Métodos de Programação I

Trabalho Prático nº 1

Miguel dos Santos Esteves - Nº 43202
Nuno Miguel Mendonça Coutinho Correia - Nº 43179
Tiago Coutinho Correia - nº43179

16 de Novembro de 2006

Resumo

Este trabalho consiste numa pequena aplicação que efectua a compilaçao e execução de uma linguagem *Prog*.

Seguidamente apresentamos o código do trabalho explicando passo a passo o seu algoritmo

Conteúdo

Parte I
Inicialização

Neste parte limitamo-nos a fazer o import de algumas bibliotecas necessárias ao longo do programa e a implementar os tipos de dados fornecidos no enunciado.

```
module Trabalho1 where


import GHC.Exception   as ExceptionBase hiding (catch)
import Control.Exception (try)
import List (sort)
import Char
import Control.Monad.State
import Control.Monad.Reader
import Control.Monad.Error
import Data.Map
import Prelude hiding (getLine,putStr)

data Prog o = Print (Exp o) | Let String (Exp o) | Seq (Prog o) (Prog o)

data Exp o = Const Int | Var String | Op o [Exp o] deriving (Ord,Eq)

class Opt o where
    arity :: o -> Int
    func :: o -> ([Int] -> Int)

data Ops = Add | Mul | Sim | Div | Addl | Mull | Pot | Eq deriving (Ord,Eq)


instance Opt Ops where
    arity Add = 2
    arity Mul = 2
    arity Div = 2
    arity Sim = 1
    arity Addl = 3
    arity Mull = 3
    arity Pot = 2
    arity Eq = 2
    func Add = (\[x,y] -> x + y)
    func Mul = (\[x,y] -> x * y)
    func Div = (\[x,y] -> div x y)
    func Sim = (\[x] -> -x)
    func Addl = (\l -> sum l)
    func Mull = (\l -> mul l)
        where mul [] = 0
              mul [x] = x
              mul (x:xs) = x*mul xs
    func Pot = (\[x,y] -> x ^ y)
    func Eq = (\[x,y] -> if (x==y) then 1 else 0)
```

```
data Instr o = PUSH Int | LOAD | STORE | IN | OUT | OP o
    deriving (Show, Read)

type VarDict = Map String Int

type MSP o = [Instr o ]

data Mem = Mem{stack :: [Int], heap :: [Int]}

data Mem2 = Mem2{stack2 :: [Int], heap2 :: [M Int]}

data M a = Null | So a deriving (Eq,Read,Ord)
```

1 Instâncias da classe Show

Foram instanciados os tipos de dados, nomeadamente *Ops, Exp e Prog* no âmbito da mostragem da linguagem. A finalidade deste método não mais resolve senão uma parte dos problemas de inter-acção com o utilizador.

1.1 Instância Show da Linguagem Prog

```
instance (Show o, Opt o) => Show (Prog o) where
    show (Print e) = "print "++ show e
    show (Let s e) = "let "++s++" = "++show e
    show (Seq a b) = show a ++"; "++show b
```

1.2 Instância Show do das Expressoes Exp

```
instance (Show o, Opt o) => Show (Exp o) where
    show (Const x) = show x
    show (Var x) = x
    show (Op o x) | arity o == 1 && length x == 1 = "("++show o ++ show (head x)++")"
                  | arity o == 2 && length x == 2 = "("++show (head x) ++ show o ++ show (last x)++")"
                  | arity o > 2 = " "++show o ++ "[" ++ f x ++ "] "
                        where f [] = ""
                              f (x:xs) = show x ++ "," ++ f xs
```

1.3 Instância Show dos Operadores Ops

```
instance Show Ops where
    show Add = "+"
    show Mul = "*"
    show Sim = "-"
    show Div = "/"
    show Addl = "+:"
    show Mull = "*:"
    show Pot = "^"
```

```
    show Eq = "="
```

1.4 Instância Show do tipo da Memoria Mem

```
instance Show Mem where
    show (Mem a b) = "Stack : " ++ show a ++ "\nHeap : "

instance Show Mem2 where
    show (Mem2 a b) = "Stack : " ++ show a ++ "\nHeap : " ++ show b
```

1.5 Instância Show M a

```
instance Show a => Show (M a) where
    show Null = "N"
    show (So x) = show x
```

2 Instância da classe Read

A existência deste método evolui a percepção do programa ao nível do utilizador estando entretanto mais próximo do conceito de linguagem aritmética empírica humana. Constiste portanto na utilização de uma linguagem mais intuitiva e próxima do utilizador para o entender.

2.1 Instância Read da Linguagem Prog

```
instance  (Read o, Opt o, Ord o) => Read (Prog o)  where
    readsPrec _ s = [((juntaP.lexL) s,"")]

juntaP ::  (Read o,Opt o, Ord o) => [String] -> Prog o
juntaP q | head x=="print" && numvirg q==0 = ((printprog.tail) x)
         | head x=="let" && numvirg q==0 = ((letprog.tail) x)
         | head x=="let" = Seq (juntaP x) ((juntaP.concat) xs)
         | head x=="print" = Seq (juntaP x) ((juntaP.concat) xs)
           where (x:xs) = s
                 s = separaP q

separaP :: [String] -> [[String]]
separaP q = f ";" q []
    where  f _ [] l = [l]
           f e (x:xs) l | e==x = l:xs:[]
                        | e/=x = f e xs (l++[x])

letprog :: (Read o, Opt o, Ord o) => [String] -> Prog o
letprog (x:y:xs) | isVar x && y=="=" = (Let x (read (concat xs)))

printprog :: (Read o, Opt o, Ord o) => [String] -> Prog o
printprog x = Print (read (concat x))
```

```
numvirg :: [String] -> Int
numvirg s = f s 0
    where f [] n = n
          f (x:xs) n | x==";" = f xs (n+1)
                     | x/=";" = f xs n
```

