{-# OPTIONS\_GHC -Wall #-}

module Doroshevych01 where

-- 1 ------------------------------------------------------------------

power3 :: [Integer]

power3 = [i\*i\*i|i<- [1..]]

-- 2 -----------------------------------------

toPower3 :: [Integer]

toPower3 = [3^i|i<-[1..]]

-- 3 -----------------------------------------

sumPower3 :: Integer -> Integer

sumPower3 x | x==0=1

| otherwise = sum[3^i|i<-[1..x]]

-- 4 -----------------------------------------

sumPower :: Integer -> Integer -> Integer

sumPower m n |n<0=error "Error"

|m<0= error "Error"

|otherwise =sum[m^i|i<-[1..n]]

-- 5 -----------------------------------------

lessMe :: [Int] -> [Int]

lessMe [x]=[]

--lessMe xs = map (\i-> length) [xs] filter length

-- 6 -----------------------------------------

frequency :: [Int] -> [(Int,Int)]

frequency [x]=[(x,1)]

--frequency xs = (length . filter (== head xs)) xs

-- 7 -----------------------------------------

hailstone :: Int -> Int

hailstone n | mod n 2 == 0 =div n 2

|otherwise = 3\*n+1

-- 8 -----------------------------------------

hailSeq :: Int -> [Int]

hailSeq 1 = [1]

hailSeq n = n : hailSeq (hailstone n)

-- 9 ----------------------------------------

allHailSeq :: [[Int]]

allHailSeq = [hailSeq i|i<-[1..]]

-- 10 -----------------------------------------

--firstHailSeq :: Int -> Int

--firstHailSeq l = if length [hailSeq i|i<-[1..]]==l then head (hailSeq i) else 0

------------------------2

{-# OPTIONS\_GHC -Wall #-}

module Doroshevych02 where

-- ������ 1 -----------------------------------------

sumFl :: [Integer] -> Integer

sumFl [] = 0

sumFl xs = foldl (+) 0 xs

-- ������ 2 -----------------------------------------

productFr :: [Integer] -> Integer

productFr [] = 0

productFr (x:xs) = foldr (\*) x xs

-- ������ 3 -----------------------------------------

concatFr :: [Int] -> [Int] -> [Int]

concatFr [] [] = []

concatFr xs [] = xs

concatFr [] ys = ys

concatFr xs ys = foldr (:) ys xs

-- ������ 4 -----------------------------------------

sortInsert :: [Int] -> [Int]

sortInsert xs = foldl insert [] xs

insert :: [Int] -> Int -> [Int]

insert xy x = if null xy then (x : []) else

if (head xy) > x then (x : xy) else ((head xy) : insert (tail xy) x)

-- ������ 5 -----------------------------------------

findIndices ::(Int -> Bool) -> [Int] -> [Int]

findIndices p xs = [p1 | (p1,p2) <- zip [0..] xs, p p2]

-- ������ 6 -----------------------------------------

allReverse :: [String] -> [String]

allReverse = map (\ xss -> if length xss >= 2 then reverse xss else xss)

-- ������ 7 -----------------------------------------

noDigits :: String -> String

noDigits xs = filter (not.(`elem` "0123456789")) xs

-- ������ 8 ------------------------------------------

cntGood :: [Int -> Bool] -> Int -> Int

cntGood ps v = length (filter ($v) ps)

-- ������ 9 ------------------------------------------

trianglePas :: [[Integer]]

trianglePas = iterate (\prev -> 1 : zipWith (+) prev (tail prev) ++ [1]) [1]

-- ������ 10 -----------------------------------------

factorialsM :: [Integer]

factorialsM = 1 : zipWith (\*) factorialsM [2..]

-------------------3

{-# OPTIONS\_GHC -Wall #-}

module Doroshevych03 where

data BinTree a = EmptyB

| Node a (BinTree a) (BinTree a)

deriving (Show, Eq)

data Tree23 a = Leaf a

| Node2 (Tree23 a) a (Tree23 a)

| Node3 (Tree23 a) a (Tree23 a) a (Tree23 a)

| Empty23

deriving (Eq, Show)

-- 1 -----------------------------------------

getNode :: BinTree a -> a

getNode (Node a \_ \_) = a

isEmpty :: (Ord a) => BinTree a -> Bool

isEmpty EmptyB = True

isEmpty (Node \_ \_ \_) = False

isSearch :: (Ord a) => BinTree a -> Bool

isSearch EmptyB = True

--пусте

isSearch (Node \_ EmptyB EmptyB) = True

--в якого є тільки лівий вузол

isSearch (Node a left EmptyB) = a > (getNode left) && isSearch left

--в якого є тільки правий вузол

isSearch (Node a EmptyB right)= a < (getNode right) && isSearch right

--з двома вузлами

isSearch (Node a left right) = a > (getNode left) && a < (getNode right)

&& isSearch left && isSearch right

-- 2-----------------------------------------

elemSearch :: (Ord a) => BinTree a -> a -> Bool

--якщо немає значень в дереві то не можемо перевірити

elemSearch EmptyB \_ = False

--поки не знайшли перевіряємо по лівому а потім правому вузлам

elemSearch (Node a tl tr) e = if (a/=e) then

if (a>e) then elemSearch tl e else elemSearch tr e else True

-- 3 -----------------------------------------

insSearch :: (Ord a) => BinTree a -> a -> BinTree a

-- в пусте дерево проосто додаємо значення

insSearch EmptyB x = Node x EmptyB EmptyB

-- визначаємо в яке піддерево додати значення

insSearch (Node value left right) x | x < value = Node value (insSearch left x) right

|x > value = Node value left (insSearch right x)

|otherwise= Node value left right

--4 -----------------------------------------

--правий вузол

getR::BinTree a -> BinTree a

getR EmptyB = error "empty"

getR (Node \_ \_ x) = x

--лівий вузол

getL::BinTree a -> BinTree a

getL EmptyB = error "empty"

getL (Node \_ x \_) = x

delSearch :: (Ord a) => BinTree a -> a -> BinTree a

--якщо пусте то нічого не видаляємо і не змінюємо

delSearch tr x= if isEmpty(tr) then tr

else if (x<getNode(tr)) then (Node (getNode tr) (delSearch (getL tr) x) (getR tr))

else if (x>getNode(tr)) then (Node (getNode tr) (getL tr) (delSearch (getR tr) x))

--якщо тільки одне значення і ми його видалили то повертаємо пусте дерево

else if (x==getNode(tr)&&isEmpty(getL tr)&&isEmpty(getR tr)) then EmptyB

--повертаємо праве піддерево

else if (x==getNode(tr)&&isEmpty(getL tr)) then getR(tr)

--повертаємо ліве піддерево

else if (x==getNode(tr)&&isEmpty(getR tr)) then getL(tr)

else (Node (getNode(getL(tr))) (delSearch (getL(tr)) (getNode(getL tr))) (getR tr))

-- 5 -----------------------------------------

listToTree :: (Ord a) => [a] -> BinTree a

--записуємо список в пусте бінарне дерево

listToTree xs = foldl insSearch EmptyB xs

sortList :: (Ord a) => [a] -> [a]

--якщо пусте то виводимо пустий список

sortList [] = []

