

Métodos de Programação I



Universidade do Minho, LESI

Ano lectivo 2006/2007

Trabalho Prático N°1

Mário Ulisses Pires Araujo Costa - 43175 Rami Galil Shashati - 43166

Vasco Almeida Ferreira - 43207

15 de Novembro de 2006

Resumo

Este trabalho foi implementado no paradigma funcional, em linguagem **Haskell**.
O objectivo é desenvolver um compilador para uma pequena linguagem de programação.

Conteúdo

1	Introdução	2
2	Análise e Especificação	2
2.1	Descrição informal do problema	2
2.2	Especificação dos Requisitos	3
2.2.1	Requisitos fundamentais	3
3	Concepção/Desenho da Resolução	4
3.1	ExecProg	4
3.2	Estruturas de Dados e respectivas instâncias	4
3.2.1	Prog o	4
3.2.2	Exp o	7
3.2.3	Ops	8
3.2.4	Instr o	8
3.2.5	Tipos de dados auxiliares	8
3.3	Estrutura da Aplicação	9
3.3.1	Compilação	9
3.3.2	Execução	10
3.3.3	Memória	11
4	Testes	13
4.1	Exemplos de prog's e msp's	13

1 Introdução

Código Haskell

```

1 {-# OPTIONS -fglasgow-exts #-}
2 --
3 -- Métodos de Programação I
4 -- LESI, Universidade do Minho
5 --
6 -- Compilador para a linguagem Prog
7 --
8 -- 2006/2007
9 --
10 -- autores:
11 -- Mário Ulisses Pires Araujo Costa - 43185
12 -- Rami Galil Shashati - 43166
13 -- Vasco Almeida Ferreira - 43207
14 --
15
16 module Trabalho1 where
17
18 import Data.Map
19 import Control.Monad.State
20 import Control.Monad.Error
21 import Data.List
22 import Maybe
23 import Char
24 import IO
25
26 infixr 5 >|<

```

Este Trabalho foi realizado no âmbito da disciplina de Métodos de Programação I, da Licenciatura em Engenharia de Sistemas e Informática da Universidade do Minho.

O Objectivo deste trabalho é desenvolver um compilador para uma pequena linguagem de programação designada **Prog**.

Este relatório contém as diferentes fases de desenvolvimento do aplicação, a saber: análise do problema, escolha/descoberta de algoritmos, implementação, testes e documentação.

2 Análise e Especificação

Conteúdo

2.1	Descrição informal do problema	2
2.2	Especificação dos Requisitos	3
2.2.1	Requisitos fundamentais	3

2.1 Descrição informal do problema

Nesta linguagem só existem três instruções sobre expressões. Estas podem ser constituídas por variáveis, inteiros ou ambos. Sempre que neste relatório nos referirmos a *expressões* deve ser considerada a definição acima.

Lista das instruções aceites pelo programa e sua breve descrição:

- **let** $x = e$, onde x é uma variável e e uma expressão.
- **print** e , que imprime a expressão e .
- **if** exp **then** p_1 **else** p_2 , onde exp é uma expressão lógica sobre variáveis e/ou inteiros e p_1 e p_2 ou são **print's** ou são **let's**.

2.2 Especificação dos Requisitos

Para levar a cabo este trabalho, foram necessários conhecimentos da linguagem **Haskell**, do funcionamento da linguagem **MSP** assim como o domínio de $\text{\LaTeX} 2_{\epsilon}$ e alguns dos seus pacotes para a elaboração deste relatório.

2.2.1 Requisitos fundamentais

Na base de desenvolvimento deste trabalho estão presentes duas ideias fundamentais. A primeira delas, consiste na "passagem" da linguagem descrita informalmente no item acima para outra linguagem chamada **MSP**. A esta passagem iremos, a partir de agora, informalmente, apelidar de *compilação*. O segundo ponto consiste na "transformação" do resultado da *compilação* numa *computação*. Esta fase, por sua vez, será denominada por *execução de um programa MSP*.