2.2 Instância Read da Expressão Exp

```
instance (Read o,Opt o,Ord o) => Read (Exp o) where
    readsPrec _ s = [((junta.str2exp) s,"")]


readOp :: (Read o,Opt o) => String -> Exp o
readOp s | (length z==2 && isOp x) && (arity o)==1 = Op o [(readOp.concat) xs]
         | (length z==3 && isOp (xs!!0)) && (arity o2)==2 = Op o2 [readOp x, readOp (xs!!1)]
         | isInt x && length z== 1 = Const i
         | isVar x && length z== 1  = Var x
           where z@(x:xs) = lexL s
                 i = read x :: Int
                 o = read x
                 o2 = read (head xs)


------------------funcoes de listas para read de Exp o-------------


-- divisao lexica
--lexL "((-3)*(4+x))" -> ["(","(","-","3",")","*","(","4","+","x",")",")"]
lexL :: String -> [String]
lexL [] = []
lexL s = s1:(lexL s2)
    where [(s1,s2)] = lex s


-- ..[(2,1),(1,(#+#)),(2,2),(0,(#+#)),(1,3)] -> ((1+2)+3)
junta ::  (Read o,Opt o, Ord o) => [(Int, Exp o)] -> Exp o
junta q | lenOp e==2 && length q>1= insOp [(junta.head) s, (junta.last) s]  e
    | lenOp e==1 && length q>1= insOp [(junta.last) s] e
    | lenOp e==0 && length q>1= insOp ((j.last) s) e
    | otherwise = e
        where (i,e) = (head.sort) q
              s = separa q

--j :: [(Int, Exp o)] -> [Exp o]
j [] = []
j (x:xs) = [(junta [x])] ++ (j xs)


-- ...[Const 1, Var "x"] -> Op [_] -> Op [Const 1, Var "x"]
insOp :: [Exp o] -> Exp o -> Exp o
insOp s (Op o x) = Op o s


-- ...Op [Const 1, Var "x"] -> 2--
lenOp :: Exp o -> Int
```

```
lenOp (Op o x) = length x
lenOp _ = -1

-- ...[2,3,1,6] -> f 1 [...] [] -> [[2,3],[6]]
separa :: Ord a => [a] -> [[a]]
separa q = f ((head.sort) q) q []
    where f _ [] l = [l]
          f e (x:xs) l | e==x = l:xs:[]
                       | e/=x = f e xs (l++[x])

-- "(1+2)+3" -> [(2,1),(1,(#+#)),(2,2),(0,(#+#)),(1,3)]
str2exp :: (Read o,Opt o) => String -> [(Int,Exp o)]
str2exp = f.str2par
    where f [] = []
          f ((i,s):xs) | isOp s && n==1 = (i,Op o [Var "#"]):(f xs)
                       | isOp s && n==2 = (i,Op o [Var "#",Var "#"]):(f xs)
                       | isOp s && n>2 = (i,Op o []):(f xs)
                       | isVar s || isInt s = (i,readOp s):(f xs)
                    where o = read s
                          n = arity o

-- .. "(1+2)+3" -> [(2,"1"),(1,"+"),(2,"2"),(0,"+"),(1,"3")]
str2par :: String -> [(Int,String)]
str2par = cat.str2p.putpar.lexL

-- ..[(1,"x"),(1,"y"),..] -> [(1, "xy"),..]
cat :: [(Int,String)] -> [(Int,String)]
cat [x] = [x]
cat ((i,s):(i2,s2):xs) | s=="," = cat ((i2,s2):xs)
                       | i==i2 = cat ((i,s++s2):xs)
                       | i/=i2 = (i,s):(cat ((i2,s2):xs))

-- ..(lexL"(1+2)+3") -> ["(","1","+","2",")","+","3"] -> [(1,"1"),(1,"+"),(1,"2"),(0,"+"),(0,"3")]
str2p :: [String] -> [(Int,String)]
str2p s = f s 0
    where f [] _ = []
          f (x:xs) n | x=="(" = f xs (n+1)
                     | x==")" = f xs (n-1)
                     | otherwise = (n,concat (x:[])):(f xs n)

-- mete parenteses a volta do elemento
--(bug fix): cat junta Ops com valores "+3" -> com parenteses-> "+(3)"
putpar :: [String] -> [String]
putpar [] = []
putpar (x:xs) | isOp x = x:(putpar xs)
              | otherwise = "(":x:")":(putpar xs)

isOp :: String -> Bool
isOp x = not(isInt x || isVar x) && (length.lexL) x ==1

isInt :: String -> Bool
isInt [] = True
isInt (x:xs) | elem x ['0'..'9'] = isInt xs
             | otherwise = False

isVar :: String -> Bool
isVar (x:xs) | (ord x>96 && ord x<123) || (ord x>64 && ord x<90) = True
             | otherwise = False
```

2.3 Instância Read de Ops

```
instance Read Ops where
    readsPrec _ = readsOps

readsOps :: ReadS Ops
readsOps s | s=="+"= [(Add,"")]
           | s=="*"= [(Mul,"")]
           | s=="-"= [(Sim,"")]
           | s=="/"= [(Div,"")]
           | s=="+:"= [(Addl,"")]
           | s=="*:"= [(Mull,"")]
           | s=="^"= [(Pot,"")]
           | s=="="= [(Eq,"")]
```