--інакше сортуємо за допомогою допоміжної функції список переведений в дерево

sortList xs = sortListB(listToTree xs)

sortListB :: BinTree a -> [a]

--пусте дерево дає пустий список

sortListB EmptyB = []

--додаємо значення а між ключами лівого і правого вузла

sortListB (Node a left right) = sortListB left ++ [a] ++ sortListB right

-- 6-----------------------------------------

isTree23 :: (Ord a) => Tree23 a -> Bool

isTree23 = undefined

--getNode23 :: BinTree a -> a

--getNode23 (Node2 a \_ \_ ) = a

--getNode23 (Node3 a a \_ \_ \_) = a

--isEmpty23 :: (Ord a) => BinTree a -> Bool

--isEmpty23 Empty23 = True

--isEmpty23 (Node2 \_ \_ \_ ) = False

--isEmpty23 (Node3 \_ \_ \_ \_ \_) = False

--isTree23 :: (Ord a) => Tree23 a -> Bool

--isTree23 Empty23 = True

--isTree23 (Node2 \_ Empty23 Empty23 ) = True

--isTree23 (Node2 a left Empty23 ) = a > (getNode23 left) && isTree23 left

--isTree23 (Node2 a Empty23 right)= a < (getNode23 right) && isTree23 right

--isTree23 (Node2 a left right) = a > (getNode23 left) && a < (getNode23 right)

-- && isTree23 left && isTree23 right

--isTree23 (Node3 \_ Empty23 \_ Empty23 ) = True

--isTree23 (Node3 a left a Empty23 ) = a > (getNode23 left) && isTree23 left

--isTree23 (Node3 a Empty23 a right)= a < (getNode23 right) && isTree23 right

--isTree23 (Node3 a left a right) = a > (getNode23 left) && a < (getNode23 right)

-- && isTree23 left && isTree23 right

--7-----------------------------------------

elemTree23 :: (Ord a) => Tree23 a -> a -> Bool

--якщо пусте то не містить

elemTree23 Empty23 \_ = False

--якщо дерево складається з листка

elemTree23 (Leaf a) e = e == a

--для двох вузлів

elemTree23 (Node2 tl x tr) e = if (e==x) then (elemTree23 tl e) || (elemTree23 tr e)

-- в ліве чи праве піддерево

else if (e < x) then elemTree23 tl e else elemTree23 tr e

--для трьох вузлів

elemTree23 (Node3 tl x tm y tr) e = if (e < x) then elemTree23 tl e

--аналізуємо далі

else if (e==x) then (elemTree23 tl e) || (elemTree23 tm e)

else if (y==e) then (elemTree23 tm e) || (elemTree23 tr e)

else if (y < e) then elemTree23 tr e else elemTree23 tm e

-- 8-----------------------------------------

eqTree23 :: (Ord a) => Tree23 a -> Tree23 a -> Bool

eqTree23 = undefined

-- 9-----------------------------------------

insTree23 :: (Ord a) => Tree23 a -> a -> Tree23 a

insTree23 = undefined

--insTree23 Empty23 x = Node2 x Empty23 Empty23

--insTree23 (Node2 value left right) x =

-- if x < value

-- then Node2 value (insTree23 left x) right

-- else if x > value

-- then Node2 value left (insTree23 right x)

-- else Node2 value left right

-- isTerminal tr = True <=>

isTerminal :: (Ord a) => Tree23 a -> Bool

isTerminal (Node2 (Leaf \_) \_ \_) = True

isTerminal (Node3 (Leaf \_) \_ \_ \_ \_) = True

isTerminal \_ = False

------------------------------------------------------

---------------------------------------------------

insert :: (Ord a) => a -> Tree23 a -> (Tree23 a, Maybe (a, Tree23 a))

insert v tr | isTerminal tr = insTerm v tr

| otherwise = insNode v tr

-- insTerm v tr -----------------------------------------------------

insTerm :: (Ord a) => a -> Tree23 a -> (Tree23 a, Maybe (a, Tree23 a))

insTerm = undefined

-- insNode v tr --------------------------------------------------

insNode :: (Ord a) => a -> Tree23 a -> (Tree23 a, Maybe (a, Tree23 a))

insNode = undefined

---

bt1, bt2 :: BinTree Int

bt1 = Node 9 (Node 4 EmptyB

(Node 8 EmptyB EmptyB))

(Node 20 (Node 10 EmptyB EmptyB)

EmptyB)

bt2 = Node 9 (Node 4 EmptyB

(Node 8 (Node 6 EmptyB EmptyB)

EmptyB))

(Node 20 (Node 10 EmptyB EmptyB)

EmptyB)

---- 2-3-

tr1, tr2, tr3, tr4,tr5 :: Tree23 Int

tr1 = Node2 (Node2 (Node2 (Leaf 0) 1 (Leaf 1))

2

(Node2 (Leaf 2) 3 (Leaf 3)))

4

(Node2 (Node2 (Leaf 4) 5 (Leaf 5))

6

(Node2 (Leaf 6) 7 (Leaf 7)))

tr2 = Node3 (Node2 (Leaf 0) 1 (Leaf 1))

2

(Node3 (Leaf 2) 3 (Leaf 3) 4 (Leaf 4))

5

(Node3 (Leaf 5) 6 (Leaf 6) 7 (Leaf 7))

tr3 = Node3 (Node2 (Leaf 2) 5 (Leaf 5))

7

(Node3 (Leaf 7) 8 (Leaf 8) 12 (Leaf 12))

16

(Node2 (Leaf 16) 19 (Leaf 19))

tr4 = Node3 (Node2 (Leaf 2) 5 (Leaf 5))

7

(Node3 (Leaf 7) 8 (Leaf 8) 12 (Leaf 12))

16

(Node3 (Leaf 16) 18 (Leaf 18) 19 (Leaf 19))

tr5 = Node2 (Node2 (Node2 (Leaf 2) 5 (Leaf 5))

7

(Node2 (Leaf 7) 8 (Leaf 8))

)

10

(Node2 (Node2 (Leaf 10) 12 (Leaf 12))

16

(Node3 (Leaf 16) 18 (Leaf 18) 19 (Leaf 19))

)

--------------------------4

{-# OPTIONS\_GHC -Wall #-}

module Doroshevych04 where

import Data.Char

type Name = String

type Attributes = [(Name, String)]

data XML = Text String | Element Name Attributes [XML]

deriving (Eq, Show)

type Stack = [XML]

-- Задача 1 -----------------------------------------

--видаляємо допоки не знайдемо інший елемент

skipSpace :: String -> String

skipSpace = dropWhile(flip elem "\n\" ")

-- Задача 2 -----------------------------------------

--не зроблена

getAttribute :: String -> XML -> String

getAttribute \_ \_ = ""

-- Задача 3 -----------------------------------------

getChildren :: String -> XML -> [XML]

--якщо пустий

getChildren \_ \_= []

--шукаємо тільки за ім'ям

getChildren s (Element \_ \_ st) = concatMap find st

where

find \_ = []

