# Métodos de programação III

LMCC & LESI, Universidade do Minho

Ano lectivo 2004/2005

Ficha Teórico-Prática Nº7

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.

```
--
-- Módulo de Autómatos Finitos Deterministas Reactivos em Haskell
--
-- Métodos de Programação III
-- Universidade do Minho
-- 2004/2005
--
module MonadDfa where
import Monad
import MonadState
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*.

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

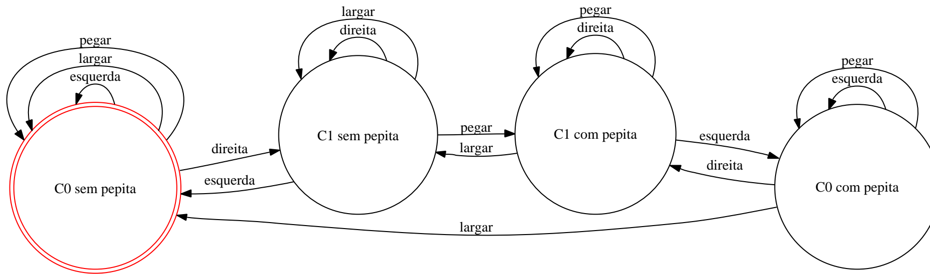
$dfawalk\ delta\ st\ sys = foldM\ delta\ st\ sys$

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

$$
\begin{array}{l}
dfaaccept :: (Monad\ m, Eq\ st) \Rightarrow Dfa\ m\ st\ sy \rightarrow [sy] \rightarrow m\ Bool \\
dfaaccept\ (Dfa\ _\ _\ s\ z\ delta)\ sent = \mathbf{do}\ st \leftarrow dfawalk\ delta\ s\ sent \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ (st \in z)
\end{array}
$$

# 2 Autómatos Reactivos: Exemplos

**2.1** *Considere o problema do robot e da petita apresentado na ficha sobre autómatos finitos não deterministas. O comportamento deste robot pode ser modelado pelo seguinte autómato finito determinista:*




*Modele este robot como uma autómato reactivo em que a reacção efectuada é apenas transitar para o estado destino.*

```
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"
```

**2.2** *Defina uma sequência de movimentos deo robot e utilize a função dfaaccept para aceitar (ou não) essa sequeência de movimentos*

```
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:

```
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.*

```
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)*

# 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. Istoi é feito utilizando a função *runState*.

```
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.*

```
robotSt :: Dfa (State Int) [Char] [Char]
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 auómato reactivo temos de dar um valor inicial ao estado. Neste caso iniciamos o acumulador de pepitas com o valor 0.

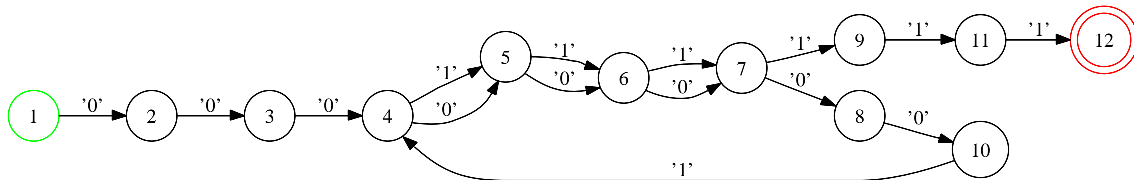
```
runRobotSt :: [String] → (Bool, Int)
runRobotSt inp = runDfa robotSt 0 inp
```

**4.2** *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:*

$$000((01)\ (01)(01)\ (001\ (01)(01)\ (01))^*)111$$

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




*Construa um autómato reactivo que modele este protocolo e como reacção acumule numa lista os valores (em decimal) enviados.*

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

Assim, o estado pode ser definido como uma par:

```
type MyState = ([Char], [Int])
```

O tipo do autómato é

```
pr :: Dfa (State MyState) Integer Char
```

e define-se facilmente da seguinte forma:

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

A função que executa o autómato é:

```
runProtocolo :: [Char] → (Bool, MyState)
runProtocolo str = runDfa pr ("", []) str

  -- funções auxiliares
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
```

# 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.*

```
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

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

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

**5.2** *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.*

*(pergunta do exame de 2002/2003)*

```
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
                      lift (putStrLn ("Retire o dinheiro: " ++ show x))
                      return 6
d _ _            = return 9

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 ()
```