Métodos de programação III

LMCC & LESI, Universidade do Minho

Ano lectivo 2005/2006

Ficha Teórico-Prática Nº8

Este texto está escrito em **literate Haskell**. Isto é, pode ser interpretado como um documento LaTeX ou como um puro programa na linguagem Haskell. Responda às perguntas sobre Haskell neste próprio ficheiro para assim produzir o programa e a sua documentação.

1 Autómatos Deterministas Reactivos em Haskell

Considere o seguinte tipo de dados algébrico que modela um autómato finito determinista reactivo em Haskell.

Solução

```
--
-- Módulo de Autómatos Finitos Deterministas Reactivos em Haskell
--
-- Métodos de Programação III
-- Universidade do Minho
-- 2005/2006
--

module MonadDfa where

import Monad
import Control.Monad.State

data Dfa m st sy  = Dfa  [sy]                -- Vocabulary
                         [st]                -- Finite set of states
                         st                  -- The start state
                         [st]                -- The set of final states
                         (st -> sy -> m st)  -- Monadic Transition Function
```

Tal como a função de aceitação de um autómato finito determinista se escreve como a função standard de recursividade sobre listas, nomeadamente a função —foldl—, a versão monádica/reactiva escreve-se agora na sua definição monádica, *i.e*, —foldM—.

Solução

```
dfawalk :: Monad m => (st -> sy -> m st) -> st -> [sy] -> m st
dfawalk delta st sys = foldM delta st sys
```

A função de aceitação para autómatos determinitas reactivo é também a versão monádica da função de aceitação para autómatos deterministas.

Solução

```
dfaaccept :: (Monad m, Eq st) => Dfa m st sy -> [sy] -> m Bool
dfaaccept (Dfa _ _ s z delta) sent =  do  st <- dfawalk delta s sent
                                          return (st `elem` z)
```

2 Autómatos Monádicos: Exemplos

2.1 *Considere de novo o autómato finito determinista* $\mathcal{A}_1$ *apresentado na ficha teórico-prática nº2, em que* $\mathcal{A}_1 = (\mathcal{V}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$*, com* $\mathcal{V} = \{a, b\}$*,* $\mathcal{Q} = \{1, 2, 3\}$*,* $\mathcal{S} = 1$*,* $\mathcal{Z} = \{3\}$ *e*
$\delta\ 1\ a = 2$
$\delta\ 1\ b = 1$
$\delta\ 2\ a = 1$
$\delta\ 2\ b = 3$
$\delta\ 3\ a = 1$
$\delta\ 3\ b = 2$

1. *Modele este autómato como um autómato monádico em Haskell*

2. *"baaab"* $\in \mathcal{L}(\mathcal{A}_2)$*?*

3. *"aab"* $\in \mathcal{L}(\mathcal{A}_2)$*?*

Solução

```
mad1 :: Monad m => Dfa m Int Char
mad1 = Dfa ['a','b'] [1,2,3] 1 [3] mad1_delta

mad1_delta :: Monad m => Int -> Char -> m Int
mad1_delta 1 'a' = return 2
mad1_delta 1 'b' = return 1
mad1_delta 2 'a' = return 1
mad1_delta 2 'b' = return 3
mad1_delta 3 'a' = return 1
mad1_delta 3 'b' = return 2

ex2_2 :: Maybe Bool
ex2_2 = dfaaccept mad1 "baaab"


ex2_3 :: Maybe Bool
ex2_3 = dfaaccept mad1 "aab"
```

2.2 *Considere a linguagem regular $\mathcal{L}$ modelada pela seguinte expressão regular p.*

$$p = a^{+}(b + c)^{*}dd$$

1. Modele $\mathcal{L}$ usando autómatos monádicos.
2. Prove usando a função `dfaaccept` que "$aabcbbccdd$" $\in \mathcal{L}$.