--якщо знайшли то додаємо елемент на початок списку і рекурсія

find e@(Element n \_ \_)| n == s = e : getChildren s e

| otherwise = getChildren s e

-- Задача 4 -----------------------------------------

getChild :: String -> XML -> XML

getChild= (head .).getChildren

-- Задача 5 -----------------------------------------

addChild :: XML -> XML -> XML

-- Передумова: другий аргумент - завжди побудований конструктором Element

--додаємо с в список

addChild s (Element n as st) = Element n as (st ++ [s])

-- Задача 6 -----------------------------------------

--не зроблено

getValue :: XML -> XML

getValue = undefined

-- Задача 7 -----------------------------------------

addText :: String -> Stack -> Stack

-- Передумова: Є по крайній мірі один елемент Element в стеку

addText t (s : st) = (addChild (Text t) s) : st

-- Задача 8 -----------------------------------------

popAndAdd :: Stack -> Stack

-- Передумова: Є по крайній мірі два елемента Elements в стеку

popAndAdd (s : e : st) = addChild s e : st

-- Початковий елемент стеку

sentinel :: XML

sentinel = Element "" [] []

-- Задача 9 -----------------------------------------

parseAttributes :: String -> (Attributes, String)

-- Передумова: Рядок, що містить XML-атрибути, синтаксично вірний

parseAttributes s = parseAtt s []

where

parseAtt k ac

--якщо > то кінець, інакше прибираємо пробіли і

| x == '>' = (ac, xs)

| otherwise = parseAtt (skipSpace rest) ((name, this):ac)

where

a@(x : xs) = skipSpace k

(name, atr)= parseName a

(l : ls)= skipSpace atr

(this, rest)= span (flip notElem "\" ") (skipSpace ls)

-- Аналіз імені елемента/атрибута

parseName :: String -> (Name, String)

parseName [] = error "Error: attempt to read empty name"

parseName s@(c1 : \_)

| isAlpha c1 = break (not . isNameChar) s

| otherwise = error ("parseName error: name " ++ show s ++" must begin with a letter")

where

isNameChar c = isAlpha c || isDigit c || elem c "-."

-- Задача 10 -----------------------------------------

parse :: String -> XML

-- Передумова: рядок, що містить XML-документ, синтаксично вірний

parse s = parse' (skipSpace s) [sentinel]

parse' :: String -> Stack -> XML

parse'= undefined

-----------------------------------------------------------------------

-- Деякі корисні функції перетворення в рядок і виводу

-- Функція перетворення в рядок ('show' function) для XML об'єктів

showXML :: XML -> String

showXML (Text t) = t

showXML (Element n as es)

= "<" ++ n ++ showAtts as ++ ">" ++ concatMap showXML es ++ "</" ++ n ++ ">"

where

showAtts ast = concatMap showAtt ast

showAtt (n1, v) = " " ++ n1 ++ "=" ++ "\"" ++ v ++ "\""

-- Функція перетворення в рядок ('show' function) для списку XML об'єктів

showXMLs :: [XML] -> String

showXMLs = concatMap showXML

-- Функція виводу XML об'єкта на екран

printXML :: XML -> IO()

printXML = putStrLn . showXML

-------------------------------------------------------------------------

-- Тестові дані

-- Прості тести XML-об'єктів (без проміжків)

s1, s2, s3 :: String

s1 = "<a>A</a>"

s2 = "<a x=\"1\"><b>A</b><b>B</b></a>"

s3 = "<a>\

\<b>\

\<c att=\"att1\">text1</c>\

\<c att=\"att2\">text2</c>\

\</b>\

\<b>\

\<c att=\"att3\">text3</c>\

\<d>text4</d>\

\</b>\

\</a>"

-- Результати аналізу попередніх XML-об'єктів

x1, x2, x3 :: XML

x1 = Element "a" [] [Text "A"]

x2 = Element "a"

[("x","1")]

[Element "b" [] [Text "A"],

Element "b" [] [Text "B"]]

x3 = Element "a"

[]

[Element "b"

[]

[Element "c"

[("att","att1")]

[Text "text1"],

Element "c"

[("att","att2")]

[Text "text2"]],

Element "b"

[]

[Element "c"

[("att","att3")]

[Text "text3"],

Element "d"

[]

[Text "text4"]]]

casablanca :: String

casablanca

= "<film title=\"Casablanca\">\n <director>Michael Curtiz</director>\n <year>1942\

\</year>\n</film>\n\n\n"

casablancaParsed :: XML

casablancaParsed

= Element "film"

[("title","Casablanca")]

[Text "\n ",

Element "director" [] [Text "Michael Curtiz"],

Text "\n ",

Element "year" [] [Text "1942"],

Text "\n"]

-- XML-документ з Мал.1

films :: String

films

= "<filmlist>\n\

\ <film title = \"Rear Window\">\n\

\ <director>Alfred Hitchcock</director>\n\

\ <composer>Franz Waxman</composer>\n\

\ <year>1954</year>\n\

\ </film>\n\

\ <film title = \"2001: A Space Odyssey\">\n\

\ <director>Stanley Kubrick</director>\n\

\ <composer>Richard Strauss</composer>\n\

\ <composer>Gyorgy Ligeti</composer>\n\

\ <composer>Johann Strauss</composer>\n\

\ <year>1968</year>\n\

\ </film>\n\

\ <film title=\"Lawrence of Arabia\" >\n\

\ <duration>228</duration>\n\

\ <director>David Lean</director>\n\

\ <composer>Maurice Jarre</composer>\n\

\ </film>\n\

\</filmlist>\n\n\n"

-- Результат аналізу попереднього докуменнту ('parse films')

filmsParsed :: XML

filmsParsed

= Element "filmlist"

[]

[Text "\n ",

Element "film" [("title","Rear Window")]

[Text "\n ",

Element "director" [] [Text "Alfred Hitchcock"],

Text "\n ",

Element "composer" [] [Text "Franz Waxman"],

Text "\n ",

Element "year" [] [Text "1954"],

Text "\n "],

Text "\n ",

Element "film" [("title","2001: A Space Odyssey")]

[Text "\n ",

Element "director" [] [Text "Stanley Kubrick"],

Text "\n ",

Element "composer" [] [Text "Richard Strauss"],

Text "\n ",

Element "composer" [] [Text "Gyorgy Ligeti"],

Text "\n ",

Element "composer" [] [Text "Johann Strauss"],

Text "\n ",

Element "year" [] [Text "1968"],

Text "\n "],

Text "\n ",

Element "film" [("title","Lawrence of Arabia")]

[Text "\n ",

Element "duration" [] [Text "228"],

Text "\n ",

Element "director" [] [Text "David Lean"],

Text "\n ",

Element "composer" [] [Text "Maurice Jarre"],

Text "\n "],

Text "\n"]

-------------------5

{-# OPTIONS\_GHC -Wall #-}

module Doroshevych05 where

data AbstractInteger = Zero

| Succ AbstractInteger

| Pred AbstractInteger

deriving (Show, Eq)