Parte II
Funções principais

3 Funcao execmsp

Esta função é responsável pela execução de um programa. Recebe portanto uma série de comandos de baixo nível (MSP).

```
--------------EXECMSP--------------

pos :: [a] -> Int -> a -> [a]
pos x n v | length x== length res = res
   where res = f [] x n v
         f z x 0 v = z++v:(drop 1 x)
         f z (x:xs) n v = f (z++[x]) xs (n-1) v

exeInstr :: Opt o => Mem -> Instr o -> IO Mem
exeInstr (Mem a b) (PUSH x) = return (Mem (x:a) b)
exeInstr (Mem (a:e:es) b) STORE = return (Mem es (pos b e a))
exeInstr (Mem (a:as) b) LOAD = return (Mem ((b!!a):as) b)
exeInstr (Mem a b) IN = putStr "valor da variavel ambigua: " >> getInt >>= \x -> return (Mem (x:a) b)
exeInstr (Mem (a:as) b) OUT = print a >> return (Mem as b)
exeInstr (Mem a b) (OP o) = return (Mem ( (func o (take n a)):(drop n a)) b)
                where n = arity o

execmsp :: Opt o => MSP o -> IO ()
execmsp p = evalStateT (aux p) emptymem
```

```
    where  aux :: Opt o => MSP o -> StateT Mem IO ()
           aux [] = return ()
           aux (x:xs) = do mem <- get
                           mem2 <- lift (exeInstr mem x)
                           put mem2
                           aux xs


execmsp2 :: (Opt o, Show o) => MSP o -> IO (Either String (M Int))
execmsp2 p = runErrorT $ evalStateT (aux p) emptymem2
    where  aux :: (Opt o, Show o) => MSP o -> StateT Mem2 (ErrorT String IO) (M Int)
           aux [] = return Null
           aux [x] = do mem <- get
                        if (execErr x mem/="") then throwError $ "barraca: "++execErr x mem
                                              else lift.lift $ putStr ""
                        return (So (head (leStack2 mem)))
           aux (x:xs) = do mem <- get
                           if (execErr x mem/="") then throwError $ "barraca: "++execErr x mem
                                                  else lift.lift $ putStr ""
                               mem2 <- lift.lift $ exeInstrE mem x
                               put mem2
                               aux xs



-------------EXECMSP ERROR-----------------------

exeInstrE :: Opt o => Mem2 -> Instr o -> IO Mem2
exeInstrE (Mem2 a b) (PUSH x) = return (Mem2 (x:a) b)
exeInstrE (Mem2 (a:e:es) b) STORE = return (Mem2 es (pos b e (So a)))
exeInstrE (Mem2 (a:as) b) LOAD = return (Mem2 ((readM (b!!a)):as) b)
exeInstrE (Mem2 a b) IN = putStr "valor da variavel ambigua: " >> getInt >>= \x -> return (Mem2 (x:a) b)
exeInstrE (Mem2 (a:as) b) OUT = print a >> return (Mem2 as b)
exeInstrE (Mem2 a b) (OP o) = return (Mem2 ( (func o (take n a)):(drop n a)) b)
                where n = arity o


execErr :: (Opt o, Show o) => Instr o -> Mem2 -> String
execErr x mem | show x=="OP /" && leStack2(mem)!!1 ==0 = "divisao por zero"
          | ((head.lexL) s=="OP" || s=="STORE" || s=="LOAD" || s=="OUT") && leStack2(mem) == []
                                                                                = "stack vazia"
          | s=="LOAD" && not(vazHeap(leHeap2(mem))) = "heap vazia"
          | ((head.lexL) s=="PUSH" || s=="LOAD" || s=="IN" ) && length(leStack2(mem))==tam_stack
                                                                                = "stack overflow"
          | s=="STORE" && (length(leStack2(mem))<2 || (leStack2(mem)!!1 < 0 || leStack2(mem)!!1
                                                  >= length(leHeap2(mem)) ) ) = "falha de segmentacao"
          | s=="LOAD" && ((h < 0 || h >= length(leHeap2(mem))) || (leHeap2(mem))!!h == Null )
                                                                                = "falha de segmentacao"
          | otherwise = ""
        where s=show x
              h=leStack2(mem)!!0



execmspE :: (Opt o, Show o) => MSP o -> IO (Either String ())
execmspE p = runErrorT $ evalStateT (aux p) emptymem2
    where aux :: (Opt o, Show o) => MSP o -> StateT Mem2 (ErrorT String IO) ()
          aux [] = return ()
```

```
         aux (x:xs) = do mem <- get
                     if (execErr x mem/="") then throwError $ "barraca: "++execErr x mem
                                        else lift.lift $ putStr ""
                         mem2 <- lift.lift $ exeInstrE mem x
                         put mem2
                         aux xs


printE :: Show a => Either String a -> IO ()
printE (Left a) = putStrLn a
printE (Right a) = return ()


getE :: Either String (M Int) -> M Int
getE (Left a) = Null
getE (Right a) = a



-----------VERSAO PARA MEM (com undefined)----

execErr' :: (Opt o, Show o) => Instr o -> Mem -> String
execErr' x mem | show x=="OP /" && leStack(mem)!!1 ==0 = "divisao por zero"
           | ((head.lexL) s=="OP" || s=="STORE" || s=="LOAD" || s=="OUT") && leStack(mem) == []
                                                                                = "stack vazia"
           | ((head.lexL) s=="PUSH" || s=="LOAD" || s=="IN" ) && length(leStack(mem))==tam_stack
                                                                                = "stack overflow"
           | s=="STORE" && (length(leStack(mem))<2 || (leStack(mem)!!1 < 0 || leStack(mem)!!1
                                                   >= length(leHeap(mem)) ) ) = "falha de segmentacao"
           | otherwise = ""
        where s=show x
              h=leStack(mem)!!0


execmspE' :: (Opt o, Show o) => MSP o -> IO (Either String ())
execmspE' p = runErrorT $ evalStateT (aux p) emptymem
    where aux :: (Opt o, Show o) => MSP o -> StateT Mem (ErrorT String IO) ()
          aux [] = return ()
          aux (x:xs) = do mem <- get
                 if (execErr' x mem/="") then throwError $ "barraca: "++execErr' x mem
                                   else lift.lift $ putStr ""
                 let h = leStack(mem)!!0
                 n <- lift.lift $ evalU $ (leHeap(mem))!!h
                 if (show x=="LOAD" && ((h < 0 || h >= length(leHeap(mem))) || n==1 ))
                                                  then throwError $ "barraca: falha de segmentacao"
                                                  else lift.lift $ putStr ""
                 mem2 <- lift.lift $ exeInstr mem x
                 put mem2
                 aux xs

execmsp2' :: (Opt o, Show o) => MSP o -> IO (Either String (M Int))
execmsp2' p = runErrorT $ evalStateT (aux p) emptymem
    where  aux :: (Opt o, Show o) => MSP o -> StateT Mem (ErrorT String IO) (M Int)
           aux [] = return Null
           aux [x] = do mem <- get
                if (execErr' x mem/="") then throwError $ "barraca: "++execErr' x mem
                                  else lift.lift $ putStr ""
                let h = leStack(mem)!!0
```

```
                n <- lift.lift $ evalU $ (leHeap(mem))!!h
                if (show x=="LOAD" && ((h < 0 || h >= length(leHeap(mem))) || n==1 ))
                                                then throwError $ "barraca: falha de segmentacao"
                                                else lift.lift $ putStr ""
                mem2 <- lift.lift $ exeInstr mem x
                return (So (head (leStack mem)))
           aux (x:xs) = do mem <- get
                   if (execErr’ x mem/="") then throwError $ "barraca: "++execErr’ x mem
                                          else lift.lift $ putStr ""
                   let h = leStack(mem)!!0
                   n <- lift.lift $ evalU $ (leHeap(mem))!!h
                   if (show x=="LOAD" && ((h < 0 || h >= length(leHeap(mem))) || n==1 ))
                                                    then throwError $ "barraca: falha de segmentacao"
                                                    else lift.lift $ putStr ""
                   mem2 <- lift.lift $ exeInstr mem x
                   put mem2
                   aux xs
```