Solução

```
amp :: Monad m => Dfa m Int Char
amp = Dfa ['a','b','c','d'] [1,2,3,4,5,6] 1 [5] delta
   where delta 1 'a' = return 2
         delta 2 'a' = return 2
         delta 2 'b' = return 3
         delta 2 'c' = return 3
         delta 2 'd' = return 4
         delta 3 'b' = return 3
         delta 3 'c' = return 3
         delta 3 'd' = return 4
         delta 4 'd' = return 5
         delta _ _   = return 6


ex2_2_2 :: Maybe Bool
ex2_2_2 = dfaaccept amp "aabcbbccdd"


amp_v2 :: Dfa IO Int Char
amp_v2 = Dfa ['a','b','c','d'] [1,2,3,4,5,6] 1 [5] delta
   where delta 1 'a' = return 2
         delta 2 'a' = return 2
         delta 2 'b' = do  putStrLn "Apareceu um b́"; return 3
         delta 2 'c' = return 3
         delta 2 'd' = return 4
         delta 3 'b' = do  putStrLn "Apareceu um b́"; return 3
         delta 3 'c' = return 3
         delta 3 'd' = return 4
         delta 4 'd' = return 5
         delta _ _   = return 6
```

2.3 *Considere o problema do robot e da pepita apresentado na ficha sobre autómatos finitos não deterministas (ficha nº 3, exercício 4.1). O comportamento deste robot pode ser modelado pelo autómato finito determinista representado graficamente de seguida.*

Modele este robot como um autómato monádico em Haskell.

Solução

```
robot :: Monad m => Dfa m [Char] [Char]
robot = Dfa  ["esquerda","direita","largar","pegar"]
             ["C0 sem pepita","C0 com pepita","C1 sem pepita","C1 com pepita"]
             "C0 sem pepita"
             ["C0 sem pepita"]
             delta
  where
    delta "C0 sem pepita" "direita"   = return "C1 sem pepita"
    delta "C0 sem pepita" _           = return "C0 sem pepita"
    delta "C0 com pepita" "largar"    = return "C0 sem pepita"
    delta "C0 com pepita" "direita"   = return "C1 com pepita"
    delta "C0 com pepita" _           = return "C0 com pepita"
    delta "C1 sem pepita" "esquerda"  = return "C0 sem pepita"
    delta "C1 sem pepita" "pegar"     = return "C1 com pepita"
    delta "C1 sem pepita" _           = return "C1 sem pepita"
    delta "C1 com pepita" "esquerda"  = return "C0 com pepita"
    delta "C1 com pepita" "largar"    = return "C1 sem pepita"
    delta "C1 com pepita" _           = return "C1 com pepita"
```

2.4 *Defina uma sequência de movimentos do robot e utilize a função —dfaaccept— para aceitar (ou não) essa sequência de movimentos*

Solução

```
moves = ["direita","pegar","esquerda","largar"]
moves2 = ["esquerda","pegar","direita","pegar","esquerda","largar"]
moves3 = ["direita","direita","pegar","direita","largar","esquerda"
         ,"pegar","direita","pegar","esquerda","largar"]
moves4 = moves2 ++ moves ++ moves3
```

E uma função que aceita os movimentos do robot é:

Solução

```
acc :: Maybe Bool
acc = dfaaccept robot moves2
```

---

3 Autómato Reactivos com Operações de IO

3.1 *Considere de novo o problema do robot. Modele o robot como uma autómato reactivo em que a reacção efectuada será enviar para o* `stdout` *a mensagem* `"Apanhei uma pepina!"` *sempre que isso acontecer.*

Solução

```
robotIO :: Dfa IO [Char] [Char]
robotIO = Dfa  ["esquerda","direita","largar","pegar"]
               ["C0 sem pepita","C0 com pepita","C1 sem pepita","C1 com pepita"]
               "C0 sem pepita"
               ["C0 sem pepita"]
               delta
  where
    delta "C0 sem pepita" "direita"   = return "C1 sem pepita"
    delta "C0 sem pepita" _           = return "C0 sem pepita"
    delta "C0 com pepita" "largar"    = return "C0 sem pepita"
    delta "C0 com pepita" "direita"   = return "C1 com pepita"
    delta "C0 com pepita" _           = return "C0 com pepita"
    delta "C1 sem pepita" "esquerda"  = return "C0 sem pepita"
    delta "C1 sem pepita" "pegar"     = do  putStrLn "Apanhei uma pepita!"
                                            return "C1 com pepita"
    delta "C1 sem pepita" _           = return "C1 sem pepita"
    delta "C1 com pepita" "esquerda"  = return "C0 com pepita"
    delta "C1 com pepita" "largar"    = return "C1 sem pepita"
    delta "C1 com pepita" _           = return "C1 com pepita"
```

3.2 *Desenhe um Autómato Determinista Reactivo (com acções associadas às transições) que modele o sistema de controlo de um elevador (SCE) num prédio de 3 andares, cujo funcionamento se descreve a seguir de forma simplificada.*