-- Задача 1 -----------------------------------------

instance Ord AbstractInteger where

(<=) Zero Zero = True

(<=) Zero (Pred \_) = False

(<=) Zero (Succ \_) = True

(<=) (Succ \_) Zero = False

(<=) (Pred \_) Zero = True

(<=) (Pred \_) (Succ \_) = True

(<=) (Succ \_) (Pred \_) = False

(<=) (Succ a) (Succ b) = a<= b

(<=) (Pred a) (Pred b) = a <= b

-- Задача 2 ----------------------------------------

aiToInteger :: AbstractInteger -> Integer

aiToInteger Zero = 0

aiToInteger (Succ ai) = 1 + (aiToInteger ai)

aiToInteger (Pred ai) = (aiToInteger ai) - 1

-- Задача 3 -----------------------------------------

plusAbs :: AbstractInteger -> AbstractInteger -> AbstractInteger

plusAbs Zero Zero = Zero

plusAbs Zero (Succ ai) = Succ ai

plusAbs Zero (Pred ai) = Pred ai

plusAbs (Pred ai) Zero = Pred ai

plusAbs (Succ ai) Zero = Succ ai

plusAbs (Pred ai1) (Succ ai2) = plusAbs ai1 ai2

plusAbs (Pred ai1) (Pred ai2) = Pred (Pred (plusAbs ai1 ai2))

plusAbs (Succ ai1) (Pred ai2) = plusAbs ai1 ai2

plusAbs (Succ ai1) (Succ ai2) = Succ (Succ (plusAbs ai1 ai2))

-- Задача 4 -----------------------------------------

timesAbs :: AbstractInteger -> AbstractInteger -> AbstractInteger

timesAbs Zero \_ = Zero

timesAbs \_ Zero = Zero

timesAbs ai (Succ Zero) = ai

timesAbs (Succ Zero) ai = ai

--pred\*pred=succ suc\*pred= pred suc\*suc=suc

timesAbs (Pred Zero) (Pred Zero) = Succ Zero

timesAbs (Succ Zero) (Pred Zero) = Pred Zero

timesAbs (Pred Zero) (Succ Zero) = Pred Zero

timesAbs (Succ Zero) (Succ Zero) = Succ Zero

timesAbs (Succ ai) (Pred Zero) = Pred (timesAbs ai (Pred Zero))

timesAbs (Pred ai) (Pred Zero) = Succ (timesAbs ai (Pred Zero))

timesAbs (Pred Zero) (Succ ai) = Pred (timesAbs ai (Pred Zero))

timesAbs (Pred Zero) (Pred ai) = Succ (timesAbs ai (Pred Zero))

timesAbs (Pred ai1) (Pred ai2) = timesAbs (timesAbs (Pred ai1) (Pred Zero )) (timesAbs (Pred ai2) (Pred Zero ))

timesAbs (Pred ai1) (Succ ai2) = (Pred ai1) + (timesAbs (Pred ai1) ai2)

timesAbs (Succ ai1) (Pred ai2) = (Pred ai2) + (timesAbs (Pred ai2) ai1)

timesAbs (Succ ai1) (Succ ai2) = (Succ ai1) + (timesAbs (Succ ai1) ai2)

-- Задача 5 -----------------------------------------

instance Num AbstractInteger where

(+) = plusAbs

(\*) = timesAbs

negate = aiNegate

fromInteger = aiFromInteger

abs Zero = Zero

abs x@(Pred \_) = negate x

abs x@(Succ \_) = x

signum Zero = Zero

signum (Succ \_) = Succ Zero

signum (Pred \_) = Pred Zero

aiNegate ::AbstractInteger -> AbstractInteger

aiNegate Zero = Zero

aiNegate (Pred x) = Succ (aiNegate x)

aiNegate (Succ x) = Pred (aiNegate x)

aiFromInteger ::Integer ->AbstractInteger

aiFromInteger 0 = Zero

aiFromInteger x |x>0 = Succ (aiFromInteger (x - 1))

|x<0 = Pred (aiFromInteger (x + 1))

-- Задача 6 -----------------------------------------

factorial :: (Eq a, Num a) => a -> a

factorial 0 = 1

factorial 1 = 1

factorial n = n \* factorial (n-1)

-- Задача 7 -----------------------------------------

data Quaternion = Quaternion Double Double Double Double deriving (Eq)

instance Show Quaternion where

show (Quaternion h i j k) = show h ++(if i >= 0 then "+" else "") ++ show i ++ "i" ++(if j >= 0 then "+" else "") ++ show j ++ "j" ++(if k >= 0 then "+" else "") ++ show k ++ "k"

-- Задача 8 -----------------------------------------

plusQuaternion :: Quaternion -> Quaternion -> Quaternion

plusQuaternion (Quaternion h i j k) (Quaternion h1 i1 j1 k1) = Quaternion (h + h1) (i + i1) (j + j1) (k + k1)

-- Задача 9 -----------------------------------------

timesQuaternion :: Quaternion -> Quaternion -> Quaternion

timesQuaternion (Quaternion h1 i1 j1 k1) (Quaternion h2 i2 j2 k2) =

(Quaternion (h1 \* h2 - i1 \* i2 - j1 \* j2 - k1 \* k2)(h1 \* i2 + i1 \* h2 + j1 \* k2 - k1 \* j2)(h1 \* j2 - i1 \* k2 + j1 \* h2 + k1 \* i2)(h1 \* k2 + i1 \* j2 - j1 \* i2 + k1 \* h2))

--- Задача 10 ----------------------------------------

instance Num Quaternion where

(+) = plusQuaternion

(\*) = timesQuaternion

negate (Quaternion h i j k) = Quaternion (-h) (-i) (-j) (-k)

fromInteger i = Quaternion (fromInteger i) 0 0 0

abs (Quaternion h i j k) = Quaternion (sqrt(h\*h+i\*i+j\*j+k\*k)) 0 0 0

signum (Quaternion h i j k) = let (Quaternion l0 \_ \_ \_) = abs (Quaternion h i j k) in (Quaternion (h/l0) (i/l0) (j/l0) (k/l0))

-----------6

{-# OPTIONS\_GHC -Wall #-}

module Doroshevych06 where

import Data.Maybe

import qualified Data.Map as M

-- Всі програми і їх елементи являються вірними (well-formed) в наступному значенні:

-- Всі оператори, функції і процедури застосовуються

-- до вірної кількості аргументів, кожний з яких має відповідний тип.

-- Вирази, які повинні обчислювати логічні значення, будуть завжди обчислюватися

-- до 0 (false) або 1 (true).

-- В присвоєнні масиву a[i] = e масив завжди визначений (в області дії).

-- Процедура, що викликається як x := p(e1, …, en), завжди буде повертати значення

-- (закінчує своє обчислення оператором return e)

-- Оператор return завжди останній оператор для виконання в блоку процедури

-- (тобто немає "мертвого коду")

--------------------------------------------------------------------

type Id = String

data Value = I Int | A [(Int, Int)]

deriving (Eq, Show)

data Op = Add | Minus | Mul | Less | Equal | Index

deriving (Eq, Show)

data Exp = Const Value |

Var Id |

OpApp Op Exp Exp |

Cond Exp Exp Exp |

FunApp Id [Exp]

deriving (Eq, Show)

data VarDef = Arr Id | Int Id deriving (Eq, Show)

type FunDef = (Id, ([VarDef], Exp))

type Binding = M.Map Id Value

type StateP = ([Binding],Binding)

-- st = ([locn,.. loc1], glob) стек локальних записів активацій + глобальний запис активації

data Statement = Assign Id Exp |

AssignA Id Exp Exp |

If Exp Block Block |

While Exp Block |

Call Id Id [Exp] |

Return Exp

deriving (Eq, Show)

type Block = [Statement]

type ProcDef = (Id, ([VarDef], Block))

type Program = ([VarDef], [FunDef], [ProcDef])

-- Задача 1 -----------------------------------------

getValue :: Id -> StateP -> Value

--getValue= undefined

-- Передумова: Значення змінної Id є в стані StateP

getValue id st = fromJust (lookup id st)

-- Задача 2 -----------------------------------------

getLocals :: StateP -> Binding

getLocals = undefined

getGlobals :: StateP -> Binding

getGlobals = undefined

-- Задача 3 -----------------------------------------

assignArray :: Value -> Value -> Value -> Value

-- Аргументи - масив, індекс і (нове) значення відповідно

-- Передумова: Три аргумента (Value) мають значення відповідного типу

-- (масив (A), ціле (I) і ціле (I)) відповідно.

