Expressões Regulares

Sintaxe Abstrata

```
data RE = REEmpty
        | REEpsilon
        | REChar Char          -- a
        | REAny                -- .
        | RESeq RE RE          -- e1 e2
        | REOr RE RE           -- e1 | e2
        | REKleene RE          -- e*
```

Exemplo

```
-- [+-]?[01]+  ≡  (ε|(+|-))((0|1)(0|1)*)
exp1 =
  let signal = REOr (REChar '+') (REChar '-')
      zeroone = REOr (REChar '0') (REChar '1') in
    RESeq (REOr REEpsilon signal)
          (RESeq zeroone (REKleene zeroone))
```

Semântica de REs

- O que uma expressão regular representa?
  - Uma *linguagem* = um conjunto de strings
- Como podemos representar conjuntos em uma linguagem funcional?
  - Função indicadora (função característica):

```
String -> Bool
```

Função Semântica de REs

– visão abstrata

```
eval : RE -> (String -> Bool)
```

– interpretador

```
eval : RE -> String -> Bool
```

Casos Simples

```
eval REEmpty _ = False

eval REEpsilon s = null s

eval (REChar c) s = (s == [c])

eval REAny s = case s of
                 [_]    -> True
                 _      -> False

eval (REOr e e') s = eval e s || eval e' s
```

Concatenação

- $s \in (E1\ E2)$ se e somente se

  $\exists\ x, y\ .\ (xy = s) \wedge (x \in E1) \wedge (y \in E2)$

- Equivalente, mas computável:
  - $\exists\ i \in [0..\text{length } s]\ .\ (\text{take } i\ s \in E1) \wedge (\text{drop } i\ s \in E2)$

```
eval (RESeq e e') s = exists [0..length s] p
  where p i = eval e (take i s) && eval e' (drop i s)
```

A Função `exists`

```
exists ::  [a] -> (a -> Bool)-> Bool
exists [] p = False
exists (x:xs) p = p x || exists xs p
```

Fecho de Kleene (repetição)

- Como podemos tratar e* ?
- Uma maneira usual de tratarmos repetições é “expandir um nível”:
  - e* ≡ (ε | e e*)
- Agora, não temos apenas indução estrutural (pois e* é definido em termos de e*).
  - Isso pode levar a loops infinitos!

```
eval «ε*» “a”          →
eval «ε | ε ε*» “a”    →
eval «ε ε*» “a”        →
eval «ε*» “a”          →
...
```

Fecho de Kleene

- Para evitar loop infinito, precisamos garantir que, a cada novo casamento com e*, a string esteja menor...
- ...ou seja, que o "e" inicial em "e e*" nunca case com a string vazia. Formalmente:
  - e* ≡ (ε | (e - ε) e*)

Fecho de Kleene

- Em Haskell:

```
-- e* ≡ (ε | (e - ε) e*)

eval es@(REKleene e) s = null s || exists [1..length s] p
  where p i = eval e (take i s) && eval es (drop i s)
```

Observações

- Em Haskell, ou qualquer outra linguagem, existem os limites de computabilidade que não temos na matemática.
  - e.g., quantificadores sobre conjuntos infinitos
- Essa implementação é particularmente ineficiente, mas mais próxima da definição matemática.
- Nosso uso de recursão não é mais apenas estrutural.
  - E, portanto, não é automaticamente “bem comportado”