3 Concepção/Desenho da Resolução

Conteúdo

3.1 ExecProg	4
3.2 Estruturas de Dados e respectivas instâncias	4
3.2.1 Prog o	4
3.2.2 Exp o	7
3.2.3 Ops	8
3.2.4 Instr o	8
3.2.5 Tipos de dados auxiliares	8
3.3 Estrutura da Aplicação	9
3.3.1 Compilação	9
3.3.2 Execução	10
3.3.3 Memória	11

3.1 ExecProg

Todo o código listado abaixo irá ser necessário para que consigamos usar a função ($> | <$), que é a base para a resolução do nosso problema.

Código Haskell

```

28 main :: IO()
29 main = do
30     putStr "Escreva o nome de um ficheiro: "
31     f <- getLine
32     ff <- readFile f
33     let prog = (read ff :: Prog Ops)
34         (>|<) compile execMSP prog
35     putStr "prima qualquer tecla para mostrar o codigo msp: "
36     c <- getChar
37     case compile prog of
38         (Right a) -> putStrLn $ show a
39         (Left a)  -> putStrLn a
40     return()
41
42 (>|<) :: (Eq o , Opt o)
43         => (Prog o -> Either String (MSP o))
44         -> (MSP o -> IO ())
45         -> Prog o -> IO ()
46 (>|<) comp exec prog = case comp prog of
47         (Left e)  -> print e
48         (Right m) -> exec m

```

3.2 Estruturas de Dados e respectivas instâncias

De seguida, vamos mostrar as estruturas de dados base deste trabalho e suas instâncias, nomeadamente;

3.2.1 Prog o

Código Haskell

```

49 data Prog o = If (Exp o) (Prog o) (Prog o)
50   | Print (Exp o)
51   | Let String (Exp o)
52   | Seq (Prog o) (Prog o)
53   | Ciclo (Prog o) (Exp o) (Prog o) (Prog o)
54   | Prints String
55
56 progExemplo :: Prog Ops
57 progExemplo = Seq
58   (Seq
59     (Seq
60       (Seq
61         (Seq
62           (Let "wee" (Const 10))
63           (Let "weee" (Const 20))
64         )
65         (If (Op AND_ [Var "x",Var "y"])
66           (Print (Var "x"))
67           (Print (Const 90)))
68       )
69       (Let "z" (Op Div [Const 2,Op Mul [Op Add [Var "x",Var "y"],Const 5]]))
70     )
71     (Print (Var "z"))
72   )
73   (If (Op AND_ [Var "x",Var "y"])
74     (Print (Const 101010))
75     (Print (Var "y")))
76   (Ciclo (Let "i" (Const 0)) (Op LE_ [Var "i",Const 10]) (Print (Var "i")) (Let "i" (Op

```

Acima mostramos a representação de um exemplo pertencente a (Prog o).

O exemplo seguinte corresponde ao modo como a linguagem deve ser escrita por um programador **Prog** num ficheiro texto.

```

let wee = 10;
let weee = 20;
if (x && y) then print x else print 90;
let z = (2/((x+y)*5));
print z;
if (x && y) then print 101010 else print y;

```

$$leti = 0; (i \leq 10); leti = (i + 1)$$

```
print i;
```

A única classe que necessitamos para a realização do trabalho é a classe `Opt`. Esta obriga tudo o que seja instância dela a responder as funções nela definidas.