3.1 Funçao execmspDEBUG

Esta função é responsável pela execução de um programa com debugging.

```
------------------DEBUG EXECMSP-----------------

execmspDEBUG :: (Show o, Opt o) => MSP o -> IO ()
execmspDEBUG p = evalStateT (aux 1 p) emptymem2
    where  aux :: (Show o, Opt o) => Int -> MSP o -> StateT Mem2 IO ()
           aux _ [] = return ()
           aux n (x:xs) = do mem <- get
                 lift (putStr ("\nITERACAO " ++ (show n) ++ "\n"))
                 mem2 <- lift (exeInstrDB mem x)
                 lift (putStr ("\nMemoria actual:\n"))
                 lift (print mem2)
                 put mem2
                 lift (getLine)
                 aux (n+1) xs


exeInstrDB :: (Show o, Opt o) => Mem2 -> Instr o -> IO Mem2
exeInstrDB (Mem2 a b) z@(PUSH x) = putStr("("++show z++"): Coloca "++show x++" no topo da Stack\n")
                    >> return (Mem2 (x:a) b)
exeInstrDB (Mem2 (a:e:es) b) z@STORE = putStr("("++show z++"): Coloca "++show a++" no endereco "
                            ++show e++" da Heap \n")
                    >> return (Mem2 es (pos b e (So a)))
```

```
exeInstrDB (Mem2 (a:as) b) z@LOAD = putStr("("++show z++"): Subsitui "++show a++" do topo da Stack por "
                              ++show (b!!a)++" do endereco " ++show a++" da Heap \n")
                    >> return (Mem2 ((readM (b!!a)):as) b)
exeInstrDB (Mem2 a b) z@IN =  putStr("("++show z++"): Coloque o valor a inserir no topo da Stack:")
                    >> getInt >>= \x -> return (Mem2 (x:a) b)
exeInstrDB (Mem2 (a:as) b) z@OUT =  putStr("("++show z++"): Retira "++show a++" do topo da Stack
                                                                                 para o ecra:\n")
                    >> print a >> return (Mem2 as b)
exeInstrDB (Mem2 a b) z@(OP o) =  putStr("("++show z++"):  "++show o++"("++ (if n==1 then show  (head a)
                                                                               else f (take n a))
                                   ++") = " ++ show res++" guardado no topo da Stack\n")
                    >> return (Mem2 ( res:(drop n a)) b)
                                where n = arity o
                                      res = func o (take n a)
                                      f [] = []
                                      f [x] = show x
                                      f (x:xs) = show x ++ "," ++ f xs

execmspDEBUGE :: (Show o, Opt o) => MSP o -> IO (Either String ())
execmspDEBUGE p = runErrorT $ evalStateT (aux 1 p) emptymem2
    where  aux :: (Show o, Opt o) => Int -> MSP o -> StateT Mem2 (ErrorT String IO) ()
           aux _ [] = return ()
           aux n (x:xs) = do mem <- get
                 lift.lift $ putStr ("\nITERACAO " ++ (show n) ++ "\n")
                     if (execErr x mem/="") then throwError $ "barraca: "++execErr x mem
                                        else lift.lift $ putStr ""
                 mem2 <- lift.lift $ exeInstrDB mem x
                 lift.lift $ putStr "\nMemoria actual:\n"
                 lift.lift $ print mem2
                 put mem2
                 lift.lift $ getLine
                 aux (n+1) xs




------------------VERSAO PARA MEM (undefined)---------------

exeInstrDB' :: (Show o, Opt o) => Mem -> Instr o -> IO Mem
exeInstrDB' (Mem a b) z@(PUSH x) = putStr("("++show z++"): Coloca "++show x++" no topo da Stack\n")
                    >> return (Mem (x:a) b)
exeInstrDB' (Mem (a:e:es) b) z@STORE = putStr("("++show z++"): Coloca "++show a++" no endereco "
                              ++show e++" da Heap \n")
                    >> return (Mem es (pos b e a))
exeInstrDB' (Mem (a:as) b) z@LOAD = putStr("("++show z++"): Subsitui "++show a++" do topo da Stack por "
                              ++show (b!!a)++" do endereco " ++show a++" da Heap \n")
                    >> return (Mem ((b!!a):as) b)
exeInstrDB' (Mem a b) z@IN =  putStr("("++show z++"): Coloque o valor a inserir no topo da Stack:")
                    >> getInt >>= \x -> return (Mem (x:a) b)
exeInstrDB' (Mem (a:as) b) z@OUT =  putStr("("++show z++"): Retira "++show a++" do topo da Stack
                                                                                 para o ecra:\n")
                    >> print a >> return (Mem as b)
exeInstrDB' (Mem a b) z@(OP o) =  putStr("("++show z++"):  "++show o++"("++ (if n==1 then show (head a)
