Métodos de programação III

LMCC & LESI, Universidade do Minho

Ano lectivo 2004/2005

Ficha Teórico-Prática Nº2

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 Expressões Regulares

1.1 *Sejam α, β e γ expressões regulares, então podemos afirmar que:*

1. $(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma)$
2. $\alpha + \emptyset = \emptyset + \alpha = \alpha$
3. $\alpha + \beta = \beta + \alpha$
4. $\alpha + \alpha = \alpha$
5. $(\alpha\beta)\gamma = \alpha(\beta\gamma)$
6. $\alpha\epsilon = \epsilon\alpha = \alpha$
7. $\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$
8. $(\beta + \gamma)\alpha = \beta\alpha + \gamma\alpha$
9. $\alpha^+ = \alpha\alpha^* = \alpha^*\alpha$
10. $\alpha^* = \epsilon + \alpha^+$
11. $(\alpha + \epsilon)^+ = (\alpha + \epsilon)^* = \alpha^*$

Prove as seguintes igualdades:

1. $b^+ + (b^+ + \epsilon)b = b^+$

2. $a^*(b+cd)b^* = a^*b^+ + a^*cd + a^*cdb^+$

3. $a + a^+ = a^+$

Simplifique as seguintes expressões regulares:

1. $a(a^*b + c)a + a^+ba$

2. $l + l(l + d)^*$

1.2 *Prove que as expressões regulares* $p = (ab+\epsilon)^+ab+a(a^*+bc)c^*$ *e* $q = abc^+ + a^+c^* + (ab)^*$ *não são equivalentes: Dê um exemplo de uma frase* α *da linguagem definida pela expressão regular* q*, mas que não é da linguagem definida por* p*, i.e.,* $\alpha \in \mathcal{L}(q)$ *e* $\alpha \notin \mathcal{L}(p)$*.*

2 Expressões Regulares em Haskell

Considere o seguinte código, escrito na ficha anterior, que define o tipo de dados `RegExp` para modelar expressões regulares em Haskell e define expressão regulares que definem as linguagens dos digitos e dos inteiros.

Solução

```
--
-- Módulo de Expressões Regulares em Haskell
--
-- Métodos de Programação III
-- Universidade do Minho
-- 2004/2005
--

module RegExp where

data RegExp = Epsilon
            | Literal   Char
            | Or        RegExp RegExp
            | Then      RegExp RegExp
            | Star      RegExp
            | OneOrMore RegExp
            | Optional  RegExp

digitos = (Literal '0') `Or` (Literal '1') `Or` (Literal '2') `Or`
          (Literal '3') `Or` (Literal '4') `Or` (Literal '5') `Or`
          (Literal '6') `Or` (Literal '7') `Or` (Literal '8') `Or`
          (Literal '9')

int'    = (Optional ((Literal '-') `Or` (Literal '+')))
          `Then` (OneOrMore digitos)
```

2.1 *Complete a definição da função* ***matches****.*

Solução

```
matches ::
matches Epsilon inp        =
matches (Literal l) inp    =
matches (Or re1 re2) inp   =
matches (Then re1 re2) inp =
matches (Star re) inp      = matches Epsilon inp ||
                               or [ matches re s1 && matches (Star re) s2
                                  | (s1,s2) <- frontSplits inp ]

splits :: [a] -> [ ([a],[a]) ]
splits s = [ splitAt n s | n <- [ 0 .. length s ] ]

frontSplits :: [a] -> [ ([a],[a]) ]
frontSplits s = [ splitAt n s | n <- [ 1 .. length s ] ]
```

2.2 *Escreva o código Haskell para definir as seguintes funções:*

- ***isInt*** *é uma função que dada uma string* α *"diz"se* α *é um inteiro ou não. Isto é, tem tipo* ***isInt :: String -> Bool****.*
- *Considere as expressões regulares p e q do exercício 1.2. Re-escreva essas expressões regulares em Haskell e defina a função* ***xxx*** *que tem como argumento uma frase e dá verdade se a frase pertence a ambas as linguagens definidas pelas expressões regulares e falso caso contrário. Use a frase* α *definida no exercício 1.2 para provar em Haskell que p e q não são equivalentes.*

Solução

```
-- isInt :: String -> Bool
```

2.3 *O tipo de dados* ***RegExp*** *apresentado já contém os constructores para modelar os operadores opcional e uma ou mais repetições usualmente usados em expressões regulares. Porém, para re-utilizar a função* ***matches*** *é necessãrio primeiro converter uma expressão regular extendida para a sua forma canónica (isto é, definida pelos cinco constructores iniciais). Escreva uma função* ***extREtoRE*** *para esse efeito.*

Solução

```
-- matches' = matches . extREtoRE
```

2.4 *Considere a função* **`extREtoRE`** *e a função* **`showRE`** *(definida na ficha-teórica nº1). Ambas as funções apresentam o mesmo padrão de recursividades: a recursividade segue a estrutura do tipo de dados* **`RegExp`**. *Recorde que na cadeira de Métodos de Programação I foram apresentados os catamorfismos para modelar estes padrões de recursividade. Escreva o código Haskell para:*

1. *Derive o catamorfismo para o tipo de dados indutivo* **`RegExp`**
2. *Re-escrever as funções* **`extREtoRE`** *e* **`showRE`** *segundo catamorfismos.*

Solução