O SCE recebe como entrada: os sinais de chamada a cada um dos 3 andares possíveis (originados dentro da cabine ou no seu exterior), os sinais de aproximação a cada um dos 3 andares, o sinal de paragem total e o sinal de portas livres. Como saída pode produzir as seguintes ordens: subir; descer; parar (quando se aproxima do andar pretendido); abrir portas (quando está parado); e fechar portas (quando estas estão livres). O SCE só recebe e processa um sinal de chamada quando está parado num andar de portas fechadas e, então, avança para o andar indicado (se for aquele onde está, apenas abre as portas) até cumrir totalmente a ordem (estacionar no andar pretendido e voltar a ficar com as portas fechadas); ignora qualquer outro sinal de chamada até ter completado a acção determinada pelo sinal recebido.

(pergunta do exame de 2003/2004)

Solução

3.3 *Desenhe um autómato reactivo que modele a máquina de pagamento do parque de estacionamento do CPIII. No seu estado inicial, a máquina começa por aceitar um bilhete. Depois aceita o respectivo pagamento, que pode ser um cartão multibanco, seguido do código, e seguido de um sinal da SIBS a indicar se o pagamento e válido ou não. O pagamento (assuma que é sempre da importância fixa de 0,40 Euros) também pode ser efectuado em moedas de 10, 20 ou 50 cêntimos. No caso da importância exceder os 0,40 Euros, terá que ser dado troco ao utente. A qualquer momento, o utente pode carregar na tecla 'cancelar', e todo o processo é interrompido, sendo devolvido o dinheiro eventualmente já introduzido. Se o pagamento for válido, então a máquina tem que perguntar se o utente quer ou não recibo. Se quiser recibo, imprime o recibo. Em todos os casos, no fim, ejecta o cartão de estacionamento.*

Ao modelar este autómato considere que a máquina não tem qualquer noção de estado. (pergunta do exame de 2002/2003)

Solução

```
cpiii :: Dfa IO Int String
cpiii = Dfa [ "cancelar"
            , "introduz bilhete"
            , "pagamento mb"
            , "pagamento moedas"
            , "codigo"
            , "cancelar"
            , "SIBS aprovado"
            , "SIBS recusado"
            , "recibo sim"
            , "recibo nao"
            , ".50 euros"
            , ".20 euros"
            , ".10 euros" ] [1,2,3,4,6,7,8,9] 1 [1] delta
   where delta 1 "introduz bilhete" = return 2
         delta 1 "cancelar"         = return 1

         delta 2 "pagamento mb"     = return 3
         delta 2 "pagamento moedas" = return 6
         delta 2 "cancelar"         = return 1

         delta 3 "codigo"   = return 4
         delta 3 "cancelar" = return 1

         delta 4 "SIBS aprovado" = do putStrLn "O pagamento foi efectuado!"
                                      return 5
         delta 4 "SIBS recusado" = do putStrLn "O código foi recusado"
                                      return 1
         delta 5 "recibo sim"    = do putStrLn "Total cobrado 0.40 cêntimos"
                                      putStrLn "Ejecta cartão"
                                      return 1
         delta 5 "recibo nao"    = do putStrLn "Ejecta cartão"
                                      return 1

         delta 6 ".50 euros" = do putStrLn "O pagamento foi efectuado!"
                                  putStrLn "Troco: .10 euros"
                                  return 5
         delta 6 ".20 euros" = return 8
         delta 6 ".10 euros" = return 7
         delta 6 "cancelar"  = return 1

         delta 7 ".50 euros" = do putStrLn "O pagamento foi efectuado!"
                                  putStrLn "Troco: .20 euros"
                                  return 5
         delta 7 ".20 euros" = return 9
         delta 7 ".10 euros" = return 8
         delta 7 "cancelar"  = do putStrLn "Troco: .10 euros"
                                  return 1

         delta 8 ".50 euros" = do putStrLn "O pagamento foi efectuado!"
```

4 Autómatos Reactivos com Estado

Para utilizarmos a noção de estado, temos de executar a função de aceitação com um valor inicial para o estado. Isto é feito utilizando a função —runState—.