--assignArray (A a) (I i) (I v) = A ((i, v) : (filter (\(x, y) -> x /= i) a))

assignArray (A arr) (I i) (I v)

= A ((i,v) : filteredArr)

where

filteredArr = filter (\(x,y) -> i /= x) arr

-- Задача 4 -----------------------------------------

updateVar :: (Id, Value) -> StateP -> StateP

updateVar = undefined

-- Задача 5 -----------------------------------------

applyOp :: Op -> Value -> Value -> Value

-- Передумова: Значення мають відповідні типи (I або A) для кожної операції

applyOp Add (I f) (I s) = I $ f + s

applyOp Minus (I f) (I s) = I $ f - s

applyOp Mul (I f) (I s) = I $ f \* s

applyOp Less (I f) (I s) = I $ fromEnum $ f < s

applyOp Equal (I f) (I s) = I $ fromEnum $ f == s

applyOp Index (A xs) (I n) = I $ fromMaybe 0 (lookup n xs)

applyOp op f s = error $ "Error "

-- Задача 6 -----------------------------------------

bindArgs :: [Id] -> [Value] -> Binding

-- Передумова: списки мають однакову довжину

bindArgs = undefined

-- Задача 7 -----------------------------------------

eval :: Exp -> [FunDef] -> StateP -> Value

eval = undefined

evalArgs :: [Exp] -> [FunDef] -> StateP -> [Value]

evalArgs es def state

= [eval e def state | e <- es]

-- Задача 8 -----------------------------------------

executeStatement :: Statement -> [FunDef] -> [ProcDef] -> StateP -> StateP

executeStatement = undefined

executeBlock :: Block -> [FunDef] -> [ProcDef] -> StateP -> StateP

executeBlock = undefined

---------------------------------------------------------------------

-- Допоміжні функції і дані для тестування...

-- Функція пошуку...

lookUp :: (Eq a, Show a) => a -> [(a, b)] -> b

lookUp x t = fromMaybe (error ("\nНе знайдено " ++ show x ))

(lookup x t)

-- Стан для тестування

sampleState :: StateP

sampleState = ([M.fromList [("x",I 5)]], M.fromList [("y",I 2),("a", listToVal [4,2,7])])

-- Перетворює список цілих в масив Value...

listToVal :: [Int] -> Value

listToVal xs = A (zip [0..] xs)

-- Перетворює ціле в Exp...

intToExp :: Int -> Exp

intToExp n = Const (I n)

-- Реалізація виконання програми

program :: Program -> StateP

program (dvx, dfx, dpx) =

let initv :: VarDef -> (Id, Value)

initv (Arr v) = (v, A [])

initv (Int v) = (v, I 0)

gl = M.fromList (map initv dvx)

in executeStatement (Call "" "main" []) dfx dpx ([],gl)

-- fib чиста функція

-- Функція fib, що обчислює число Фібоначчі

-- func fib(n) =

-- (n < 3 ? 1 : fib(n-1) + fib(n-2))

fib :: FunDef

fib = ("fib",

([Int "n"], Cond (OpApp Less (Var "n") (Const (I 3)))

(Const (I 1))

(OpApp Add (FunApp "fib" [OpApp Minus (Var "n") (Const (I 1))])

(FunApp "fib" [OpApp Minus (Var "n") (Const (I 2))]))

)

)

-- Масив

sampleArray :: Exp

sampleArray = Const (listToVal [9,5,7,1])

-- Сума елементів масиву 0..n ...

sumA1 :: ProcDef

sumA1 = ("sumA1",

([Arr "a", Int "n"],

[Assign "s" (Const (I 0)),

Assign "i" (Const (I 0)),

Assign "limit" (OpApp Add (Var "n") (Const (I 1))),

While (OpApp Less (Var "i") (Var "limit"))

[Assign "s" (OpApp Add (Var "s")

(OpApp Index (Var "a") (Var "i"))),

Assign "i" (OpApp Add (Var "i") (Const (I 1)))

],

Return (Var "s")]

)

)

-- Додавання двох чисел...

gAdd :: ProcDef

gAdd = ("gAdd",

([Int "x", Int "y"], [Assign "gSum" (OpApp Add (Var "x") (Var "y"))])

)

-- Повна програма

pr1 :: Program

pr1 = ([Int "gSum"], [], [gAdd, ("main",([],[Call "" "gAdd" [intToExp 5, intToExp 10] ]))])

-----------7

{-# OPTIONS\_GHC -Wall #-}

module Doroshevych07 where

type Index = Int

data BExp = Prim Bool | IdRef Index | Not BExp | And BExp BExp | Or BExp BExp

deriving (Eq, Ord, Show)

type Env = [(Index, Bool)]

type NodeId = Int

type BDDNode = (NodeId, (Index, NodeId, NodeId))

type BDD = (NodeId, [BDDNode])

-- 1 -----------------------------------------

lookUp :: Eq a => a -> [(a, b)] -> b

lookUp x xs = snd $ head $ filter (\(a, b) -> a == x) xs

checkSat :: BDD -> Env -> Bool

checkSat (root, nodes) env = checkSat' root

where

checkSat' :: NodeId -> Bool

checkSat' 0 = False

checkSat' 1 = True

checkSat' id | lookUp i env = checkSat' r

| otherwise = checkSat' l

where

(i, l, r) = lookUp id nodes

-- 2 -----------------------------------------

sat :: BDD -> [[(Index, Bool)]]

sat (0, \_) = []

sat bdd = sat' nodeID

where

(nodeID, nodes) = bdd

sat' 0 = []

sat' 1 = [[]]

sat' n = satL ++ satR

where

satL = map ((i, False):) (sat' l)

satR = map ((i, True):) (sat' r)

(i, l, r) = lookUp n nodes

-- 3 -----------------------------------------

simplify :: BExp -> BExp

simplify e = e

simplify (And (Prim b1) (Prim b2)) = Prim (b1 && b2)

simplify (Not (Prim b)) = Prim (not b)

simplify (Or (Prim b1) (Prim b2)) = Prim (b1 || b2)

