module Arbore (Tree,

adauga,

cauta,

ini,

parcurgere,

elem\_nivele\_impare

)

where

data Tree a = Leaf

| Node a (Tree a) (Tree a)

deriving Show

-- tree for testing

root :: Tree Int

root = (Node 7 (Node 3 (Node 1 Leaf Leaf) (Node 5 Leaf Leaf)) (Node 10 Leaf Leaf))

-- 7

-- / \

-- 3 10

-- / \

-- 1 5

adauga :: Ord a => a -> Tree a -> Tree a

adauga a = f

where

f Leaf = Node a Leaf Leaf

f (Node v left right) | a == v = Node a left right

| a <= v = Node v (f left) right

| otherwise = Node v left (f right)

cauta :: Ord a => a -> Tree a -> Maybe a

cauta a = f

where

f Leaf = Nothing

f (Node v left right) | a == v = (Just v)

| a < v = f left

| otherwise = f right

ini :: Ord a => [a] -> Tree a

ini [] = Leaf

ini (x:xs) = adauga x (ini xs)

-- parcurgere (ini [1,5,2,8,10,3,11,6,7]) == [1,2,3,5,6,7,8,10,11]

parcurgere :: Tree a -> [a]

parcurgere Leaf = []

parcurgere (Node v left right) = (parcurgere left) ++ [v] ++ (parcurgere right)

aux :: Int -> Tree a -> [a]

aux \_ Leaf = []

aux niv (Node v left right ) = if mod niv 2 == 0 then (aux (niv+1) left) ++ (aux (niv+1) right)

else [v] ++ (aux (niv+1) left) ++ (aux (niv+1) right)

elem\_nivele\_impare :: Ord a => Tree a -> [a]

elem\_nivele\_impare Leaf = []

elem\_nivele\_impare arb = aux 0 arb

module Principal where

import Arbore

main :: IO () -- [Int]

main = do

text <- putStrLn "Introdu numerele:"

numere\_string <- getLine

let result = words numere\_string

let numere\_int = map (read) result::[Int]

let arbore = ini numere\_int

--return (parcurgere arbore)

putStrLn (unwords (map (show) (parcurgere arbore)))

maybeToList :: Maybe a -> [a]

maybeToList Nothing = []

maybeToList (Just a) = [a]

-- Exercise 7

toList :: Ord k => Keymap k a -> [(k,a)]

toList Leaf = []

toList (Node k a stanga dreapta) = toList stanga ++ [(k,a)] ++ toList dreapta

-- Exercise 8

set :: Ord k => k -> a -> Keymap k a -> Keymap k a

set key value = f

where

f Leaf = Node key value Leaf Leaf

f (Node k v left right) | key == k = Node k value left right

| key <= k = Node k v (f left) right

| otherwise = Node k v left (f right)

newtype Keymap k a = K [(k,a)]

size :: Eq k => Keymap k a -> Int

size (K xs) = length xs

del :: Eq k => k -> Keymap k a -> Keymap k a

del key (K xs) = K (filter ((/=key).fst) xs)

data Exp = Var String -- Variabilă x

| Val Int -- Întreg 5

| Op Exp String Exp -- Operație 3 <= x

| If Exp Exp Exp -- Conditional if (3 <= x) then 3 else x

| Lambda String Exp -- Funcție anonimă \ x -> 3 + x

deriving Show

subExp :: Exp -> [Exp]

subExp (Var \_) = []

subExp (Val \_) = []

subExp (Op exp1 s exp2) = [exp1, exp2]

subExp (If exp1 exp2 exp3) = [exp1, exp2, exp3]

subExp (Lambda s exp) = [exp]

aggExps :: Exp -> [Exp] -> Exp

aggExps e es = if length (subExp e) /= length es then e

else f e es

where

f (Op exp1 s exp2) [exp3, exp4] = Op exp3 s exp4

f (If exp1 exp2 exp3) [exp4, exp5, exp6] = If exp4 exp5 exp6

f (Lambda s exp) [exp1] = Lambda s exp1

foldExp :: (Exp -> [b] -> b) -> Exp -> b

foldExp f e = f e (map (foldExp f) (subExp e))

transform :: (Exp -> Exp) -> Exp -> Exp

transform f (Var x) = f (Var x)

transform f (Val x) = f (Val x)

transform f exp = aggExps exp (map (transform f) (subExp exp))

freeVars :: Exp -> [String]

freeVars (Var x) = [x]

freeVars (Val x) = []

freeVars (Op e1 op e2) = freeVars e1 ++ freeVars e2

freeVars (If e1 e2 e3) = freeVars e1 ++ freeVars e2 ++ freeVars e3

freeVars (Lambda v exp) = filter (/= v) (freeVars exp)