Solução

```
runDfa :: Eq st => Dfa (State s) st sy -> s -> [sy] -> (Bool,s)
runDfa dfa initSt str = runState (dfaaccept dfa str) initSt
```

4.1 *Considere de novo o problema do robot. Modele o robot como um autómato reactivo em que a reacção efectuada será contar o número de pepitas que o robot apanhou.*

1. *Defina em Haskell qual o tipo de estado usado por este robot.*
2. *Defina em Haskell qual o valor inicial desse.*
3. *Define o tipo do autómato reactivo em Haskell.*
4. *Associe reacções ao autómato de modo a este ter o comportamento desejado.*
5. *Qual o resultado produzido por este autómato ao aceitar a frase definida no exercício 2.4*

Solução

```
-- alinea 1
type EstadoRobot    = State Integer

-- alinea 2
estadoInicialRobot  = 0

-- alinea 3
robotSt :: Dfa EstadoRobot [Char] [Char]


-- alinea 4
robotSt = Dfa  ["esquerda","direita","largar","pegar"]
               ["C0 sem pepita","C0 com pepita","C1 sem pepita","C1 com pepita"]
               "C0 sem pepita"
               ["C0 sem pepita"]
               delta
  where
    delta "C0 sem pepita" "direita"   = return "C1 sem pepita"
    delta "C0 sem pepita" _           = return "C0 sem pepita"
    delta "C0 com pepita" "largar"    = return "C0 sem pepita"
    delta "C0 com pepita" "direita"   = return "C1 com pepita"
    delta "C0 com pepita" _           = return "C0 com pepita"
    delta "C1 sem pepita" "esquerda"  = return "C0 sem pepita"
    delta "C1 sem pepita" "pegar"     = do modify ( c -> c+1)
                                           return "C1 com pepita"
    delta "C1 sem pepita" _           = return "C1 sem pepita"
    delta "C1 com pepita" "esquerda"  = return "C0 com pepita"
    delta "C1 com pepita" "largar"    = return "C1 sem pepita"
    delta "C1 com pepita" _           = return "C1 com pepita"
```

Para executarmos este autómato reactivo temos de dar um valor inicial ao estado.

Solução

```
runRobotSt :: [String] -> (Bool, Integer)
runRobotSt inp = runDfa robotSt estadoInicialRobot inp
```

4.2 *Considere de novo a linguagem $\mathcal{L}$ definida pela expressão regular p no exercício 2.2. Apresenta os seguinte autómatos reactivos:*

1. *Autómato reactivo que conta o número de símbolos b existentes numa frase.*
2. *Autómato reactivo que conta o número de símbolos b e c existentes numa frase.*

Solução

```
ex4_2_1 :: Dfa (State Integer) Int Char
ex4_2_1 = Dfa ['a','b','c','d'] [1,2,3,4,5,6] 1 [5] delta
   where delta 1 'a' = return 2
         delta 2 'a' = return 2
         delta 2 'b' = do  modify ( c -> c + 1); return 3
         delta 2 'c' = return 3
         delta 2 'd' = return 4
         delta 3 'b' = do  modify ( c -> c + 1); return 3
         delta 3 'c' = return 3
         delta 3 'd' = return 4
         delta 4 'd' = return 5
         delta _ _   = return 6


runEx4_2_1 = runDfa ex4_2_1 0


ex4_2_2 :: Dfa (State (Integer, Integer)) Int Char
ex4_2_2 = Dfa ['a','b','c','d'] [1,2,3,4,5,6] 1 [5] delta
   where delta 1 'a' = return 2
         delta 2 'a' = return 2
         delta 2 'b' = do  modify ( (b,c) -> (b + 1, c)); return 3
         delta 2 'c' = do  modify ( (b,c) -> (b, c + 1)); return 3
         delta 2 'd' = return 4
         delta 3 'b' = do  modify ( (b,c) -> (b + 1, c)); return 3
         delta 3 'c' = do  modify ( (b,c) -> (b, c + 1)); return 3
         delta 3 'd' = return 4
         delta 4 'd' = return 5
         delta _ _   = return 6

runEx4_2_2 = runDfa ex4_2_2 (0,0)
```

4.3 *A configuração de uma placa gráfica de um* Pc *obedece ao seguinte protocolo: a comunicação estabelece-se enviando um código inicial, constituído pelo padrão de bits* 000*. Posteriormente, são enviados valores em binário, de comprimento de 3 bits, para configurar vários parâmetros da placa. Esses valores são separados por uma sequência especial de bits:* 001*. Para indicar o fim da comunicação envia-se a sequência de bits* 111*.*

Este protocolo pode ser formalmente definido pela seguinte expressão regular:

$$000((0|1)(0|1)(0|1)(001(0|1)(0|1)(0|1))^*)111$$

Aplicando as técnicas estudadas nas fichas anteriores obtém-se o seguinte autómato finito determinista.

