UNIVERSIDADE BEIRA INTERIOR

Teoria da Computação
Aula - Expressões regulares
Álgebra e Algoritmia

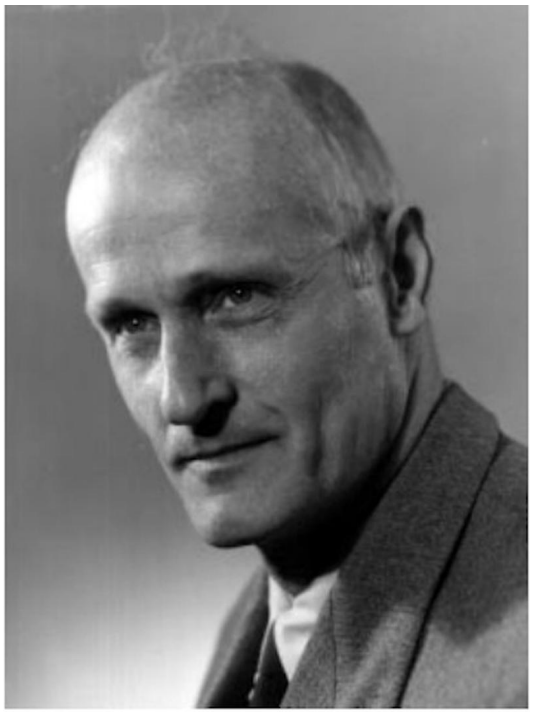
Simão Melo de Sousa

(re)lease
REliable And SEcure Computation Group

Introdução

Stephen Cole Kleene, 1909-1994

fundador, em conjunto com Turing, Church ou ainda com Post da **Teoria da Computação**



autor da teoria das funções recursivas, modelo alternativo às máquinas de Turing e ao λ-cálculo

inventor do conceito de expressões regulares e de autómato

em consequência do seu estudo da resposta animal a estímulos externos [1], com base no trabalho seminal de McCulloch e Pitts [2] (os inventores das redes neuronais artificiais)

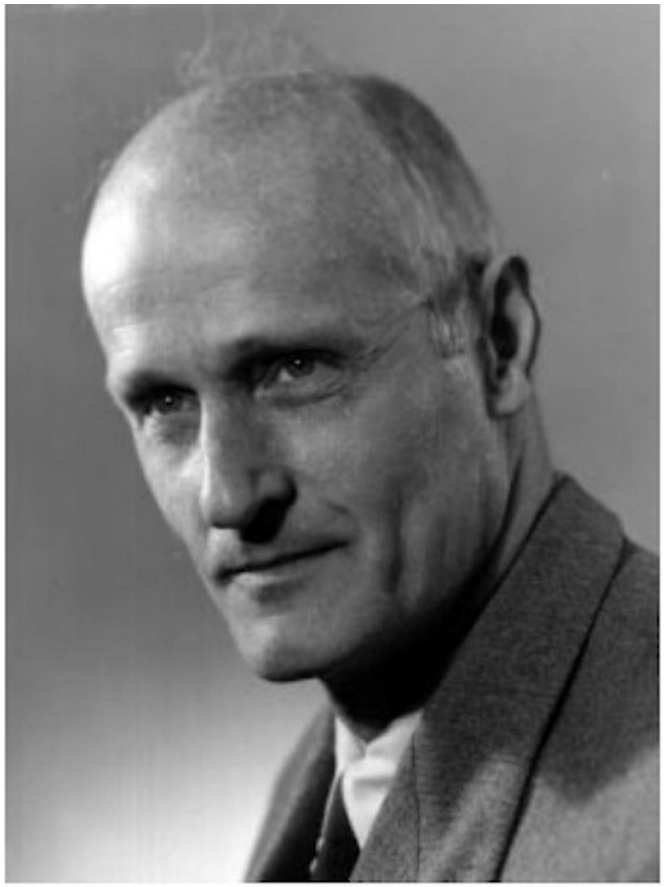
---

[1] Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In C. Shannon & J. McCarthy (ed.), Automata Studies (pp. 3–41) . Princeton University Press .

[2] Warren McCulloch and Walter Pitts "A logical calculus of the ideas immanent in nervous activity" Bull. Math. Biophysics 5 (1943), pp. 115-133

Stephen Cole Kleene, 1909-1994

fundador, em conjunto com Turing, Church ou ainda com Post da **Teoria da Computação**



conjectura que existe uma **teoria equacional** (conjunto de axiomas) **completa** que permita derivar todas as equações válidas que envolvem expressões regulares

essa teoria equacional forma as **Álgebras de Kleene**

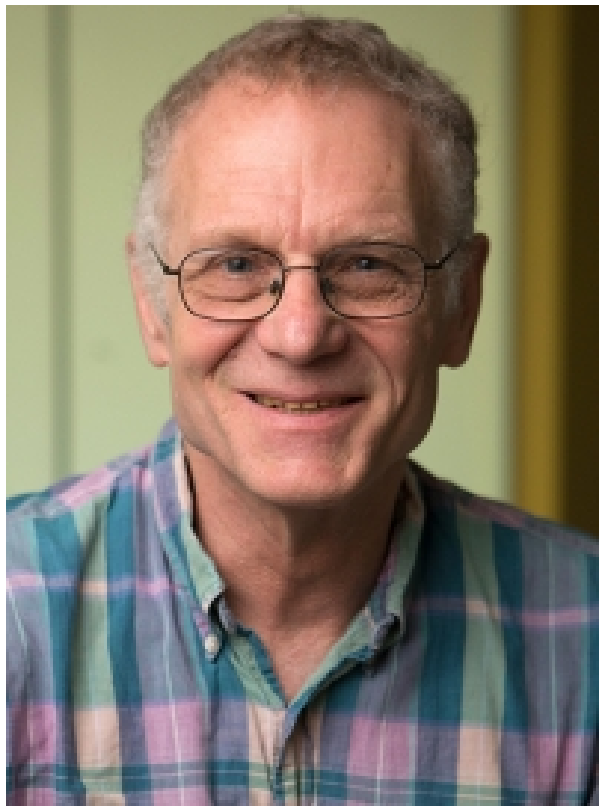
as álgebras de Kleene que vamos aqui estudar foram batizadas desta forma em honra da sua contribuição seminal