                                                                               else f (take n a))
                                   ++") = " ++ show res++" guardado no topo da Stack\n")
                    >> return (Mem ( res:(drop n a)) b)
                                where n = arity o
                                      res = func o (take n a)
```

```
                                    f [] = []
                                    f [x] = show x
                                    f (x:xs) = show x ++ "," ++ f xs
```

```
execmspDEBUGE' :: (Show o, Opt o) => MSP o -> IO (Either String ())
execmspDEBUGE' p = runErrorT $ evalStateT (aux 1 p) emptymem
    where  aux :: (Show o, Opt o) => Int -> MSP o -> StateT Mem (ErrorT String IO) ()
           aux _ [] = return ()
           aux n (x:xs) = do mem <- get
                 lift.lift $ putStr ("\nITERACAO " ++ (show n) ++ "\n")
                     if (execErr' x mem/="") then throwError $ "barraca: "++execErr' x mem
                                       else lift.lift $ putStr ""
                 let h = leStack(mem)!!0
                 n <- lift.lift $ evalU $ (leHeap(mem))!!h
                 if (show x=="LOAD" && ((h < 0 || h >= length(leHeap(mem))) || n==1 ))
                                       then throwError $ "barraca: falha de segmentacao"
                                       else lift.lift $ putStr ""
                 mem2 <- lift.lift $ exeInstrDB' mem x
                 lift.lift $ putStr "\nMemoria actual:\n"
                 lift.lift $ showHeapU mem2
                 put mem2
                 lift.lift $ getLine
                 aux (n+1) xs
```

```
--------------------------------------------------------
```

4 Funcao compile

Esta função compila programas escritos em Prog devolvendo uma série de comandos sobre a memória (MSP).

```
---------------COMPILE-------------

compile :: Opt o => Prog o -> MSP o
compile p = evalState (aux p) empty
  where aux :: Opt o => Prog o -> State VarDict (MSP o)
    aux (Print e) = do vd <- get
               return ((aval vd e)++[OUT])
    aux (Let v e) = do vd <- get
               svd <- return (size vd)
                       put (insert v svd vd )
```

```
              return ([PUSH svd]++(aval vd e)++[STORE])
    aux (Seq l r )= do p <- aux l
                      q <- aux r
              return (p++q)

aval :: Opt o => VarDict -> Exp o -> MSP o
aval vd (Const x) = [PUSH x]
aval vd (Var v) | member v vd = [PUSH (vd!v), LOAD]
        | otherwise = [IN]
aval vd (Op o l) = (mapa (aval vd) (reverse l))++[OP o]
    where mapa f [] = []
          mapa f (a:x) = (f a) ++ (mapa f x)
```

```
-------------COMPILE ERROR----------------------


compErr :: Opt o => Exp o -> String
compErr (Op o l) | length l/= arity o = "operador com numero de argumantos errado"
         | otherwise =""
compErr _ = ""


compileE :: Opt o => Prog o -> IO (Either String (MSP o))
compileE p = runErrorT $ evalStateT (aux p) empty
  where aux :: Opt o => Prog o -> StateT VarDict (ErrorT String IO) (MSP o)
    aux (Print e) = do vd <- get
              if (compErr e /="") then throwError $ "barraca: "++ compErr e
                              else lift.lift $ putStr ""
              return ((aval vd e)++[OUT])
    aux (Let v e) = do vd <- get
              svd <- return (size vd)
                      put (insert v svd vd )
              return ([PUSH svd]++(aval vd e)++[STORE])
    aux (Seq l r )= do p <- aux l
                      q <- aux r
              return (p++q)
```

```
----------------COMPILER-READER-STATE------------------------
compiler :: Opt o => Prog o -> MSP o
compiler p = evalState (aux p) empty
  where aux :: Opt o => Prog o -> State VarDict (MSP o)
    aux (Print e) = do vd <- get
              return ((runReader (avalr e) vd)++[OUT])
    aux (Let v e) = do vd <- get
              svd <- return (size vd)
```

```
                       put (insert v svd vd )
               return ([PUSH svd]++(runReader (avalr e) vd)++[STORE])
    aux (Seq l r )= do p <- aux l
                       q <- aux r
               return (p++q)


avalr :: Opt o => Exp o -> Reader VarDict (MSP o)
avalr (Const x) = return ([PUSH x])
avalr (Var v) = ask >>= \vd -> return ([PUSH (vd!v), LOAD])
avalr (Op o l) = do z <- (mapM avalr (reverse l))
            return ((concat z)++[OP o])
```