Considere que se pretende desenvolver um autómato reactivo que modela este protocolo e como reacção acumula numa lista os valores (em decimal) enviados. Considere ainda a função `converte` *que converte uma sequência de bits na sua representação decimal. Responda às alíneas seguintes*

1. *Indique qual o tipo de reacção do autómato (indicando o tipo do estado usado pelo autómato).*
2. *Defina o tipo do autómato finito determinista reactivo*
3. *Associe reacções ao autómato de modo a este ter o comportamento desejado.*

Solução

```
converte :: [Char] -> Int
converte []        = 0
converte ('0':xs)  = converte xs
converte ('1':xs)  = expo 2 (length xs) + converte xs

expo v e  | e > 0     = v * (expo v (e-1))
          | otherwise = 1
```

Para resolver este exercício vamos usar duas "variáveis globais": uma para acumular os bits que formam os valores enviados (variável do tipo —[Char]— e uma outra para acumular os vários números inteiros enviados —[Int]—.

Assim, o estado pode ser definido como uma par de listas e o seu valor inicial serão as duas listas vazias.

Solução

```
type ProtocolSt          =  State ([Char],[Int])

estado_inicial_protocol  =  ([],[])
```

O tipo do autómato é

Solução

```
pr :: Dfa ProtocolSt Integer Char
```

e define-se facilmente da seguinte forma:

Solução

```
pr = Dfa ['1','0'] [1,2,3,4,5,6,7,8,9,10,11,12,13] 1 [12] delta
  where
    delta 1   '0' = return 2
    delta 2   '0' = return 3
    delta 3   '0' = return 4
    delta 4   '0' = do  accum '0' ; return 5
    delta 4   '1' = do  accum '1' ; return 5
    delta 5   '0' = do  accum '0' ; return 6
    delta 5   '1' = do  accum '1' ; return 6
    delta 6   '0' = do  accum '0' ; accumList ; return 7
    delta 6   '1' = do  accum '1' ; accumList ; return 7
    delta 7   '0' = return 8
    delta 7   '1' = return 9
    delta 8   '0' = return 10
    delta 9   '1' = return 11
    delta 10  '1' = do  init ; return 4
    delta 11  '1' = do  init ; return 12
    delta _   _   = return 13

    accum x     = modify ( (b,v) -> (b++[x],v))   -- Acumula bits
    init        = modify ( (b,v) -> ("",v))
    accumList   = modify ( (b,v) -> (b,v ++ [converte b])) -- Acumula ints
```

A função que executa o autómato é:

Solução

```
runProtocolo :: [Char] -> (Bool,([Char],[Int]))
runProtocolo str = runDfa pr estado_inicial_protocol str
```

5 Autómatos Reactivos com Estado e Operações de IO

5.1 *Considere de novo o problema do robot. Modele o robot como um autómato reactivo em que a reacção efectuada será enviar para o* `stdout` *a mensagem* `"Larguei uma pepina!"` *sempre que isso acontecer. O Robot terá ainda de contrar o número de pepitas que apanhou.*

Solução

```
robotStIO :: Dfa (StateT Int IO) [Char] [Char]
robotStIO = Dfa  ["esquerda","direita","largar","pegar"]
                 ["C0 sem pepita","C0 com pepita","C1 sem pepita","C1 com pepita"]
                 "C0 sem pepita"
                 ["C0 sem pepita"]
                 delta
  where
    delta "C0 sem pepita" "direita"   = return "C1 sem pepita"
    delta "C0 sem pepita" _           = return "C0 sem pepita"
    delta "C0 com pepita" "largar"    = do  lift (putStrLn "Larguei uma pepita!")
                                            return "C0 sem pepita"
    delta "C0 com pepita" "direita"   = return "C1 com pepita"
    delta "C0 com pepita" _           = return "C0 com pepita"
    delta "C1 sem pepita" "esquerda"  = return "C0 sem pepita"
    delta "C1 sem pepita" "pegar"     = do  modify (+1)
                                            lift (putStrLn "Apanhei uma pepita!")
                                            return "C1 com pepita"
    delta "C1 sem pepita" _           = return "C1 sem pepita"
    delta "C1 com pepita" "esquerda"  = return "C0 com pepita"
    delta "C1 com pepita" "largar"    = do  lift (putStrLn "Larguei uma pepita!")
                                            return "C1 sem pepita"
    delta "C1 com pepita" _           = return "C1 com pepita"
```