-- 4 -----------------------------------------

restrict :: BExp -> Index -> Bool -> BExp

restrict (Prim b) \_ \_= Prim b

restrict (IdRef i) i0 v | (i == i0) = Prim v

| otherwise = IdRef i

restrict (And b1 b2) i v = simplify $ And (restrict b1 i v) (restrict b2 i v)

restrict (Not b) i v = simplify $ Not (restrict b i v)

restrict (Or b1 b2) i v = simplify $ Or (restrict b1 i v) (restrict b2 i v)

-- 5 -----------------------------------------

buildBDD :: BExp -> [Index] -> BDD

buildBDD e xs = buildBDD' e 2 xs

buildBDD' :: BExp -> NodeId -> [Index] -> BDD

buildBDD' (Prim b) \_ \_ | b = (1, [])

| otherwise = (0, [])

buildBDD' e n xs = (n, addFun e n xs)

addFun :: BExp -> NodeId -> [Index] -> [BDDNode]

addFun \_ \_ [] = []

addFun e n (i:xs) = let

rExp = restrict e i True

lExp = restrict e i False

rNodeId = 1 + n \* 2

lNodeId = n \* 2

in (n, (i, lNodeId, rNodeId)) :(addFun lExp lNodeId xs) ++(addFun rExp rNodeId xs)

addFun e n (i:[]) = let

rExp = restrict e i True

lExp = restrict e i False

rNodeId = if (Prim True) /= rExp then 0 else 1

lNodeId = if (Prim True) /= lExp then 0 else 1

in [(n, (i, lNodeId, rNodeId))]

-- 6 -----------------------------------------

buildROBDD :: BExp -> [Index] -> BDD

buildROBDD = undefined

------------------------------------------------------

b1, b2, b3, b4, b5, b6, b7, b8, b9 :: BExp

b1 = Prim False

b2 = Not (And (IdRef 1) (Or (Prim False) (IdRef 2)))

b3 = And (IdRef 1) (Prim True)

b4 = And (IdRef 7) (Or (IdRef 2) (Not (IdRef 3)))

b5 = Not (And (IdRef 7) (Or (IdRef 2) (Not (IdRef 3))))

b6 = Or (And (IdRef 1) (IdRef 2)) (And (IdRef 3) (IdRef 4))

b7 = Or (Not (IdRef 3)) (Or (IdRef 2) (Not (IdRef 9)))

b8 = Or (IdRef 1) (Not (IdRef 1))

b9 = And (IdRef 3) (Or (IdRef 2) (And (Not (IdRef 2)) (IdRef 1)))

bdd1, bdd2, bdd3, bdd4, bdd5, bdd6, bdd7, bdd8, bdd9 :: BDD

bdd1 = (0,[])

bdd2 = (2,[(4,(2,1,1)),(5,(2,1,0)),(2,(1,4,5))])

bdd3 = (5,[(5,(1,0,1))])

bdd4 = (2,[(2,(2,4,5)),(4,(3,8,9)),(8,(7,0,1)),(9,(7,0,0)),

(5,(3,10,11)),(10,(7,0,1)),(11,(7,0,1))])

bdd5 = (3,[(4,(3,8,9)),(3,(2,4,5)),(8,(7,1,0)),(9,(7,1,1)),

(5,(3,10,11)),(10,(7,1,0)),(11,(7,1,0))])

bdd6 = (2,[(2,(1,4,5)),(4,(2,8,9)),(8,(3,16,17)),(16,(4,0,0)),

(17,(4,0,1)),(9,(3,18,19)),(18,(4,0,0)),(19,(4,0,1)),

(5,(2,10,11)),(10,(3,20,21)),(20,(4,0,0)),(21,(4,0,1)),

(11,(3,22,23)),(22,(4,1,1)),(23,(4,1,1))])

bdd7 = (6,[(6,(2,4,5)),(4,(3,8,9)),(8,(9,1,1)),(9,(9,1,0)),

(5,(3,10,11)),(10,(9,1,1)),(11,(9,1,1))])

bdd8 = (2,[(2,(1,1,1))])

bdd9 = (2,[(2,(1,4,5)),(4,(2,8,9)),(8,(3,0,0)),(9,(3,0,1)),(5,(2,10,11)),(10,(3,0,1)),(11,(3,0,1))])

---------------------8

{-# OPTIONS\_GHC -Wall #-}

module Doroshevych08 where

import Data.Array

type Graph = Array Int [Int]

--Testing samples

gr1, gr2, gr3, gr4, gr5 :: Graph

gr1 = array (1,9) [(1,[2]), (2,[3,5]), (3, [7]), (4, [7]), (5, [8]), (6, [9]), (7, [8]), (8, [3]), (9, [])]

gr2 = array (1,9) [(1,[2]), (2,[3,5]), (3, [7]), (4, [7]), (5, [8]), (6, [9]), (7, []), (8, [3]), (9, [])]

gr3 = array (1,4) [(1, [2,3,4]), (2, [3,4]), (3, [4]), (4, [])]

gr4 = array (1,4) [(1, [2,3,4]), (2, [3]), (3, [4]), (4, [])]

gr5 = array (1,9) [(1,[2]), (2,[1,3,5]), (3, [2,7,8]), (4, [7]), (5, [2,8]), (6, [9]), (7, [3,4,8]), (8, [3,5,7]), (9, [6])]

-- 1 ------------------------------------------

longWay :: Graph -> Int -> Int -> Maybe [Int]

longWay gr st end =

let arr = dfs1 gr st end []

in if (null arr) then Nothing else Just (snd $ maximum $ map (\x -> (length x, x)) (arr))

-- ������ 2 -----------------------------------------

isNoCycle :: Graph -> Bool

isNoCycle = undefined

-- ������ 3 -----------------------------------------

isTransitive :: Graph -> Bool

isTransitive = undefined

-- ������ 4 -----------------------------------------

isGraph :: Graph -> Bool

isGraph = undefined

-- ������ 5 -----------------------------------------

shortWay :: Graph -> Int -> Int -> Maybe [Int]

shortWay = undefined

-- ������ 6 -----------------------------------------

isConnecting :: Graph -> Bool

isConnecting = undefined

-- ������ 7 -----------------------------------------

components :: Graph -> [[Int]]

components = undefined

-- ������ 8 -----------------------------------------

topolSorting :: Graph -> Maybe[Int]

topolSorting = undefined

dfs1 :: Graph -> Int -> Int -> [Int] -> Array Int [Int]

dfs1 gr st end t

| (st == end) = Array st [st]

| (st `elem` t) = []

| otherwise = foldl (\a b -> map (\c -> st : c) (dfs1 gr b end (st:t)) ++ a) [] (gr !! (st-1))

dfs2 :: Graph -> Int -> [Int] -> Bool

dfs2 gr st t = (not (st `elem` t)) && (foldl (\a b -> (dfs2 gr b (st:t)) && a) True (gr !! (st-1)))

dfs3 :: Graph -> Int -> [Int] -> [Int]

dfs3 gr st t

| (st `elem` t) = []

| otherwise = st: foldl (\a b -> dfs3 gr b (st:t) ++ a) [] (gr !! (st-1))

------------------9

{-# OPTIONS\_GHC -Wall #-}

module Doroshevych09 where

import Data.List

data RE = Null |

Term Char |

Seq RE RE |

Alt RE RE |

Rep RE |

Plus RE |

Opt RE

deriving (Eq, Show)

type State = Int

data Label = C Char | Eps deriving (Eq, Ord, Show)

type Transition = (State, State, Label)

type Automation = (State, [State], [Transition])

type MetaState = [State]

type MetaTransition = (MetaState, MetaState, Label)