Dexter Kozen, 1951 -

matemático e informático da universidade de Cornell (USA)

em 1991, propôs uma teoria equacional na forma de um conjunto finito e completo de axiomas para as álgebras de Kleene

(... e respectiva prova de completude)



resolveu assim a conjectura formulada por Kleene e definiu *na pedra* o que elas são[3]

[3] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. Infor. and Comput., 110(2):366-390, 1994. Previoulsy published in 1991 at LICS'91

Ken Thompson introduziu no sistema UNIX uma primitiva de procura de texto particularmente interessante, rica e eficiente baseada nas expressões regulares que vamos aqui estudar:

Ken Thompson, "Regular Expression Search Algorithm" Comm. Assoc. Comp. Mach., Vol. 11, 6, pp. 419422, 1968.

uma nota de Russ Cox (link), retirada dos apontamentos de Thierry Coquand (fonte: ver bibliografia), particularmente interessante:

> *"Historically, regular expressions are one of computer science's shining examples of how using good theory leads to good programs.*
>
> *Today, regular expressions have also become a shining example of how ignoring good theory leads to bad programs. The regular expression implementations used by today's popular tools are significantly slower than the ones used in many of those thirty-year-old Unix tools."*

Expressões regulares e álgebras de Kleene

expressões regulares: (lembrete) uma **notação finita** para representar linguagens **infinitas** cujas palavras apresentam **padrões regulares**

álgebras de Kleene: formalização algébrica do conceito de **evento regular**

exemplo: sequências de caracteres (conjunto das palavras) expressas por expressões regulares

a saída 0100110110101110101001010 é gerada por qual destes dois programas?

```
while (Random.bool ()) do
  if (Random.bool ())
    then print_int 1
    else print_int 0
done; print_newline ()
```

```
while (Random.bool ()) do
  print_int 0
done;
while (Random.bool ()) do
  print_int 1;
  while (Random.bool ()) do
    print_int 0
  done
done; print_newline ()
```

como caracterizar as saídas de cada um destes dois programas? como se relacionam?

Estes programas são equivalentes?

```
let () =
  while (a && b) do
    p()
  done;
  while a do
    q();
    while (a && b) do
      p()
    done
  done
```

```
let () =
  while a do
    if b then p()
    else q()
  done
```

a algoritmia própria às álgebras de Kleene[4] permite responder a estas questões de forma completa e automática

[4] Álgebras de Kleene com testes, no caso deste exemplo

Definições de base

um **semi-grupo** é uma estrutura algébrica $(S, \cdot)$ onde

- S é um conjunto e
- $\cdot$ é uma **lei de composição interna** em S (i.e. uma aplicação binária de $S \times S \to S$) **associativa**, i.e.

$$\forall x, y, z \in S,\ x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

ou seja podemos escrever $x \cdot y \cdot z$ sem risco de ambiguidade

um **monoíde** é uma estrutura algébrica $(M, \cdot, \mathbb{1})$ tal que

- $(M, \cdot)$ é um **semi-grupo** e
- $\mathbb{1}$ é um elemento particular de M que é a **identidade** tanto **esquerda** como **direita** para $\cdot$, i.e.

$$\forall x \in M,\ \mathbb{1} \cdot x = x \cdot \mathbb{1} = x$$

por questões de conveniência escreveremos xyz no lugar de $x \cdot y \cdot z$ quando o contexto não introduz ambiguidades

- $(\mathbb{N}, +, 0)$
- $(\mathbb{N}, \times, 1)$
- $(R^{n\times n}, \times, I)$ onde $R^{n\times n}$ é o conjunto das matrizes de n por n de elementos de um anel [5] R, $\times$ é a multiplicação clássica de matrizes e I é a matriz identidade.

de notar que os dois primeiros exemplos são também comutativos

[5] $(A, +, \times, 0)$ é um anel se as duas leis internas são associativas, se $+$ é comutativa, se $\forall a \in A, \exists b \in A$ tal que $a + b = 0$, se 0 é elemento neutro de $+$ e se $\times$ é distributiva (à direita e à esqueda) em relação a $+$.

Semi-anel idempotente

um **semi-anel** $(S, +, \times, \mathbb{0}, \mathbb{1})$ é uma estrutura algébrica tal que

- $(S, +, \mathbb{0})$ é um monoíde comutativo
- $(S, \times, \mathbb{1})$ é um monoíde
- $\times$ é distributiva (à direita e à esquerda) em relação a $+$:
  $\forall x, y, z \in S,\ x \times (y + z) = x \times y + x \times z \quad \wedge \quad (x + y) \times z = x \times z + y \times z$
- $\mathbb{0}$ é elemento absorvente para $\times$ (i.e. para todo o x de S, $\mathbb{0} \times x = x \times \mathbb{0} = \mathbb{0}$)

convenções notacionais sobre $+$ e $\times$: usaremos as abreviaturas habituais, em particular $(x \times y) + (z \times (x + y)))$ será abreviado em $xy + z(x + y)$

um semi-anel é **idempotente** se $\forall x \in S,\ x + x = x$

agrupando todas as propriedades que a define, um **semi-anel** $(S, +, \times, \mathbb{0}, \mathbb{1})$ **idempotente** é uma estrutura algébrica tal que $\forall x, y, z \in S$:

$$
\begin{array}{lcl}
x + (y + z) & = & (x + y) + z \\
x(yz) & = & (xy)z \\
x(y + z) & = & xy + xz \\
(x + y)z & = & xz + yz \\
x + x & = & x
\end{array}
\qquad
\begin{array}{lcll}
x + y & = & y + x & \\
x + \mathbb{0} & = & x & \\
\mathbb{0}x & = & x\mathbb{0} & = 0 \\
\mathbb{1}x & = & x\mathbb{1} & = x
\end{array}
$$

um **anel** é um semi-anel no qual o monoíde aditivo forma um grupo, isto é, no qual se tem $\forall x \in S,\ \exists y,\ x + y = \mathbb{0}$ (para todo o elemento, existe um elemento inverso para a soma)