4.1 Funçao compileDEBUG

Esta funçao deve ser responsavel por compilar com Debug um Prog para MSP

```
------------------DEBUG COMPILE-----------------


compileDEBUG :: (Show o, Opt o) => Prog o -> IO (MSP o)
compileDEBUG p = evalStateT (aux p) empty
  where aux :: (Show o, Opt o) => Prog o -> StateT VarDict IO (MSP o)
    aux z@(Print e) = do vd <- get
                 lift(putStr ("\n("++show e++") -> "))
                 res <- lift (avalDB vd e)
                 lift(putStr ("\n"++show z ++" -> "++show(res++[OUT])++"\n"))
                 lift(getLine)
                 return (res++[OUT])
    aux z@(Let v e) = do vd <- get
                 svd <- return (size vd)
                         put (insert v svd vd )
                 vd1 <- get
                 lift(putStr ("\n("++show e++") -> "))
                 res <- lift (avalDB vd e)
                 lift(putStr ("\n("++show z++") ->" ++show([PUSH svd]++res++[STORE])++"\n"))
                 lift(putStr ("##VarDict actual: "++show vd1++"\n"))
                 lift(getLine)
                 return ([PUSH svd]++res++[STORE])
    aux (Seq l r )= do p <- aux l
                       q <- aux r
               return (p++q)

avalDB :: (Show o, Opt o) => VarDict -> Exp o -> IO (MSP o)
avalDB vd (Const x) = putStr(show (PUSH x::Instr Ops)++", ") >> return ([PUSH x])
avalDB vd (Var v) | member v vd = putStr(show (PUSH (vd!v)::Instr Ops)++","
                                        ++show (LOAD::Instr Ops)++" , ")>> return ([PUSH (vd!v), LOAD])
          | otherwise = return [IN]
avalDB vd (Op o l) = do putStr(show (OP o)++" {")
           z <- (mapM (avalDB vd) (reverse l))
```

```
            putChar('\b')
            putChar('\b')
            putStr("}")
            getLine
            return ((concat z)++[OP o])

---------------------------------------------------------------
```