-- Задача 1 -----------------------------------------

simplify :: RE -> RE

simplify Null = Null

simplify (Term с) = Term с

simplify (Seq r1 r2) = Seq (simplify r1) (simplify r2)

simplify (Alt r1 r2) = Alt (simplify r1) (simplify r2)

simplify (Plus re) = Seq (simplify re) (Rep ( simplify re))

simplify (Rep re) = Rep ( simplify re)

simplify (Opt re) = Alt (simplify re) Null

simplify re = re

-- Задача 2 -----------------------------------------

startState :: Automation -> State

terminalStates :: Automation -> [State]

transitions :: Automation -> [Transition]

startState (beg, \_, \_) = beg

terminalStates (\_, fin, \_) = fin

transitions (\_, \_, nxt) = nxt

-- Задача 3 -----------------------------------------

isTerminal :: State -> Automation -> Bool

isTerminal s aut = elem s (terminalStates aut)

-- Задача 4 -----------------------------------------

transitionsFrom :: State -> Automation -> [Transition]

transitionsFrom s aut = filter(\(st, \_, \_) -> st == s) (transitions aut)

-- Задача 5 -----------------------------------------

labels :: [Transition] -> [Label]

labels trx = nub (filter (/= Eps) (map (\(\_, \_, t) -> t) trx))

-- Задача 6 -----------------------------------------

stStep :: Automation -> State -> Label -> [State]

setStep :: Automation -> [State] -> Label -> [State]

closure :: Automation -> [State] -> [State]

stStep = undefined

setStep = undefined

closure = undefined

-- Задача 7 -----------------------------------------

accepts :: Automation -> String -> Bool

accepts aut st = accepts' aut st

accepts' :: Automation -> String -> Bool

accepts' (beg,fin,\_) [] = elem beg fin

accepts' (beg,fin,nxt) s = foldl (||) False (map (try (beg,fin,nxt) s) (transitionsFrom beg (beg,fin,nxt)))

try :: Automation -> String -> Transition -> Bool

try (beg,fin,\_) [] \_ = elem beg fin

try (\_,fin,nxt) s (\_,s2,Eps) = accepts' (s2,fin,nxt) s

try (\_,fin,nxt) (s:st) (\_,s2,C ch) = if s == ch then accepts' (s2,fin,nxt) st else False

-- Задача 8 -----------------------------------------

makeNDA :: RE -> Automation

make :: RE -> Int -> Int -> Int -> ([Transition], Int)

makeNDA re = (1, [2], sort transitions)

where

(transitions, l) = make (simplify re) 1 2 3

make (Term c) beg fin nxt = ([(beg, fin, C c)], nxt)

make Null beg fin nxt = ([(beg, fin, Eps)], nxt)

make (Seq re1 re2) beg fin nxt = ((nxt, nxt + 1, Eps) : s1 ++ s2, l2)

where

(s2, l2) = make re2 (nxt + 1) fin l1

(s1, l1) = make re1 beg nxt (nxt + 2)

make (Alt re1 re2) beg fin nxt = ((beg, nxt, Eps) : (nxt + 1, fin, Eps) : (beg, nxt + 2, Eps) : (nxt + 3, fin, Eps) : s1 ++ s2, l2)

where

(s2, l2) = make re2 (nxt + 2) (nxt + 3) (l1)

(s1, l1) = make re1 nxt (nxt + 1) (nxt + 4)

make (Rep re) beg fin nxt = ((beg, nxt, Eps) : (beg, fin, Eps) : (nxt + 1, fin, Eps) : (nxt + 1, nxt, Eps) : s1, l1)

where

(s1, l1) = make re nxt (nxt + 1) (nxt + 2)

-- Задача 9 -----------------------------------------

getFrontier :: State -> Automation -> [Transition]

getFrontier = undefined

groupTransitions :: [Transition] -> [(Label, [State])]

groupTransitions = undefined

makeDA' :: Automation -> [State] -> [MetaState] -> [MetaTransition]

-> (MetaState, [MetaState], [MetaTransition])

makeDA' = undefined

makeDA :: Automation -> Automation

makeDA = undefined

-------------------------------------------------------

-- showRE - Функція може бути корисною при тестуванні

showRE :: RE -> String

showRE (Seq re re') = showRE re ++ showRE re'

showRE (Alt re re') = "(" ++ showRE re ++ "|" ++ showRE re' ++ ")"

showRE (Rep re) = showRE' re ++ "\*"

showRE (Plus re) = showRE' re ++ "+"

showRE (Opt re) = showRE' re ++ "?"

showRE re = showRE' re

showRE' :: RE -> String

showRE' Null = ""

showRE' (Term c) = [c]

showRE' (Alt re re') = showRE (Alt re re')

showRE' re = "(" ++ showRE re ++ ")"

--------------------------------------------------------

-- Тестові приклади

reFigure, re1, re2, re3, re4, re5 :: RE

reFigure = Seq (Rep (Alt (Term 'a') (Term 'b'))) (Term 'c')

re1 = Seq (Alt (Term 'x') (Term 'y')) (Alt (Term '1') (Term '2'))

re2 = Seq (Term 'x') (Rep (Term '\''))

re3 = Rep (Alt (Seq (Term 'a') (Term 'b')) (Term 'c'))

re4 = Seq (Alt (Term 'a') Null) (Term 'a')

re5 = Seq (Opt (Seq (Term 'a') (Term 'b'))) (Plus (Term 'd'))

ndaFigure, nda1, nda2, nda3, nda4, nda5, ndaTest :: Automation

daFigure, da1, da2, da3, da4, da5 :: Automation

ndaFigure

= (1,[2],[(1,3,Eps),(1,5,Eps),(3,4,Eps),(4,2,C 'c'),(5,7,Eps),

(5,9,Eps),(6,3,Eps),(6,5,Eps),(7,8,C 'a'),(8,6,Eps),

(9,10,C 'b'),(10,6,Eps)])

daFigure

= (1,[2],[(1,1,C 'a'),(1,1,C 'b'),(1,2,C 'c')])

nda1 = (1,[2],[(1,5,Eps),(1,7,Eps),(3,4,Eps),(4,9,Eps),(4,11,Eps),

(5,6,C 'x'),(6,3,Eps),(7,8,C 'y'),(8,3,Eps),(9,10,C '1'),

(10,2,Eps),(11,12,C '2'),(12,2,Eps)])

da1 = (1,[3],

[(1,2,C 'x'),(1,2,C 'y'),(2,3,C '1'),(2,3,C '2')])

nda2 = (1,[2],[(1,3,C 'x'),(3,4,Eps),(4,2,Eps),(4,5,Eps),(5,6,C '\''),

(6,2,Eps),(6,5,Eps)])

da2 = (1,[2],

[(1,2,C 'x'),(2,2,C '\'')])

nda3 = (1,[2],[(1,2,Eps),(1,3,Eps),(3,5,Eps),(3,7,Eps),(4,2,Eps),

(4,3,Eps), (5,9,C 'a'),(6,4,Eps),(7,8,C 'c'),(8,4,Eps),

(9,10,Eps),(10,6,C 'b')])