relembramos que uma ordem parcial R é uma relação binária sobre um conjunto (digamos S, ou seja $R \subseteq S \times S$) tal que R seja reflexiva, anti-simétrica e transitiva

todo o semi-anel idempotente tem uma ordem natural $\leq$ definida como

$$x \leq y \quad \triangleq \quad x + y = y$$

(**exercício**: provar que $\leq$ é de facto uma ordem parcial)

em qualquer semi-anel idempotente $(S, +, \times, \mathbb{0}, \mathbb{1})$, as operações $+$ **e** $\times$ **são monótonas em relação a** $\leq$
ou seja, $\forall x, y, z \in S$,

$$\begin{aligned} x \leq y &\rightarrow x + z \leq y + z \\ x \leq y &\rightarrow z + x \leq z + y \\ x \leq y &\rightarrow xz \leq yz \\ x \leq y &\rightarrow zx \leq zy \end{aligned}$$

num conjunto parcialmente ordenado $(C, \leq)$ e, considerando um subconjunto S de C $(S \subseteq C)$, o **supremo** ou **limite superior mínimo**[6] de S é um elemento s de C tal que

$$\forall x \in S, x \leq s \quad \wedge \quad \forall s' \in C, \forall x \in S, x \leq s' \implies s \leq s'$$

é, informalmente, o menor elemento de todos os elementos que são maiores do que qualquer elemento de S (i.e. **o** menor dos majorantes)

de notar que este pode não existir

por questões de conveniência, designaremos o supremo por ***lub*** (iniciais de *least upper bound*)

[6] em inglês: ***supremum*** ou ***least upper bound***.

num semi-anel idempotente $(S, +, \times, \mathbb{0}, \mathbb{1})$ o operador $+$ designa o *lub* de qualquer par de elementos em relação a ordem $\leq$,

isto é, o elemento $x + y$ é o *lub* do par (x, y)

(**exercício**: demonstre tal propriedade)

num semi-anel idempotente $(S, +, \times, \mathbb{0}, \mathbb{1})$, $\mathbb{0}$ é o menor elemento de S relativamente à ordem natural $\leq$, ou seja,

$$\forall x \in S, \mathbb{0} \leq x$$

(**exercício**: demonstrar!)

informalmente o operador $\times$ captura a noção de composição sequencial e o operador $+$ captura a noção de escolha

nesta linha, vamos agora juntar um operador suplementar (unário, notação posfixa), na verdade essencial e central, o operador $*$ para a **iteração**, **repetição**

é designado de **operador estrela**, **fecho de Kleene** ou ainda o **fecho reflexivo transitivo**

uma **Álgebra de Kleene** $(K, +, \times, *, \mathbb{0}, \mathbb{1})$ é uma estrutura algébrica tal que $(K, +, \times, \mathbb{0}, \mathbb{1})$ seja um semi-anel idempotente que, considerando a ordem natural $\leq$, verifica as propriedades suplementares seguintes

$$\begin{array}{lcl} \mathbb{1} + xx^* & \leq & x^* \\ \mathbb{1} + x^*x & \leq & x^* \\ b + ax & \leq & x \longrightarrow a^*b \leq x \\ b + xa & \leq & x \longrightarrow ba^* \leq x \end{array}$$

(de notar que estamos a assumir as precedências $* > \times > +$ para simplificar a escrita de expressões)

resumindo, uma **Álgebra de Kleene** $(K, +, \times, *, \mathbb{0}, \mathbb{1})$ verifica

$$
\begin{array}{lcl}
x + (y + z) & = & (x + y) + z \\
x(yz) & = & (xy)z \\
x(y + z) & = & xy + xz \\
(x + y)z & = & xz + yz \\
x + x & = & x
\end{array}
\qquad
\begin{array}{lcl}
x + y & = & y + x \\
x + \mathbb{0} & = & x \\
\mathbb{0}x & = & x\mathbb{0} \; = \; \mathbb{0} \\
\mathbb{1}x & = & x\mathbb{1} \; = \; x
\end{array}
$$

$$
\begin{array}{lcl}
\mathbb{1} + xx^* & \leq & x^* \\
\mathbb{1} + x^*x & \leq & x^* \\
b + ax \leq x & \longrightarrow & a^*b \leq x \\
b + xa \leq x & \longrightarrow & ba^* \leq x
\end{array}
$$

(todas estas equações e inequações são quantificadas universalmente sobre o conjunto dos termos K e $x \leq y \triangleq x + y = y$)

o fecho de Kleene é uma peça central nas álgebras de Kleene

os seus axiomas permitam calcular soluções para equações lineares referentes a elementos da álgebra considerada

por exemplo[7] (qualquer que sejam a e b de K) a menor solução à inequação

$$b + aX \leq X$$

é

$$X = a^* b$$

$a^* b$ é solução:

$$\begin{array}{rcll} \mathbb{1} + aa^* \leq a^* & \rightarrow & (\mathbb{1} + aa^*)b \leq a^* b & \text{(axioma } \mathbb{1} + xx^* \leq x^*\text{)} \\ & \rightarrow & b + a(a^* b) \leq a^* b & \end{array}$$

$a^* b$ é a menor solução:

directamente pelo axioma $b + ax \leq x \longrightarrow a^* b \leq x$

[7] Reencontraremos (uma instanciação d) este resultado mais adiante sobre o nome de **Lema de Arden**

$$\mathbb{1} + xx^* \leq x^* \ (*) \quad \text{e} \quad b + ax \leq x \longrightarrow a^* b \leq x \ (**)$$

podemos promover, à custa da axiomática das KAs, $\mathbb{1} + xx^* \leq x^*$ para

$$\mathbb{1} + xx^* = x^*$$

para tal, basta demonstrar que $x^* \leq \mathbb{1} + xx^*$ (por anti-simetria de $\leq$ obtemos a igualdade)

- consideremos assim (**) com $a = x$, $b = \mathbb{1}$, $x = \mathbb{1} + xx^*$, temos

$$\mathbb{1} + x(\mathbb{1} + xx^*) \leq (\mathbb{1} + xx^*) \longrightarrow x^*\mathbb{1} \leq \mathbb{1} + xx^*$$