A função de aceitação é

Solução

```
runRobotStIO inp = do  x <- evalStateT (dfaaccept robotStIO inp) 0
                       putStrLn (show x)
```

Solução

```
execRobotStIO inp = do  x <- execStateT (dfaaccept robotStIO inp) 0
                        putStrLn ("Apanhou " ++ (show x) ++ " pepitas")
```

5.2 *Considere de novo a máquibna de pagamentos do exercício 3.3. Considere agora que esta máquina armazena no seu estado a quantia que o utilizador introduz para efectuar o pagamento do bilhete.*

Responda às alíneas seguintes

1. *Indique qual o tipo de reacção do autómato (indicando o tipo do estado usado pelo autómato).*

2. Defina o tipo do autómato finito determinista reactivo

3. Associe reacções ao autómato de modo a este ter o comportamento desejado.

Solução

```
pe :: Dfa (StateT Int IO) Int String
pe = Dfa  ["bilhete","cartaoMB","codigo","SIBS_sim","SIBS_nao"
          ,"10","20","50","cancelar","recibo_sim","recibo_nao"]
          [1,2,3,4,5,6,7,8,9]
          1
          [6]
          d
  where  d 1 "bilhete"     = do  lift (putStrLn "Total a pagar: 40")
                                 return 2
         d 2 "cartaoMB"    = return 3
         d 3 "codigo"      = return 4
         d 4 "SIBS_sim"    = do  lift (putStrLn "Pagamento efectuado.")
                                 lift (putStrLn "Deseja recibo?")
                                 return 5
         d 4 "SIBS_nao"    = do  lift (putStrLn "Codigo nao aceite!")
                                 lift (putStrLn "Retire o bilhete.")
                                 return 6
         d 5 "recibo_sim"  = do  lift (putStrLn "Retire o recibo.")
                                 lift (putStrLn "Retire o bilhete.")
                                 return 6
         d 5 "recibo_nao"  = do  lift (putStrLn "Retire o bilhete.")
                                 return 6
         d 2 "10"          = do  x <- get
                                 put (x + 10)
                                 return (if x + 10 >= 40 then 8 else 7)
         d 2 "20"          = do  x <- get
                                 put (x + 20)
                                 return (if x + 20 >= 40 then 8 else 7)
         d 2 "50"          = do  x <- get
                                 put (x + 50)
                                 return (if x + 50 >= 40 then 8 else 7)
         d 7 "10"          = do  x <- get
                                 put (x + 10)
                                 return (if x + 10 >= 40 then 8 else 7)
         d 7 "20"          = do  x <- get
                                 put (x + 20)
                                 return (if x + 20 >= 40 then 8 else 7)
         d 7 "50"          = do  x <- get
                                 put (x + 50)
                                 return (if x + 50 >= 40 then 8 else 7)
         d 8 "recibo_sim"  = do  x <- get
                                 lift (putStrLn ("Retire o troco de " ++ show(x - 40)))
                                 lift (putStrLn "Retire o recibo.")
                                 lift (putStrLn "Retire o bilhete.")
                                 return 6
         d 8 "recibo_nao"  = do  x <- get
                                 lift (putStrLn ("Retire o troco de " ++ show (x-40)))
                                 lift (putStrLn "Retire o bilhete.")
                                 return 6
         d _ "cancelar"    = do  lift (putStrLn "Operacao cancelada!")
                                 x <- get
```

Solução

```
pe_inp1 = ["bilhete","cartaoMB","codigo","SIBS_sim","recibo_sim"]
pe_inp2 = ["bilhete","cartaoMB","codigo","SIBS_nao"]
pe_inp3 = ["bilhete","cartaoMB","codigo","cancelar"]
pe_inp4 = ["bilhete","10","10","10","10","recibo_sim"]
pe_inp5 = ["bilhete","20","10","20","recibo_sim"]
pe_inp6 = ["bilhete","10","20","50","recibo_nao"]
pe_inp7 = ["bilhete","10","20","50","cancelar"]


evalPE inp = do  x <- evalStateT (dfaaccept pe inp) 0
                 return ()
```