Código Haskell

```

78 class Opt o where
79   arity :: o -> Int
80   func  :: o -> ([Int] -> Int)
81
82 instance Opt Ops where
83   arity Add = 2
84   arity Mul = 2
85   arity Sim = 1
86   arity Sub = 2
87   arity Div = 2
88
89   arity OR_ = 2
90   arity AND_ = 2
91   arity NOT_ = 1
92   arity GE_ = 2
93   arity GT_ = 2
94   arity LE_ = 2
95   arity LT_ = 2
96   arity EQ_ = 2
97   arity NE_ = 2
98

```

```

99     func Add = \[x,y] -> x + y
100    func Mul = \[x,y] -> x * y
101    func Sim = \[x]   -> -x
102    func Sub = \[x,y] -> x - y
103    func Div = \[x,y] -> x 'div' y
104
105    func OR_  = \[x,y] -> if (x /= 0) || (y /= 0) then 1 else 0
106    func AND_ = \[x,y] -> if (x /= 0) && (y /= 0) then 1 else 0
107    func NOT_ = \[x]   -> if x == 0 then 1 else 0
108    func GE_  = \[x,y] -> if x >= y then 1 else 0
109    func GT_  = \[x,y] -> if x > y then 1 else 0
110    func LE_  = \[x,y] -> if x <= y then 1 else 0
111    func LT_  = \[x,y] -> if x < y then 1 else 0
112    func EQ_  = \[x,y] -> if x == y then 1 else 0
113    func NE_  = \[x,y] -> if x /= y then 1 else 0

```

Instância de Show para o tipo de dados Prog o.

Código Haskell

```

114 instance (Opt o, Show o) => Show (Prog o) where
115     show (Print a)      = "print " ++ show a
116     show (Let s a)     = "let " ++ s ++ " = " ++ show a
117     show (Seq l r)     = show l ++ show r
118     show (If exp p1 p2) = "if (" ++ show exp ++ ")" ++
119                           "then (" ++ show p1 ++ ")" ++
120                           "else (" ++ show p2 ++ ");\n"
121     show (Ciclo inic exp c1 fim) = "[" ++ show inic ++
122                                     ", " ++ show exp ++ ";" ++
123                                     show fim ++ "]\n\t" ++ show c1 ++ "\n"
124     show (Prints e)    = "prints " ++ show e ++ "\n"

```

É apresentada agora a solução para a instância **Read** do data **Prog Ops**;

Código Haskell

```

125 instance Read (Prog Ops) where
126     readList = reads' . readsProg
127     readsPrec _ s =
128         let
129             removeComents :: String -> String
130             removeComents = let remove l = [x | x <- l , (fst $ (\[1] -> 1) x) /= "#"]
131                             in init . unlines . Prelude.map unwords . concat .
132                                 Prelude.map (\(a,b) -> [a++b]) . Prelude.map unzip .
133                                 remove . Prelude.map lex . lines
134             read' :: String -> Prog Ops
135             read' = ins . reverse . fst . (\[1] -> 1) . reads' . readsProg
136         in [(read' $ removeComents s,[])]
137
138
139 ins :: [Prog Ops] -> Prog Ops
140 ins [e] = e
141 ins (h:t) = Seq (ins t) h
142
143 reads' :: [(Prog Ops,String)] -> [(Prog Ops,String)]
144 reads' [(a,b)] = [(a++c,fim) | (c,fim) <- readsProg b]
145
146 readsProg :: ReadS [Prog Ops]
147 readsProg s | s == [] = [([],[])]
148             | otherwise = [(h:t,rest) | ("let",p1) <- lex s,
149                                     (h,fim) <- readsProgLetPrint s,
150                                     (t,rest) <- readsProg fim ] ++
151             [(h:t,rest) | ("print",p1) <- lex s,
152                                     (h,p2) <- readsProgLetPrint s,
153                                     (t,rest) <- readsProg p2 ] ++
154             [((If exp_ h t):tt ,rest) | ("if",p1) <- lex s,
155                                     (exp_ ,p2) <- readsExp p1,
156                                     ("then",p3) <- lex p2,
157                                     (h,p4) <- readsProgThen p3,
158                                     ("else",p5) <- lex p4,