- assim, desde que $\mathbb{1} + x(\mathbb{1} + xx^*) \leq (\mathbb{1} + xx^*)$, temos $x^*\mathbb{1} \leq \mathbb{1} + xx^*$

$$\mathbb{1} + xx^* \leq x^* \text{ (*)} \quad \text{e} \quad b + ax \leq x \longrightarrow a^*b \leq x \text{ (**)}$$

- ora $x^*\mathbb{1} = x^*$ (pelo axioma $\forall x.x = x\mathbb{1}$, em particular quando x é x^*)
- ou seja, desde que $\mathbb{1} + x(\mathbb{1} + xx^*) \leq (\mathbb{1} + xx^*)$, temos $x^* \leq \mathbb{1} + xx^*$
- por monotonia de $\leq$ em relação a $+$ e a $\times$, temos $\mathbb{1} + x(\mathbb{1} + xx^*) \leq (\mathbb{1} + xx^*)$ desde que $\mathbb{1} + xx^* \leq x^*$
- ora, isto é nos garantido pelo axioma (*).

QED. $\mathbb{1} + xx^* = x^*$

exercício: demonstre que $\mathbb{1} + x^* x \leq x^*$

$\mathbb{1} \leq a^*$

$a \leq a^*$

$a^* a^* = a^*$

$a^{**} = a^*$

$ac = cb \longrightarrow a^* c = cb^*$

$(ab)^* a = a(ba)^*$ a.k.a. regra shifting

$(a + b)^* = a^*(ba^*)^*$ a.k.a. regra denesting

$a \leq b \longrightarrow a^* \leq b^*$ $*$ é monótono

demonstração de $(a+b)^* \leq a^*(ba^*)^*$

primeiro, observemos que

$$\begin{array}{lcl} \mathbb{1} & \leq & a^*(ba^*)^* \\ aa^*(ba^*)^* & \leq & a^*(ba^*)^* \\ ba^*(ba^*)^* & \leq & (ba^*)^* \\ & \leq & a^*(ba^*)^* \end{array}$$

logo

$$\begin{array}{lcl} \mathbb{1} + (a+b)a^*(ba^*)^* & \leq & \mathbb{1} + aa^*(ba^*)^* + ba^*(ba^*)^* \\ & \leq & a^*(ba^*)^* \end{array}$$

do axioma $b + ax \leq x \rightarrow a^*b \leq x$ segue-se que $(a+b)^* \leq a^*(ba^*)^*$ **QED**

demonstração de $a^*(ba^*)^* \leq (a+b)^*$

usemos a monotonia dos diferentes operadores das álgebras de Kleene

$$\begin{aligned} a^*(ba^*)^* &\leq (a+b)^*((a+b)(a+b)^*)^* \\ &\leq (a+b)^*((a+b)^*)^* \\ &\leq (a+b)^* \end{aligned}$$

QED. $(a+b)^* = a^*(ba^*)^*$

pragmaticamente para derivar outras equivalências de forma equacional, podemos usar as propriedades de base seguintes que nos são dadas pela axiomática das álgebras de Kleene mais algumas propriedades de base imediatas: uma **Álgebra de Kleene** $(K, +, \times, *, \mathbb{0}, \mathbb{1})$ verifica

$$
\begin{array}{lcl}
x + (y + z) & = & (x + y) + z \\
x(yz) & = & (xy)z \\
x(y + z) & = & xy + xz \\
(x + y)z & = & xz + yz \\
x + x & = & x
\end{array}
\qquad
\begin{array}{lcll}
x + y & = & y + x & \\
x + \mathbb{0} & = & x & \\
\mathbb{0}x & = & x\mathbb{0} & = \mathbb{0} \\
\mathbb{1}x & = & x\mathbb{1} & = x \\
\mathbb{1} + xx^* & = & x^* & \\
\mathbb{1} + x^*x & = & x^* &
\end{array}
$$

$$
\begin{array}{lcl@{\qquad}lcl}
b + ax & \leq & x & \longrightarrow & a^*b & \leq & x \\
b + xa & \leq & x & \longrightarrow & ba^* & \leq & x
\end{array}
$$

em conjunção com o Lema de Arden que estabelece que uma solução menor a $X = b + aX$ é $X = a^*b$

outras identidades notáveis

as seguintes expressões são equivalentes

- $(a+b)^*$
- $(a^*+b)^*$
- $a^*(a+b)^*$
- $(a+ba^*)^*$
- $(a^*b^*)^*$
- $(a^*b)^*a^*$
- $a^*(ba^*)^*$

temos igualmente

- $\mathbb{0}^* = \mathbb{1}$
- $(\mathbb{1}+a^*) = a^*$
- $(\mathbb{1}+a)^* = a^*$
- $(\mathbb{1}+a)a^* = a^*$
- $a^* = (aa)^* + a(aa)^*$
- $(ab)^* = \mathbb{1} + a(ba)^*b$
- $a^*a = aa^*$
- $a^* = (\mathbb{1}+a+a^2+\cdots+a^{n-1})(a^n)^*,\ (n>1)$

exercícios: demonstrar estas equivalências

Seja A um alfabeto, o conjunto das expressões regulares $RegExp(A)$ sobre o alfabeto A equipado das constantes e das operações habituais, isto é $(RegExp(A), +, \cdot, *, \emptyset, \epsilon)$, **forma uma álgebra de Kleene**

nota-se que $\mathbb{0} \rightsquigarrow \emptyset$, $\mathbb{1} \rightsquigarrow \epsilon$, $\times \rightsquigarrow \cdot$

Demonstração: basta verificar que as expressões regulares verificam **todos** os axiomas das álgebras de Kleene, ou sejam que esses são equivalências verificadas nas expressões regulares

por exemplo, $x.\emptyset \sim \emptyset$.
$L(x.\emptyset) = \{a.b \mid a \in L(x) \wedge b \in L(\emptyset)\} = \{a.b \mid a \in L(x) \wedge b \in \emptyset\} = \emptyset$ (porque $b \in \emptyset$, e $a.\emptyset = \emptyset$) $= L(\emptyset)$.
QED.