Parte III
Funções Secundarias

5 Funçao sim

Funçao que simplifica uma expressão.

```
-----------------SIMPL-----------------


exVar :: Exp Ops -> Bool
exVar l = elem True (f l)
    where f (Const x) = [False]
          f (Var v) = [True]
          f (Op o l) = concat (Prelude.map f l)


sim :: Exp Ops -> Exp Ops
sim x | exVar x && simp x==x = x
      | exVar x = sim (simp x)
      | otherwise = x

simp :: Exp Ops -> Exp Ops
simp (Var v) = Var v
simp (Const c) = Const c
simp (Op Pot [a,b]) = (Op Pot [a,b])
simp (Op Sim [Op Sim[s]]) = s
simp (Op Sim [s]) | exVar s && s==Op Add[q,w] = Op Add [Op Sim [q], Op Sim [w]]
          | exVar s = Op Sim [simp s]
          | not(exVar s) = Op Sim [s]
                   where Op Add [q,w]= s
simp (Op Add[a,b]) | exVar a && exVar b && a==b = Op Mul[simp a,Const 2]
           | exVar a && exVar b && a/=b && a==Op Sim[s] && b==s = Const 0
           | exVar a && exVar b && a/=b && b==Op Sim[s1] && a==s1 = Const 0
```

```
           | exVar a && exVar b && a/=b && a==Op Add [q,w] && exVar q && not(exVar w)
                                                       = Op Add[simp (Op Add[simp b,q]),simp w]
           | exVar a && exVar b && a/=b && a==Op Add [q,w] && not(exVar q) && exVar w
                                                       = Op Add[simp (Op Add[simp b,w]),simp q]
           | exVar a && exVar b && a/=b && b==Op Add [q1,w1] && exVar q1 && not(exVar w1)
                                                       = Op Add[simp (Op Add[simp a,q1]),simp w1]
           | exVar a && exVar b && a/=b && b==Op Add [q1,w1] && not(exVar q1) && exVar w1
                                                       = Op Add[simp (Op Add[simp a,w1]),simp q1]
           | exVar a && exVar b && a==Op Mul [m1,m2] && exVar m1 && not(exVar m2) && b==m1
                                                       = Op Mul [m1,Op Add[m2,Const 1]]
           | exVar a && exVar b && a==Op Mul [m1,m2] && not(exVar m1) && exVar m2 && b==m2
                                                       = Op Mul [m2,Op Add[m1,Const 1]]
           | exVar a && exVar b && b==Op Mul [n1,n2] && exVar n1 && not(exVar n2) && a==n1
                                                       = Op Mul [n1,Op Add[n2,Const 1]]
           | exVar a && exVar b && b==Op Mul [n1,n2] && not(exVar n1) && exVar n2 && a==n2
                                                       = Op Mul [n2,Op Add[n1,Const 1]]
           | exVar a && exVar b && a==Op Mul [m1,m2] && exVar m1 && not(exVar m2)
                 && b==Op Mul [n1,n2] && exVar n1 && not(exVar n2) && m1==n1 = Op Mul [m1,Op Add[m2,n2]]

           | exVar a && exVar b && a==Op Mul [m1,m2] && not(exVar m1) && exVar m2
                 && b==Op Mul [n1,n2] && exVar n1 && not(exVar n2) && m2==n1 = Op Mul [m2,Op Add[m1,n2]]

           | exVar a && exVar b && a==Op Mul [m1,m2] && not(exVar m1) && exVar m2
                 && b==Op Mul [n1,n2] && not(exVar n1) && exVar n2 && m2==n2 = Op Mul [m2,Op Add[m1,n1]]

           | exVar a && exVar b && a==Op Mul [m1,m2] && exVar m1 && not(exVar m2)
                 && b==Op Mul [n1,n2] && not(exVar n1) && exVar n2 && m1==n2 = Op Mul [m1,Op Add[m2,n1]]

           | exVar a && exVar b && a/=b = Op Add[simp a,simp b]
           | exVar a && not(exVar b) = Op Add [simp a, b]
           | not(exVar a) && exVar b = Op Add [a, simp b]
           | not(exVar a) && not(exVar b) = Op Add [a, b]
                                   where Op Add [q,w]= a
                                         Op Add [q1,w1]= b
                                         Op Sim [s] = a
                                         Op Sim [s1] = b
                                         Op Mul [m1,m2] = a
                                         Op Mul [n1,n2] = b
simp (Op Mul [a,b]) | exVar a && exVar b && a==b = Op Pot[simp a,Const 2]
            | exVar a && exVar b && a/=b && a==Op Mul [q,w] && exVar q && not(exVar w)
                                                               = Op Mul[simp (Op Mul[simp b,q]),simp w]
            | exVar a && exVar b && a/=b && a==Op Mul [q,w] && not(exVar q) && exVar w
                                                               = Op Mul[simp (Op Mul[simp b,w]),simp q]
            | exVar a && exVar b && a/=b && b==Op Mul [q1,w1] && exVar q1 && not(exVar w1)
                                                               = Op Mul[simp (Op Mul[simp a,q1]),simp w1]
            | exVar a && exVar b && a/=b && b==Op Mul [q1,w1] && not(exVar q1) && exVar w1
                                                               = Op Mul[simp (Op Mul[simp a,w1]),simp q1]
            | exVar a && exVar b && a/=b = Op Mul[simp a,simp b]
            | exVar a && not(exVar b) = Op Mul [simp a, b]
            | not(exVar a) && exVar b = Op Mul [a, simp b]
            | not(exVar a) && not(exVar b) = Op Mul [a, b]
                                   where Op Mul [q,w]= a
                                         Op Mul [q1,w1]= b



equa :: Exp Ops -> Exp Ops
```

```
equa (Op Eq [a,b]) | exVar a && not(exVar b) = igua (sim a) b
           | not(exVar a) && exVar b  = igua (sim b) a


igua :: Exp Ops -> Exp Ops -> Exp Ops
igua (Var v) x = x
igua (Op Sim [s]) x   | exVar s = igua s (Op Sim [x])
igua (Op Add [a,b]) x | exVar a && not(exVar b) = igua a (Op Add[x,Op Sim[b]])
              | exVar b && not(exVar a) = igua b (Op Add[x,Op Sim[a]])
              | exVar a && exVar b && a==b = igua (Op Mul[a,Const 2]) x
igua (Op Mul [a,b]) x | exVar a && not(exVar b) = igua a (Op Div[x,b])
              | exVar b && not(exVar a) = igua b (Op Div[x,a])
igua (Op Div [a,b]) x | exVar a = igua a (Op Mul[x,b])
              | exVar b = igua b (Op Mul[x,a])
```