```

```

159         (t,p6)      <- readsProgLetPrint p5,
160         (tt,rest)  <- readsProg p6      ] ++
161     [((Ciclo i cond (ins p9) inc),p10) |
162         ("[" ,p1)   <- lex s,
163         (i,p2)     <- readsProgThen p1,
164         (",",p3)   <- lex p2,
165         (cond,p4)  <- readsExp p3,
166         (",",p5)   <- lex p4,
167         (inc,p6)   <- readsProgThen p5,
168         ("]",p7)   <- lex p6,
169         ("{" ,p8)  <- lex p7,
170         (p9,p10)  <- readsProg p8      ] ++
171         [[],p1) | ("}",p1) <- lex s      ]
172     where
173     readsProgLetPrint :: ReadS (Prog Ops)
174     readsProgLetPrint s =
175         [(Print exp_,rest) | ("print",p1) <- lex s,
176         (exp_,p2) <- readsExp p1,
177         (",",rest) <- lex p2      ] ++
178         [(Let val exp2_,rest) | ("let",p1) <- lex s,
179         (val,p2) <- lex p1,
180         ("=",exp_) <- lex p2,
181         (exp2_,p3) <- readsExp exp_,
182         (",",rest) <- lex p3      ]
183     readsProgThen :: ReadS (Prog Ops)
184     readsProgThen s =
185         [(Print exp_,rest) | ("print",p1) <- lex s,
186         (exp_,rest) <- readsExp p1      ] ++
187         [(Let val exp2_,rest) | ("let",p1) <- lex s,
188         (val,p2) <- lex p1,
189         ("=",exp_) <- lex p2,
190         (exp2_,rest) <- readsExp exp_ ]

```

3.2.2 Exp o

Código Haskell

```

191 data Exp o = Const Int
192           | Var String
193           | Op o [Exp o]
194
195 instance (Show o,Opt o) => Show (Exp o) where
196     show (Const n) = show n
197     show (Var s)   = s
198     show (Op o l) | arity o == 2 = "(" ++ (show $ head l) ++ show o ++
199                                     (show $ last l) ++ ")"
200     | arity o == 1 = "(" ++ show o ++ (show $ head l) ++ ")"
201
202 instance Read (Exp Ops) where
203     readsPrec _ s = readsExp s
204
205 readsExp :: ReadS (Exp Ops)
206 readsExp s =
207     [((leOp "~" [a]),p4) | ("(",p1) <- lex s,
208     ("~",p2) <- lex p1,
209     (a,p3) <- readsExp p2,
210     (")",p4) <- lex p3] ++
211     [((leOp op [a,b]),p5) | ("(",p1) <- lex s,
212     (a,p2) <- readsExp p1,
213     (op,p3) <- lex p2,
214     op == "+" || op == "*" || op == "/" ||
215     op == "-" || op == "||" || op == "&&" ||
216     op == "==" || op == "!=" || op == ">=" ||
217     op == "<=" || op == "!" || op == ">" ||
218     op == "<",
219     (b,p4) <- readsExp p3,
220     (")",p5) <- lex p4      ] ++
221     [((Const ((read a)::Int)),sx) | (a,sx) <- lex s, all isDigit a] ++
222     [((Var a),sx) | (a,sx) <- lex s, all isAlpha a]
223     where

```

```

224 leOp :: String -> [Exp Ops] -> Exp Ops
225 leOp o = Op (read o::Ops)

```

3.2.3 Ops

Código Haskell

```

227 data Ops = Add
228           | Mul
229           | Sim
230           | Sub
231           | Div
232           | OR_
233           | AND_
234           | NOT_
235           | GE_
236           | GT_
237           | LE_
238           | LT_
239           | EQ_
240           | NE_
241   deriving Eq
242
243 instance Show Ops where
244   show Add = "+"
245   show Mul = "*"
246   show Sim = "~"
247   show Sub = "-"
248   show Div = "/"
249
250   show OR_ = "||"