(**exercício**: demonstrar os casos restantes)

vamos, nos acetatos que seguem, comodamente confundir expressões regulares e álgebras de Kleene

confundiremos igualmente a equivalência $\sim$ de expressões regulares com a igualdade $=$ de termos de uma álgebra de Kleene

notação: pelas mesmas questões de comodidade, poderemos usar r^+ onde se espera rr^* ou $r?$ onde se espera $(\epsilon + r)$

- seja $\Sigma = \{a, b\}$, demonstrar que

$$b?(ab)^*a? \quad = \quad (ab)^* + b(ab)^* + (ab)^*a + b(ab)^*a$$

- sejam r e s duas expressões regulares, demonstrar que

$$(r^*s)^* = \epsilon + (r + s)^*s$$

- seja $\Sigma = \{0, 1\}$, demonstrar que

$$(\epsilon + 1)(01)^*(\epsilon + 0) \quad = \quad (01)^* + 1(01)^* + (01)^*0 + 1(01)^*0$$

- seja $\Sigma = \{a, b\}$, consideremos o raciocínio seguinte:

$$\begin{aligned}(b + ab + aab)^*(\epsilon + a + aa) &= (a?a?b)^*a?a? \\ &= a?a?(ba?a?)^* \\ &= (\epsilon + a + aa)(b + ba + baa)^*\end{aligned}$$

  - que linguagem está associada as expressões regulares envolvidas?
  - a que equivalências temos de recorrer para validá-lo?
  - demonstrar estas equivalências.

algoritmos de base para as expressões regulares

retomemos aqui as definições de base

seja A um alfabeto finito, o conjunto $RegExp(A)$ das expressões regulares sobre o alfabeto A é definida pela gramática BNF (*Backus Naur Form*) seguinte

$$\begin{aligned} RegExp(A) \quad ::= \quad & \emptyset \mid \epsilon \mid a \\ & \mid (RegExp(A) + RegExp(A)) \\ & \mid (RegExp(A) \ . \ RegExp(A)) \\ & \mid RegExp(A)^* \end{aligned}$$

(com $a \in A$)

como é já habitual, omitiremos as parêntesis e o . usando as convenções (prioridades etc..) habituais desde que não cause ambiguidades

se admitirmos que o alfabeto é o tipo char em OCaml,

```
type regexp =
        | V                             (* a expressão regular vazia *)
        | E                             (* epsilon                   *)
        | C of char                     (* a de A, com A=char        *)
        | U of regexp * regexp          (* a + b                     *)
        | P of regexp * regexp          (* a.b                       *)
        | S of regexp                   (* a*                        *)
```

$c(a+b)^*d \longrightarrow$ `P (C 'c' , P (S (U (C 'a' , C 'b')) , C 'd'))`

designamos por $|r|_\Sigma$ o comprimento alfabético de uma expressão regular r e é definida pelo número de símbolos do alfabeto Σ que r contém

```
(* alphabet length of a regular expression*)
let rec alfa_length = function
 | V | E                -> 0
 | C c                  -> 1
 | U (a,b) | P (a,b) -> alfa_length a + alfa_length b
 | S r                  -> alfa_length r
```

$$|c(a+b)^*d|_\Sigma = 4$$

propriedade da linguagem vazia

uma expressão regular r denota a linguagem vazia se $L(r) = \emptyset$, ou equivalentemente $r \sim \emptyset$

ou seja (versão álgebra de Kleene) se r é equivalente a $\emptyset$ (i.e. $r = \emptyset$)

neste caso diz-se de r que tem a **propriedade da linguagem vazia** (abreviada em $elp(r)$ ou $\emptyset(r)$)

esta propriedade é trivial de determinar visto precisar somente de uma verificação estrutural da expressão regular em causa:

$$\emptyset(r) = \begin{cases} \top & se\ r = \emptyset \\ \bot & se\ r = \epsilon \\ \bot & se\ r = a \\ \emptyset(a) \vee \emptyset(b) & se\ r = a.b \\ \emptyset(a) \wedge \emptyset(b) & se\ r = a + b \\ \bot & se\ r = q^* \end{cases}$$

$$\emptyset((aa + bc)^*.\emptyset) = \bot \vee \top = \top$$

```
(* empty language property *)
let rec elp = function
  | V         -> true
  | P (a, b) -> (elp a) ||  (elp b)
  | U (a, b) -> (elp a) &&  (elp b)
  | _         -> false

(* utilitários *)
let empty      = elp
let notempty r = not (elp r)
```

uma expressão regular r tem a **propriedade da palavra vazia** ou ainda que é **anulável** quando $\epsilon \in L(r)$
(notação $ewp(r)$ ou ainda $\epsilon(r)$)

de forma semelhante à propriedade anterior, a definição da propriedade **ewp** é estrutural

$$\epsilon(r) = \begin{cases} \bot & se\ r = \emptyset \\ \top & se\ r = \epsilon \\ \bot & se\ r = a \\ \epsilon(a) \wedge \epsilon(b) & se\ r = a.b \\ \epsilon(a) \vee \epsilon(b) & se\ r = a + b \\ \top & se\ r = q^* \end{cases}$$

$$\epsilon((\epsilon + b).(ab)^*) = (\top \vee \bot) \wedge \top = \top$$

```
(* empty word property *)
let rec ewp = function
  | E | S  _   -> true
  | P (a, b)   -> (ewp a) &&  (ewp b)
  | U (a, b)   -> (ewp a) ||  (ewp b)
  | _          -> false


(* utilitários *)
let epsilon      = ewp
let notepsilon r = not (ewp r)
```

duas expressões regulares r e s são equi-anuláveis quando

$$\zeta(r, s) \triangleq (\epsilon(r) = \epsilon(s))$$

```
(* equi-nullability*)
let equinullable r s =  ewp r = ewp s
```

$$\zeta((a + b)^*, (ab)^*(ba + aac)^*) = (\top = \top) = \top$$

$$\zeta((a + b)^*, (ab)(ba + aac)^*) = (\top = \bot) = \bot$$

a função $\xi : RegExp(A) \rightarrow \mathbb{B}$ testa se uma expressão regular r representa uma linguagem que contém no máximo o ϵ (i.e. $L(r) \subseteq \{\epsilon\}$, ou ainda $L(r) = \emptyset \vee L(r) = \{\epsilon\}$)