da3 = (1,[1],

[(1,1,C 'c'),(1,2,C 'a'),(2,1,C 'b')])

nda4 = (1,[2],[(1,5,Eps),(1,7,Eps),(3,4,Eps),(4,2,C 'a'),(5,6,C 'a'),

(6,3,Eps),(7,8,Eps),(8,3,Eps)])

da4 = (1,[2,3],[(1,2,C 'a'),(2,3,C 'a')])

nda5 = (1,[2],[(1,5,Eps),(1,7,Eps),(3,4,Eps),(4,11,C 'd'),(5,9,C 'a'),

(6,3,Eps),(7,8,Eps),(8,3,Eps),(9,10,Eps),(10,6,C 'b'),

(11,12,Eps),(12,2,Eps),(12,13,Eps),(13,14,C 'd'),

(14,2,Eps),(14,13,Eps)])

da5 = (1,[2],[(1,2,C 'd'),(1,3,C 'a'),(2,2,C 'd'),(3,4,C 'b'),

(4,2,C 'd')])

ndaTest = (1, [1], [(1,2, C 'a'), (1,4, Eps), (1,3, C 'b'), (2,3, Eps),

(3,5, Eps), (3,4, C 'a'), (4,4, Eps), (4,1, Eps), (5,2, Eps), (5,4,Eps)] )

------------------10

{-# OPTIONS\_GHC -Wall #-}

module Doroshevych10 where

import Text.ParserCombinators.Parsec

-- Задача 1 -----------------------------------------

evExpr :: String -> Maybe Integer

evExpr s = case (parse pevExpr "" s) of

Right ex -> Just $ read ex

Left \_ -> Nothing

pevExpr :: Parser String

pevExpr = do {n <- parens number;return $ show n}

sign :: Parser String

sign = string "-" <|> pure ""

number :: Parser Int

number = do s <- sign

cs <- many1 digit

return $ read (s ++ cs)

reserved :: String -> Parser ()

reserved s = do { \_ <- string s; spaces}

lexem :: Parser a -> Parser a

lexem p = do {a <- p; spaces ; return a}

parens :: Parser a -> Parser a

parens p = do reserved "("

n <- lexem p

reserved ")"

return n

-- Задача 2 -----------------------------------------

data Expr = Add Expr Expr | Sub Expr Expr

| Mul Expr Expr | Mod Expr Expr | Div Expr Expr

| Var String | Lit Int

deriving (Show, Eq)

fullExpr :: Parser Expr

fullExpr = do spaces;

ex <- expr

eof

return ex

astExpr :: String -> Maybe Expr

astExpr str = case (parse fullExpr "" str) of

Left \_ -> Nothing

Right e -> Just e

int :: Parser Expr

int = do { n <- lexem number; return (Lit n)}

infixOp :: String -> (a -> a -> a) -> Parser (a -> a -> a)

infixOp x f = reserved x >> return f

addop, mulop :: Parser (Expr -> Expr -> Expr)

addop = (infixOp "+" Add) <|> (infixOp "-" Sub)

mulop = (infixOp "\*" Mul) <|> (infixOp "%" Mod) <|> (infixOp "/" Div)

expr, term, factor :: Parser Expr

expr = chainl1 term addop

term = chainl1 factor mulop

factor = int <|> parens expr <|> iden

iden :: Parser Expr

iden = do{n <- letter; m <- many(letter<|>digit);spaces; return (Var (n:m)) }

-- Задача 3 -----------------------------------------

data RE = Null | -- Нуль вираз

Term Char | -- Термінальний символ

Seq RE RE | -- Послідовність

Alt RE RE | -- Альтернатива

Rep RE | -- Повторення (\*)

Plus RE | -- Повторення (+)

Opt RE -- Необов’язкове входження (?)

deriving (Eq, Show)

reg :: Parser RE

reg = do e <- exprRE

\_ <- eof

return e

regExp :: String -> Maybe RE

regExp str = case (parse reg "" str) of

Left \_ -> Nothing

Right rg -> Just rg

termRE :: Parser RE

termRE = chainl1 factRE $ pure Seq

exprRE :: Parser RE

exprRE = chainl1 termRE altop

altop :: Parser (RE -> RE -> RE)

altop = infixOp "|" Alt

symbolRE :: Parser RE

symbolRE = do

с <- symbolRE'

return (Term с)

symbolRE' :: Parser Char

symbolRE' = noneOf "()|\*+?"

primeRE :: Parser RE

primeRE = symbolRE <|> parens exprRE

singleOp :: String -> (a -> a) -> Parser (a -> a)

singleOp x f = do \_ <- string x

return f

factRE :: Parser RE

factRE = try (do s1 <- primeRE

s2 <-(singleOp "+" Plus) <|> (singleOp "\*" Rep) <|> (singleOp "?" Opt)

return (s2 s1) )

<|> do s1 <- primeRE

return s1

-- Задача 4 -----------------------------------------

type Name = String

type Attributes = [(String, String)]

data XML = Text String | Element Name Attributes [XML]

deriving (Eq, Show)

anXML :: String -> Maybe XML

anXML str = case (parse fullXML "" str) of

Left \_ -> Nothing

Right xml -> Just xml

textXML :: Parser XML

textXML = do s <- many1 (noneOf "<>")

return (Text s)

valueXML:: Parser String

valueXML = do s <- many (noneOf "\"")

return s

nameXML :: Parser Name

nameXML = do c <- letter

cs <- many (digit <|> letter <|> oneOf ".-")

return (c:cs)

element :: Parser XML

element = do \_ <- string "<"

nam <- nameXML

att <- many attribute

\_ <- string ">"

x <- many xmlXML

\_ <- string "</"

\_ <- (string nam)

\_ <- string ">"

return (Element nam att x)

attribute :: Parser (String,String)

attribute = do \_ <- spaces

nam <- nameXML

\_ <- spaces

\_ <- string "="

\_ <- spaces

\_ <- string "\""

val <- valueXML

\_ <- string "\""

return (nam,val)

xmlXML :: Parser XML

xmlXML = textXML <|> try element

fullXML :: Parser XML

fullXML = do \_ <- spaces

el <- element

\_ <- spaces

\_ <- eof

return el

------------------------------------------------------

re1, re2, re3, re4, re5 :: RE

re1 = Seq (Alt (Term 'x') (Term 'y')) (Alt (Term '1') (Term '2'))

re2 = Seq (Term 'x') (Rep (Term '\''))

re3 = Rep (Alt (Seq (Term 'a') (Term 'b')) (Term 'c'))

re4 = Seq (Alt (Term 'a') Null) (Term 'a')

re5 = Seq (Opt (Seq (Term 'a') (Term 'b'))) (Plus (Term 'd'))

casablanca :: String

casablanca

= "<film title=\"Casablanca\">\n <director>Michael Curtiz</director>\n <year>1942\

\</year>\n</film>\n\n\n"

casablancaParsed :: XML

casablancaParsed

= Element "film"

[("title","Casablanca")]

[Text "\n ",

Element "director" [] [Text "Michael Curtiz"],

Text "\n ",

Element "year" [] [Text "1942"],

Text "\n"]