```

```

251   show AND_ = "&&"
252   show NOT_ = "!"
253   show GE_ = ">="
254   show GT_ = ">"
255   show LE_ = "<="
256   show LT_ = "<"
257   show EQ_ = "=="
258   show NE_ = "!="
259
260 instance Read Ops where
261   readsPrec _ "+" = [(Add, [])]
262   readsPrec _ "-" = [(Sub, [])]
263   readsPrec _ "*" = [(Mul, [])]
264   readsPrec _ "/" = [(Div, [])]
265   readsPrec _ "~" = [(Sim, [])]
266   readsPrec _ "||" = [(OR_, [])]
267   readsPrec _ "&&" = [(AND_, [])]
268   readsPrec _ "!" = [(NOT_, [])]
269   readsPrec _ ">=" = [(GE_, [])]
270   readsPrec _ ">" = [(GT_, [])]
271   readsPrec _ "<=" = [(LE_, [])]
272   readsPrec _ "<" = [(LT_, [])]
273   readsPrec _ "==" = [(EQ_, [])]
274   readsPrec _ "!=" = [(NE_, [])]

```

3.2.4 Instr o

Código Haskell

```

275 data Instr o = PUSH Int
276              | LOAD
277              | STORE
278              | IN
279              | OUT
280              | OP o
281              | JMP String
282              | JMPF String
283              | HALT
284              | Label String
285              | OUTC
286   deriving Eq
287
288 instance (Show o) => Show (Instr o) where
289   show (PUSH n) = "\tPUSH " ++ show n ++ "\n"
290   show (OP op) = "\tOP " ++ show op ++ "\n"
291   show (LOAD) = "\tLOAD" ++ "\n"
292   show (STORE) = "\tSTORE" ++ "\n"
293   show (IN) = "\tIN" ++ "\n"
294   show (OUT) = "\tOUT" ++ "\n"
295   show (HALT) = "\tHALT" ++ "\n"
296   show (JMP s) = "\tJMP " ++ s ++ "\n"
297   show (JMPF s) = "\tJMPF " ++ s ++ "\n"
298   show (Label s) = "\t " ++ s ++ "\n"
299
300   showList [] _ = []
301   showList (h:t) _ = show h ++ showList t []

```

3.2.5 Tipos de dados auxiliares

Código Haskell

```

302 data Mem = Mem {stack :: [Int] , heap :: [Maybe Int]}
303 type MSP o = [Instr o]
304 type VarDict = Map String Int

```

3.3 Estrutura da Aplicação

O compilador de **Prog** tem como base duas funções fundamentais, a saber:

3.3.1 Compilação

A função **compile** tem como objectivo a "passagem" do tipo de dados **Prog o** para **MSP o**.

Esta função tem contradomínio *Either String (MSP o)* pelo simples facto dela gerar código **MSP o** correcto e pronto a ser executado pela função **execMSP** propagar um erro de compilação (atraves do uso de uma String).

A função auxiliar à *compile*, *compile_* tem contradomínio *State VarDict (Either String (MSP o))* para garantir que mantemos o nosso dicionário de variáveis associadas a valores sempre actualizado. Estas variáveis nada mais são do que as String's que são usadas aquando da declaração das mesmas na linguagem **Prog**, por exemplo:

```

let n1 = 10;
let n2 = 20;
let n3 = 30;

```

Note-se que o tipo de dados de *VarDict* corresponde ao mapeamento de String com o respectivo valor.

A melhor estratégia que encontramos para propagar erros foi através do operador de combinador de monads (\ll). Com isto deixamos de ter a preocupação das verificações exaustivas de computações falhadas. Para isto usamos uma função fundamental que avalia as expressões, retornando *Left Erro* em caso de insucesso e *Right MSP o* caso contrário.