$$\xi(r) = \begin{cases} \top & se\ r = \emptyset \\ \top & se\ r = \epsilon \\ \bot & se\ r = a \\ \xi(a) \wedge \xi(b) & se\ r = a + b \\ \emptyset(a) \vee \emptyset(b) \vee (\xi(a) \wedge \xi(b)) & se\ r = a.b \\ \xi(q) & se\ r = q^* \end{cases}$$

```
let rec atmost_epsilon = function
  | V | E      -> true
  | S a        -> atmost_epsilon a
  | U (a, b)   -> (atmost_epsilon a) && (atmost_epsilon b)
  | P (a, b)   -> (elp a) || (elp b) ||
                  ((atmost_epsilon a) && (atmost_epsilon b))
  | _          -> false
```

$$\xi((ab)^* + \emptyset) = \top \wedge \top = \top$$

determinar se uma linguagem é infinita

a função $\infty : RegExp(A) \to \mathbb{B}$ testa se uma expressão regular r representa uma linguagem infinita (que contém um número infinito de palavras) (i.e. $|L(r)| = \aleph_0$)

$$\infty(r) = \begin{cases} \bot & se\ r = \emptyset \\ \bot & se\ r = \epsilon \\ \bot & se\ r = a \\ \infty(a) \vee \infty(b) & se\ r = a + b \\ (\infty(a) \wedge \neg\emptyset(b)) \vee (\neg\emptyset(a) \wedge \infty(b)) & se\ r = a.b \\ \neg(\xi(q)) & se\ r = q^* \end{cases}$$

```
let rec infinity = function
  | S a       -> not  (atmost_epsilon a)
  | U (a, b)  -> (infinity a) || (infinity b)
  | P (a, b)  -> (infinity a  && notempty b) ||
                 (infinity b  && notempty a)
  | _         -> false
```

$$\begin{array}{rcl} \infty((a^* + b)^* b^*) & = & (\infty((a^* + b)^*) \wedge \neg\emptyset(b)) \vee \\ & & (\infty(b^*) \wedge \neg\emptyset(a^* + b)) \\ & = & (\neg\xi(a^* + b) \wedge \top) \vee \\ & & (\neg\xi(b) \wedge \top) \\ & = & \top \vee \top \\ & = & \top \end{array}$$

conjuntos de expressões

para os algoritmos seguintes precisaremos de uma estrutura para arquivar e manipular conjuntos de expressões, conjuntos de pares de expressões e conjuntos de caracteres

em OCaml, usemos o tipo conjunto equipado com uma função de comparação (do módulo Set)

```
(* module/type for sets of regular expression *)
module ReS = Set.Make (struct
    type t = regexp
    let compare = compare
  end)

(* module/type for pairs of sets of regular expressions *)
module RePS = Set.Make (struct
    type t = ReS.t * ReS.t
    let compare = compare
  end)

(*module/type for set of chars *)
module CS = Set.Make(Char)
```

por exemplo estender as funções sobre expressões regulares para conjuntos de expressões regulares é assim fácil, por exemplo:

```
(* extending usual functions over regular expressions   *)
(* to sets of regular expressions                       *)
let ewps             = ReS.exists ewp
let atmost_epsilons = ReS.for_all atmost_epsilon
let infinitys        = ReS.exists infinity
```

uma operação utilitária quando se manipula um conjunto de expressões regulares, digamos $S \triangleq \{s_1, \cdots, s_n\}$, é a concatenação à direita de uma expressão regular r, denotada por $S \odot r$,

i.e. $\{s.r \mid s \in S\}$

$$S \odot r = \begin{cases} \emptyset & se\ r = \emptyset \\ S & se\ r = \epsilon \\ \{s.r \mid s \in S\} & \text{senão} \end{cases}$$

```
(* S.r  when S is a set of regexps  *)
(* and r a regexp                   *)
let rigth_concat s = function
  | V -> ReS.empty
  | E -> s
  | r -> ReS.map (fun e -> P (e,r)) s

(*  utility function,   *)
(*  for nice infix use  *)
let ( *.* ) = rigth_concat
```

para simplificar, poderemos anotar $S \odot r$ por $S.r$ ou mesmo $S\,r$ quando o contexto não induzir ambiguidades

derivadas parciais de expressões regulares

um conceito interessante que envolve expressões regulares e linguagens é o de **linguagens quocientes esquerdas**

sejam Σ um alfabeto, L uma linguagem sobre este alfabeto e w uma palavra de Σ^*

a **linguagem quociente esquerda** de L em relação a w, denotada por $E_w(L)$ é definida por

$$E_w(L) \triangleq \{v \mid v \in \Sigma^* \wedge wv \in L\}$$

ou seja, é o conjunto das palavras v de Σ^* que sufixando w formam uma palavra de L

alternativamente, se considerarmos L' como o subconjunto de L das palavras que têm w como prefixo, $E_w(L)$ é o conjunto dos sufixos de L'

nota: na literatura encontra-se a notação alternativa $w^{-1}L$ para $E_w(L)$

um caso particular de $E_w(L)$ ocorre quando w é uma letra de Σ (digamos $a \in \Sigma$) assim, parafraseando,

$$E_a(L) \triangleq \{v \mid v \in \Sigma^* \wedge av \in L\}$$

uma nota interessante é a de que

$$L = \bigcup_{a \in \Sigma} a.E_a(L)$$

(notação alternativa encontrada na literatura: $a^{-1}L$)

na literatura, a linguagem quociente esquerda $E_a(L)$ é também designada de **linguagem derivada de L em a**$(com\ a \in \Sigma)$

Lemma

seja Σ um alfabeto, $L \subseteq \Sigma^*$ uma linguagem e $a \in \Sigma$

$$ax \in L \textbf{ se e só se } x \in E_a(L)$$

uma questão é **como calcular $E_a(L)$?**

o **cálculo** de uma linguagem derivada de **uma linguagem regular** faz-se comodamente de forma **estruturalmente recursiva** sobre a expressão regular que a gere, i.e. **sobre r tal que $L(r) = L$**

o conceito de derivada de uma expressão regular foi definido originalmente por **Brzozowski** nos anos 60[8] no contexto da construção de circuitos digitiais sequenciais a partir de expressões regulares complementadas com intersecção e complemento

Mirkin, na mesma altura, complementou (na verdade, **generalizou**) este conceito com a noção de base e de prebase[9] de uma expressão regular para a construção de um autómato não determinístico a partir de uma expressão regular

este mesmo conceito foi redescoberto trinta anos mais tarde por **Antimirov**, que as designou de **derivadas parciais**
[10]

[8] J.A. Brzozowski, Derivatives of regular expressions, J. ACM 11(4) (1964) p481-494.

