratorResult))])), snd charIteratorResult)

\*)

let newPos = pos + 1 in

let charIteratorResult = charIterator newPos (List[]) in

(List(concatList myList (List[(fst (charIteratorResult))])), snd charIteratorResult)

| ']' ->

let newPos = pos + 1 in

let res = charIterator newPos (List[]) in

(List(concatList myList (fst (res))), snd res)

| ' ' ->

let newPos = pos + 1 in

charIterator newPos myList

| ch ->

let rec digitCollector pos =

match stringView.[pos] with

| ' ' | ']' | '[' -> ([], pos)

| ch ->

let digitCollectorResult = digitCollector (pos + 1) in

(ch :: (fst (digitCollectorResult)), (snd (digitCollectorResult)))

in

let digitCollectorResult = digitCollector pos in

let stringNumber = String.of\_seq (List.to\_seq ((fst digitCollectorResult))) in

let value =

try IntAtom(int\_of\_string stringNumber)

with

| Failure \_ -> FloatAtom(float\_of\_string stringNumber)

| \_ -> IntAtom(int\_of\_string stringNumber) in

(List(concatList (List[value]) (fst (charIterator (snd digitCollectorResult) (List[])))), (snd digitCollectorResult))

(\*let newPos = pos + 1 in

List(concatList (List[IntAtom(int\_of\_char(ch) - int\_of\_char('0'))]) (charIterator newPos (List[])))\*)

else (myList, 0)

in

match fst (charIterator 0 (List[])) with

| List(h::t) -> h

| \_ -> List[];;

(\*

type token = Left\_br | Right\_br | Num of int | White\_sp

type complexlist = ComplexList of node list

and

node = Number of int | Node of complexlist list;;

let scan\_char c =

match c with

| '[' -> Left\_br

| ']' -> Right\_br

| '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' -> Num (Char.code c - Char.code '0')

| \_ -> White\_sp;;

let rec scan\_list txt pos =

if(pos<String.length txt)

then

let l=scan\_list txt (pos+1) in

let tk = scan\_char txt.[pos] in

match tk with

| White\_sp -> l

| \_ -> tk::l

else [];;

let parse\_list tl =

let rec aux (acc:complexlist) tl =

match acc with ComplexList ns ->

match tl with

| [] -> acc, []

| Right\_br::t -> acc,t

| Num n::t -> aux (ComplexList ( ns @ [Number n ] )) t

| Left\_br::t -> let (r,newt) = (aux (ComplexList[]) t) in

(

match ns, newt with

| [],[] -> r,[]

| \_,\_ -> aux(ComplexList(ns @ [Node [r]])) newt

)

| White\_sp::t -> aux acc t

in

let (r,\_) = aux (ComplexList[]) tl in

r;;\*)

(\* Goal \*)

printList myIntList;;

print\_string("\n");;

printList(myTask(myIntList));;

print\_string("\n");;

print\_string("\n");;

printList myFloatList;;

print\_string("\n");;

printList(myTask(myFloatList));;

(\* - - - Test part - - - \*)

print\_string("\n");;

print\_string("\n");;

printStringByChar stringView1;;

print\_string("\n");;

printList(listParser(stringView1));;

print\_string("\n");;

print\_string("\n");;

printStringByChar stringView4;;

print\_string("\n");;

printList(listParser(stringView4));;

let readFile fileName =

let ic = open\_in fileName in

let line = input\_line ic in

close\_in ic;

line;;

let fileInput =

print\_string("\n\n---File--\n");

printList(listParser(readFile "Input.txt"));

print\_string("\n");

printList(myTask(listParser(readFile "Input.txt")));

print\_string("\n---File--\n");;

let main =

if Array.length Sys.argv = 1 then

begin

print\_string "\nList = ";

let inputString = read\_line () in

printList(listParser(inputString));

print\_string("\n");

printList(myTask(listParser(inputString)));

end

else

let inputString = Sys.argv.(1) in

printList(listParser(inputString));

print\_string("\n");

printList(myTask(listParser(inputString)));;

main;;

(\*printList(parse\_list(scan\_list(stringView)));;\*)

(\* 7 вариант

type 'a variable =

| IntAtom of 'a

| List of 'a variable list

let myList =

List [

IntAtom 30;

IntAtom 1;

List [

IntAtom 13;

List [IntAtom 25]];

IntAtom 5];;

let myList1 =

List [

IntAtom 30.0;

IntAtom 1.0;

List [

IntAtom 13.0;

List [IntAtom 25.0]];

IntAtom 5.0];;

let rec incrementer (a, b) =

if a > 0 then

let a\_modified = a - 1 in

let b\_modified = b + 1 in

incrementer(a\_modified, b\_modified);

else b;;

let rec printList myList =

match myList with

| IntAtom a ->

print\_int a;

print\_string " "

| List l ->

print\_string "[ ";

let rec printIns li =

match li with

| [] -> ()

| x :: xs ->

printList x;

printIns xs in

printIns l;

print\_string "] ";;

let chooseType myList =

match myList with

| \_ when Obj.is\_int myList -> print\_string("int")

| \_ when Obj.is\_float myList -> print\_string("float");;

let rec sumList myList =

match myList with

| IntAtom x -> x

| List x ->

let rec sumListAux (accumulation, lst) =

match lst with

| [] -> accumulation

| hd :: tl -> sumListAux ((incrementer (sumList hd, accumulation)), tl)

in

sumListAux (0, x);;

printList myList;;

print\_string("\n");;

print\_int(sumList(myList));;

chooseType(myList);;

\*)