De seguida lista-se o código referente à função acima descrita:

Código Haskell

```

305
306 compile :: (Eq o , Opt o) => Prog o -> Either String (MSP o)
307 compile p = evalState (compile' p) empty
308   where
309     compile' :: (Eq o , Opt o) => Prog o -> State VarDict (Either String (MSP o))
310     compile' (Print e) = do vd <- get
311                           return (evalExp vd e >>= (\x -> return (x ++ [OUT])))
312     compile' (Let x e) = do vd <- get
313                           if (Data.Map.member x vd)
314                             then return ()
315                             else put $ Data.Map.insert x (succ (Data.Map.size vd)) vd
316                           vd' <- get
317                           return (evalExp vd e
318                                   >>= (\m -> return ([PUSH (vd' ! x)] ++
319                                                         m ++ [STORE])))
320     compile' (Seq x y) = compile' x >>= (\x -> compile' y >>=
321     (\y -> return (x >>= (\x1 -> y >>=
322     (\y1 -> return (x1 ++ y1)))))
323     compile' (If exp_ p1 p2) = do vd <- get
324                               let evale = evalExp vd exp_
325                               compile' p1 >>= (\x -> compile' p2 >>=
326     (\y -> return (x >>= (\then_ -> y >>=
327     (\else_ -> evale >>=
328     (\eval -> return (eval ++ [JMPF ("#THEN")] ++
329     then_ ++ [(JMP "#FIM")] ++

```

```

330                                     else_ ++ [(Label "#FIM")])))))))
331 compile' (Ciclo (Let x e) exp_ c (Let _ e1)) =
332     do vd <- get
333       put (Data.Map.insert x 1024 vd)
334       vd <- get
335       compile' c >>= (\cicl -> return (cicl >>= (\ciclo -> evalExp vd exp_ >>=
336         (\cond -> evalExp vd e >>= (\exp1 -> evalExp vd e1 >>=
337           (\exp2 -> return ( [PUSH 1024] ++ exp1 ++ [STORE] ++ [Label "#CICLO"] ++
338             cond ++ [JMPF "#FIM"] ++ ciclo ++ [PUSH 1024] ++ exp2 ++
339             [STORE] ++ [JMP "#CICLO"])))))))))
340 compile' (Prints s) = return $ return $ concat $ Prelude.map (\c -> [PUSH (ord c)] ++ [OUTC]) s
341
342 evalExp      :: Opt o => VarDict -> Exp o -> Either String (MSP o)
343 evalExp _ (Const x) = return [PUSH x]
344 evalExp vd (Var x)  | (Data.Map.member x vd) = return ([PUSH (vd ! x)] ++ [LOAD])
345                   | otherwise = Left ("!! Erro Compilacao -> Variavel " ++ x ++ " nao declarada!")
346 evalExp vd (Op o l) | arity o == 1 = evalExp vd (head l) >>= (\x -> return (x ++ [OP o]))
347                   | arity o == 2 = let eval1 = evalExp vd (head l)
348                                   eval2 = evalExp vd (l!!1)
349                                   in eval1 >>= (\x -> eval2 >>= (\y -> return (x ++ y ++ [OP o]))

```

3.3.2 Execução

```
aux :: (Eq o, Show o, Opt o) => MSP o -> StateT Mem (ErrorT String IO) ()
```

O objectivo da `aux` é conseguirmos um transformador de monads tanto de State como de IO como de Erro.

Através do uso do monad StateT conseguimos ir actualizando a memória do compilador.

Já o monad responsável pela propagação de erros, neste caso monad ErrorT, possibilita-nos a utilização da função `throwError` que nos permite pegar numa String (um erro) e encapsulá-lo dentro de um monad.

O uso do monad IO faz com que possamos imprimir os resultados que vamos obtendo, quer estes sejam erros ou resultados válidos provenientes de uma computação.

A nossa função que usa a `aux` e que executa código (MSP o) tem como assinatura:

```
execMSP :: (Eq o, Show o, Opt o) => MSP o -> IO ()
```

Ou seja, pega num (MSP o) e executa a função `runErrorT` que devolve um (m Either e a) que irá ser fundamental para a função acima descrita.