[9] B. Mirkin, An algorithm for constructing a base in a language of regular expressions, Eng. Cybern. 5 (1966) p110-116.

[10] V.M. Antimirov, Partial derivatives of regular expressions and finite automaton constructions, Theor. Comput. Sci. 155(2) (1996) p291-319.

na literatura o conceito de derivada parcial é expresso tanto como uma operação que devolve uma expressão regular ou então que devolve um conjunto de expressões regulares

a diferença reside essencialmente no tratamento do operador $+$ nas derivadas parciais

num caso preferiremos devolver $r + s$ enquanto no outro $\{r, s\}$

na nossa exposição optaremos pela versão **conjunto**[11]:

$\partial_a(r)$, a derivada parcial de r na letra a, é o conjunto de expressões regulares s que representam as palavras que podem ser formadas tais que

$$\underline{a.s \in L(r)}$$

(i.e. escolhendo a na primeira posição)

[11] há vantagens que ultrapassam a exposição desta aula ligadas às questões de igualdade.

Considerando um alfabeto Σ, uma expressão regular r e uma letra a de Σ, o conjunto $\partial_a(r)$ das derivadas parciais de r relativamente a a é definido de forma estruturalmente recursiva por

$$\partial_a(r) = \begin{cases} \emptyset & se\ r = \emptyset \\ \emptyset & se\ r = \epsilon \\ \emptyset & se\ r = b \quad \wedge \quad b \neq a \\ \{\epsilon\} & se\ r = a \\ \partial_a(s) \cup \partial_a(t) & se\ r = s + t \\ \partial_a(s) \odot t \cup \partial_a(t) & se\ r = s.t \quad \wedge \quad \epsilon(s) \\ \partial_a(s) \odot t & se\ r = s.t \quad \wedge \quad \neg\epsilon(s) \\ \partial_a(s) \odot s^* & se\ r = s^* \end{cases}$$

derivada parcial de uma expressão regular

```
(* partial derivative of a regular expression *)
let rec pd a = function
 | V | E                -> ReS.empty
 | C  b    when b<>a    -> ReS.empty
 | C  b                 -> ReS.singleton E
 | U (r,s)              -> ReS.union (pd a r)  (pd a s)
 | P (r,s) when ewp r -> ReS.union ((pd a r) *.* s) (pd a s)
 | P (r,s)              -> (pd a r) *.* s
 | S r     as re        -> (pd a r) *.* re
```

exemplos de derivadas parciais

$$
\begin{aligned}
\partial_a(abab + abba) &= \partial_a(abab) \cup \partial_a(abba) \\
&= \partial_a(a) \odot bab \cup \partial_a(a) \odot bba \\
&= \{\{\epsilon\} \odot bab, \{\epsilon\} \odot bba\} \\
&= \{\epsilon.bab, \epsilon.bba\} \\
&= \{bab, bba\}
\end{aligned}
$$

$$
\begin{aligned}
\partial_b(abab + abba) &= \partial_b(abab) \cup \partial_b(abba) \\
&= \partial_b(a) \odot bab \cup \partial_b(a) \odot bba \\
&= \emptyset \odot bab \cup \emptyset \odot bba \\
&= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\partial_a(a^* b) &= \partial_a(a^*) \odot b \cup \partial_a(b) \\
&= (\partial_a(a) \odot a^*) \odot b \cup \emptyset \\
&= (\{\epsilon\} \odot a^*) \odot b \\
&= \{\epsilon.a^*\} \odot b \\
&= \{a^*\} \odot b \\
&= a^* b
\end{aligned}
$$

$$
\begin{aligned}
\partial_a((ab)^* a) &= \partial_a((ab)^*) \odot a \cup \partial_a(a) \\
&= \partial_a((ab)^*) \odot a \cup \epsilon \\
&= (\partial_a(ab) \odot (ab)^*) \odot a \cup \epsilon \\
&= ((\partial_a(a) \odot b) \odot (ab)^*) \odot a \cup \epsilon \\
&= ((\{\epsilon\} \odot b) \odot (ab)^*) \odot a \cup \epsilon \\
&= \{b(ab)^* a, \epsilon\}
\end{aligned}
$$

a noção de derivada estende-se naturalmente para dois casos interessantes:

sejam S um conjuntos de expressões regulares e a uma letra do alfabeto

$$\partial_a(S) = \bigcup_{r \in S} \partial_a(r)$$

```
(* utility function                                          *)
(* repeated unions over sets of regular expressions        *)
(* U_(r in S) f(r)                                         *)
let rec unions f s =
   ReS.fold (fun re acc -> ReS.union (f re) acc ) s ReS.empty


(* partial derivative of a set of regular expressions s *)
(* over a letter a*)
let rec pds a s = unions (pd a) s
```

sejam $w \in \Sigma^*$, a uma letra de Σ e r uma expressão regular
a derivada de r sobre uma palavra w é definida por recursão estrutural sobre a constituição da palavra w

$$\partial_w(r) \triangleq \begin{cases} \{r\} & se\ w = \epsilon \\ \partial_a(\partial_v(r))) & se\ w = v.a) \end{cases}$$

(nota: é importante realçar que a recursão é feita da direita para a esquerda sobre a palavra w)

```
(* partial derivative of a set of regular expressions sr *)
(* by a word p given as a list of its characters in      *)
(* reverse order <-- must be ensured on first call       *)
let rec pdw (sr: ReS.t) = function
  | []    -> sr
  | a::tl -> pds a (pdw sr tl)
```

algoritmia para testar $w \in L(r)$

a linguagem do conjunto de expressões regulares $\partial_a(r)$ é o quociente esquerdo de r por a, i.e.