6 Funções IO e outras

```
-----------------MEMORIA---------------
-----STACK--------------------HEAP-----

emptymem :: Mem
emptymem = Mem{stack = [], heap = replicate 10 undefined}
tam_stack = 10

emptymem2 :: Mem2
emptymem2 = Mem2{stack2 = [], heap2 = replicate 10 Null}

readM :: M a -> a
readM (So a) = a

leStack :: Mem -> [Int]
leStack (Mem a b) = a

leHeap :: Mem -> [Int]
leHeap (Mem a b) = b

leStack2 :: Mem2 -> [Int]
leStack2 (Mem2 a b) = a

leHeap2 :: Mem2 -> [M Int]
leHeap2 (Mem2 a b) = b

evalHeap :: [M Int] -> M Int
evalHeap [] = Null
evalHeap (x:xs) | (isInt.show) x = x
                | otherwise = evalHeap xs

vazHeap :: [M Int] -> Bool
vazHeap [] = False
```

```
vazHeap (x:xs) | (isInt.show) x = True
               | otherwise = vazHeap xs



showHeapU :: Mem -> IO ()
showHeapU mem = do s <- f (leHeap mem) ""
           putStr $ show mem ++"["++s++"\b]\n"
    where f [] s = return s
          f (x:xs) s0 = do n <- evalU x
                       let s = s0 ++ if n==1 then "undef," else show x++","
                   f xs s

evalU :: Int -> IO Int
evalU x = do z <- Control.Exception.try (putStr $ show x++"\b\b\b\b\b")
         return (if (head.lexL.show) z=="Right" then 0 else 1)



--excepU : Int -> IO Bool
excepU x = do catchException (print x) (\e -> print 0)
          return 1


-------original------
eval :: Opt o => VarDict -> Exp o -> Int
eval vd (Const x) = x
eval vd (Var v) = vd!v
eval vd (Op o l) = func o (Prelude.map (eval vd) l)

execprog :: Opt o => Prog o -> Int
execprog p = evalState (aux p) empty
   where aux :: Opt o => Prog o -> State VarDict Int
     aux (Print e) = do vd <- get
                        return (eval vd e)
     aux (Let v e) = do vd <- get
                        put (insert v (eval vd e) vd )
                return 0
     aux (Seq l r )= do aux l
                        aux r





--IO--

main :: IO ()
main = maib Null (Const 0)

maib :: M Int -> Exp Ops -> IO ()
maib it a = do putStr ">"
           z <- getLine
           let p = " "++z
           case ((head.lexL) p) of
                   "ajuda"   -> do putStrLn ">\tsair\n>\texecmsp\n>\tdebug\n>\tit\n>\tsp\n
                                                                    >\tsimp\n>\tresolve\n"
                           putStrLn "escreva um programa (e. g. let x = 3 ; print x*3+2 )"
                           maib it a
```

```
"sair"    -> return ()

"it"      -> do excep $ print it
        maib it a

"sp"      -> do print a
        maib it a

"execmsp" -> do let r = read (drop 8 p) :: MSP Ops
        --execmsp m
        m''<-execmsp2' r
        excep $ printE m''
        let m' = getE m''
        excep $ print m'
        maib m' a

"debug"   -> do let r = read (drop 6 p) :: Prog Ops
        m <- compileDEBUG r
        excep $ print m
        m' <- execmspDEBUGE' m
        excep $ printE m'
        maib it a

"resolve" -> do let r = read (drop 8 p) :: Exp Ops
        let sp = equa r
        excep $ print sp
        let m = compile (Print sp)
        excep $ print m
        --execmsp m
        m''<-execmsp2' m
        excep $ printE m''
        let m' = getE m''
        excep $ putStrLn ("resultado da variavel = "++show m')
        maib m' sp

"simp"    -> do let r = read (drop 5 p) :: Exp Ops
        let sp = sim r
        excep $ print sp
        maib it sp

"print"   -> do let r' = if (it/=Null) then " let it = "++show (readM it)++" ;"++p
                   else p
        let r'' = if (a/=(Const 0)) then " let sp = "++show a++" ;"++r'
                        else r'
        let r = read r'' :: Prog Ops
        let m = compile r
        excep $ print m
        --execmsp m
        m''<-execmsp2' m
        excep $ printE m''
        let m' = getE m''
        excep $ print m'
        maib m' a

"let"     -> do let r' = if (it/=Null) then " let it = "++show (readM it)++" ;"++p
                   else p
        let r'' = if (a/=(Const 0)) then " let sp = "++show a++" ;"++r'
```

```
                                else r'
                    let r = read r'' :: Prog Ops
                    let m = compile r
                    excep $ print m
                    --execmsp m
                    m''<-execmsp2' m
                    excep $ printE m''
                    let m' = getE m''
                    excep $ print m'
                    maib m' a

            otherwise -> maib it a

       where excep x = catchException x (\e -> putStr "Erro nao definido" >> getLine >> maib it a)


getInt :: IO Int
getInt = getLine >>= (\s -> return (read s))

getLine :: IO String
getLine = getChar >>=
         \c -> if c == '\n'
           then return ""
           else getLine >>= (\s -> return (c:s))

putStr :: String -> IO ()
putStr [] = return ()
putStr (h:t) = putChar h >>= \_ -> putStr t
```

Parte IV
Testes

7 exemplos

```
--Exemplos--

exp1 = Op Mul [Const 2,Var "x"]

prog :: Prog Ops
prog = Seq (Let "x" (Const 2))
            (Print (Op Mul [Op Sim [Const 3],
                            Op Add [Const 4, Var "x"]]))

mspprog :: MSP Ops
mspprog = [PUSH 0, PUSH 2, STORE , PUSH 3, OP Sim,
            PUSH 4, PUSH 0, LOAD, OP Add, OP Mul , OUT ]
```

```
t::VarDict
t = insert "y" 1 (insert "x" 2 empty)
s1=Seq (Let "x" (Const 2)) (Let "y" (Var "x") )
s2=Seq (Let "x" (Const 2)) (Let "y" (Op Add [Var "x",Const 1]) )
s3=Seq (Let "x" (Const 2))
       (Seq (Let "y" (Var "x"))
        (Print (Var "y"))  )


tes :: Prog Ops -> IO ()
tes s = do let e= compile s
           print e
           z<-(execmsp e)
           return ()
```