De seguida lista-se o código referente à função acima descrita:

Código Haskell

```

350 execMSP      :: (Eq o, Show o, Opt o) => MSP o -> IO ()
351 execMSP msp = do eith <- runErrorT (evalStateT (aux msp) emptyMem)
352                case eith of
353                  (Left a)  -> print a
354                  (Right b) -> if b /= ()
355                              then print b
356                              else putStr ""
357
358 where
359   aux      :: (Eq o, Show o, Opt o) => MSP o -> StateT Mem (ErrorT String IO) ()
360   aux []   = return ()
361   aux (h:t) =
362     case h of
363       (PUSH x) -> do mem <- get
364                    put $ pushOrin mem x
365                    aux t
366       (IN)     -> do mem <- get
367                    x <- lift $ lift $ getLine
368                    put $ pushOrin mem $ (read x :: Int)

```

```

368         aux t
369     (STORE) -> do mem <- get
370             case store mem of
371             (Left e) -> throwError e
372             (Right m) -> do put m
373                        aux t
374     (LOAD)  -> do mem <- get
375             case load mem of
376             (Left e) -> throwError e
377             (Right m) -> do put m
378                        aux t
379     (OUT)   -> do mem <- get
380             case checkMem mem 1 of
381             False ->
382                 do let err = "!! Erro Execucao -> Out: not enough arguments for function
383                    throwError err
384                 True  -> do lift $ lift $ putStrLn $ show $ head $ stack mem
385                        put $ out mem
386                        aux t
387     (OP o)  -> do mem <- get
388             case op mem o of
389             (Left e) -> throwError e
390             (Right m) -> do put m
391                        aux t
392     (JMPF s) -> do mem <- get
393             case checkMem mem 1 of
394             False ->
395                 do let err = "!! Erro Execucao -> Jmpf: not enough arguments for function
396                    throwError err
397                 True  -> if ((head $ stack mem) /= 0)
398                        then aux t
399                        else aux $ tail $ dropWhile (/=(JMP "#FIM")) t
400             put(out mem)
401     (HALT)  -> return()
402     (JMP "#FIM") -> do aux $ tail $ dropWhile (/=(Label "#FIM")) t
403     (Label "#CICLO") -> do aux $ takeWhile (/=(JMPF "#FIM")) t
404                        mem <- get
405                        if (head $ stack mem) == 0
406                        then aux $ tail $ dropWhile (/=(JMP "#CICLO")) t
407                        else do aux $ takeWhile (/=(JMP "#CICLO")) $
408                               tail $ dropWhile (/=(JMPF "#FIM")) t
409                        aux (h:t)
410     (Label s) -> do aux t
411     (OUTC)   -> do mem <- get
412             case checkMem mem 1 of
413             False ->
414                 do let err = "!! Erro Execucao -> OutC: not enough arguments for functio
415                    throwError err
416                 True  -> do lift $ lift $ putStr $ (:) (chr $ head $ stack mem) []
417                        put $ out mem
418                        aux t

```

3.3.3 Memória

Código Haskell

```

419 emptyMem :: Mem
420 emptyMem = Mem {stack = [], heap = replicate 1024 Nothing}

```

A memória sendo representada por uma lista de inteiros á qual designamos stack e uma lista de Maybe Int chamada heap, necessita das seguintes funções básicas que vão trabalhar sobre ela, nestas temos já em atenção muitos dos possíveis erros que podem surgir quando se executa **MSP**.

Código Haskell

```

421 injh :: Mem -> [Maybe Int] -> Mem
422 injh = (\mem l -> Mem {stack = stack mem, heap = l})
423
424 injS :: Mem -> [Int] -> Mem

```

```

425 injs = (\mem l -> Mem {stack = l, heap = heap mem})
426
427 pushOrin      :: Mem -> Int -> Mem
428 pushOrin mem i = injs mem $ (:) i $ stack mem
429
430 store        :: Mem -> Either String Mem
431 store mem | not(checkMem mem 2) = Left "!! Erro Execucao -> Store: not enough arguments for function l
432         | otherwise           = Right (injh mem' ((take (e - 1) hm) ++ [Just v] ++ (drop e hm)))
433     where
434         hm = heap mem
435         sm = stack mem
436         v  = head sm
437         e  = head $ tail sm
438         mem' = injs mem $ drop 2 sm
439
440 load         :: Mem -> Either String Mem
441 load mem | not(checkMem mem 1) = Left "!! Erro Execucao -> Load: not enough arguments for function l
442         | (heap mem !! (end-1)) == Nothing = Left "!! Erro Execucao -> Load: variavel nao declarada
443         | otherwise = Right (injs mem $ fromJust (b !! (end - 1)) : t)
444     where
445         s = stack mem
446         end = head s
447         t = tail s
448         b = heap mem
449
450 out         :: Mem -> Mem
451 out mem = (injs mem $ tail $ stack mem)
452
453 op         :: (Opt o, Show o, Eq o) => Mem -> o -> Either String Mem
454 op mem o | not(checkMem mem ar) = Left "!! Erro Execucao -> Op: not enough arguments for function o
455         | otherwise = if (show o == "/" ) && ((last $ reverse $ tk) == 0)
456         then Left "!! Erro Execucao -> Op: Div 0 !"
457         else Right (injs mem ((func o $ reverse $ tk) : dr))
458     where
459         ar = arity o
460         s = stack mem
461         dr = drop ar s
462         tk = take ar s

```

4 Testes

Conteúdo

4.1 Exemplos de prog's e msp's	13
--------------------------------------	----

4.1 Exemplos de prog's e msp's

Código Haskell

```

464 mspprog' = [PUSH 6, PUSH 3, OP Div, PUSH 1024, PUSH 0, STORE, Label "#CICLO", PUSH 1024, LOAD, PUSH
465
466
467 checkMem :: Mem -> Int -> Bool
468 checkMem mem n = let stck = stack mem
469                   in length stck >= n
470
471 exp1 :: Exp Ops
472 exp1 = Op Add [Const 3 , Op Mul [Const 2 ,Var "x"]]
473
474 msp :: MSP Ops
475 msp = [PUSH 3, PUSH 3, STORE, PUSH 3, LOAD, OUT,PUSH 99, OUTC,PUSH 99, OUTC,
476        PUSH 99, OUTC,PUSH 99, OUTC,PUSH 99, OUTC,PUSH 99, OUTC,PUSH 99, OUTC,
477        PUSH 99, OUTC,PUSH 99, OUTC,PUSH 99, OUTC,PUSH 99, OUTC ]
478
479 prog'' = Seq (Seq (Seq (Seq (Seq (Let "x" (Const 10))
480                                (Let "y" (Const 20)))
481                                (If (Op AND_ [Var "x",Const 34])
482                                    (Print (Var "x"))
483                                    (Print (Const 90))))
484                                (Let "y" (Op Add [Var "x", Var "y"])))
485                                (Ciclo (Let "i" (Const 0)
486                                       (Op LE_ [Var "i",Const 10])
487                                       (Print (Var "i"))
488                                       (Let "i" (Op Add [(Var "i"), (Const 1)]))))
489                                (If (Op AND_ [Var "x",Const 345])
490                                    (Print (Const 101010))
491                                    (Print (Var "y"))))
492                                (Print (Const 1)))
493                                (Print (Const 3)))
494
495 tt :: Prog Ops
496 tt = If (Op OR_ [Var "123",Var "345"]) (Print (Const 4)) (Print (Const 3))
497
498 prog :: Prog Ops
499 prog = Seq (Seq (Seq (Seq (Seq (Let "x" (Const 10))
500                                (Let "y" (Const 20)))
501                                (Let "z" (Op Add [Var "x", Var "y"])))
502                                (If (Op AND_ [Var "z",Const 345]) (Print (Const 1)) (Print (Const 3))))
503                                (Print (Var "z")))
504                                (Prints "mp1 a melhor cadeira de sempre! \n"))
505
506 mspprog :: MSP Ops
507 mspprog = [PUSH 2, PUSH 2, STORE, PUSH 3, OP Sim,
508            PUSH 4, PUSH 2, LOAD, OP Sub, OP Mul, OUT]

```