$$L(\partial_a(r)) = E_a(L(r)))$$

Theorem

Sejam Σ um alfabeto, w uma palavra de Σ^* e r uma expressão regular sobre Σ^*

$$w \in L(r) \textbf{ se e só se } \epsilon(\partial_w(r))$$

intuitivamente, se a derivada de r por w devolver um conjunto de expressões regulares **ao qual pertence ϵ** então isso significa que **um dos sufixos aceites a seguir a w é ϵ** (ϵ pertence ao quociente esquerdo de w), logo $w.\epsilon \in L(r)$ ou seja $w \in L(r)$

$w \in L(r)$ **se e só se** $\epsilon(\partial_w(r))$

este teorema sugere um algoritmo para testar se $w \in L(r)$: basta calcular a derivada de r por w e verificar se o resultado contém ϵ

```
(* belongs r p = the word p is in the language  *)
(* induced by the regular expression r          *)
(* p is given as a list of its characters in    *)
(* reverse order                                *)
let belongs r lc = ewps (pdw (ReS.singleton r) lc)
```

este algoritmo é essencialmente o algoritmo de Ken Thompson

e funciona igualmente no caso das expressões regulares estendidas com intersecção e complemento (o caso considerado originalmente pelo Brzozowski)

$$
\begin{aligned}
\partial_{abb}(ab^*) &= \partial_b(\partial_b(\partial_a(ab^*))) \\
&= \partial_b(\partial_b(\partial_a(a) \odot b^*)) \\
&= \partial_b(\partial_b(\{b^*\})) \\
&= \partial_b(\partial_b(b) \odot b^*) \\
&= \partial_b(\{b^*\}) \\
&= \{b^*\}
\end{aligned}
$$

$$\epsilon(\partial_{abb}(ab^*)) = \epsilon(\{b^*\}) = \top$$

logo

$$abb \in L(ab^*)$$

algoritmia para testar $r \sim s$

antes de apresentar o algoritmo para testar a equivalência de duas expressões regulares, vamos apresentar o contexto formal que justifica que este algoritmo exista e funcione

o conjunto das derivadas parciais $PD(\alpha)$ de uma expressão regular α é definido por

$$PD(\alpha) = \bigcup_{w \in \Sigma^*} \partial_w(\alpha)$$

Antimirov demonstrou que o conjunto $PD(\alpha)$ **é sempre finito**
e que $|PD(\alpha)| \leq |\alpha|_{\Sigma} + \mathbf{1}$

mas como construir efetivamentre (... *por algoritmo*) o conjunto $PD(\alpha)$ para uma dada expressão regular α?

Champarnaud e Ziadi[12] mostraram que é possível calcular de forma recursiva a função $\pi(\alpha)$, que se designa de **suporte da expressão regular** α

$$\begin{array}{lcl@{\qquad}lcl} \pi(\mathbb{0}) & = & \emptyset & \pi(\alpha+\beta) & = & \pi(\alpha)\cup\pi(\beta) \\ \pi(\mathbb{1}) & = & \emptyset & \pi(\alpha\beta) & = & \pi(\alpha)\odot\beta\cup\pi(\beta) \\ \pi(a) & = & \{\epsilon\} & \pi(\alpha^*) & = & \pi(\alpha)\odot\alpha^* \end{array}$$

essa função é a chave para uma construção algorítmica do conjunto $PD(\alpha)$

[12] J.-M. Champarnaud, D. Ziadi, From Mirkin's prebases to Antimirov's word partial derivatives, Fundam. Inform. 45(3) (2001) 195-205.

$$
\begin{array}{lcl@{\qquad}lcl}
\pi(\mathbb{0}) & = & \emptyset & \pi(\alpha+\beta) & = & \pi(\alpha)\cup\pi(\beta) \\
\pi(\mathbb{1}) & = & \emptyset & \pi(\alpha\beta) & = & \pi(\alpha)\odot\beta\cup\pi(\beta) \\
\pi(a) & = & \{\epsilon\} & \pi(\alpha^*) & = & \pi(\alpha)\odot\alpha^*
\end{array}
$$

```
(* support of a regular expression *)
let rec pi = function
  | V | E       -> ReS.empty
  | C  _        -> ReS.singleton E
  | U (r,s)     -> ReS.union (pi r) (pi s)
  | P (r,s)     -> ReS.union ((pi r) *.* s) (pi s)
  | S r  as re -> (pi r) *.* re
```

desta definição é possível demonstrar que $|\pi(\alpha)| \leq |\alpha|_\Sigma$

Champarnaud e Ziadi mostraram igualmente que

$$PD(\alpha) = \{\alpha\} \cup \pi(\alpha)$$

temos o nosso algoritmo!

```
(* set of partial derivatives of a regexp *)
let pdregexp r =  ReS.union (ReS.singleton r) (pi r)
```

e constataram novamente que $|PD(\alpha)| \leq |\alpha|_\Sigma + 1$

$$
\begin{aligned}
\pi((ab+c)^*) &= \pi(ab+c) \odot (ab+c)^* \\
&= (\pi(ab) \cup \pi(c)) \odot (ab+c)^* \\
&= (\pi(a) \odot b \cup \pi(b) \cup \{\epsilon\}) \odot (ab+c)^* \\
&= (\{\epsilon\} \odot b \cup \{\epsilon\} \cup \{\epsilon\}) \odot (ab+c)^* \\
&= (\{b\} \cup \{\epsilon\}) \odot (ab+c)^* \\
&= \{b(ab+c)^*, (ab+c)^*\}
\end{aligned}
$$

$$
\begin{aligned}
PD((ab+c)^*) &= \{(ab+c)^*\} \cup \pi((ab+c)^*) \\
&= \{(ab+c)^*\} \cup \{b(ab+c)^*, (ab+c)^*\} \\
&= \{b(ab+c)^*, (ab+c)^*\}
\end{aligned}
$$

seja $\phi(\alpha)$ a função unária sobre expressões regulares definida por

$$\phi(\alpha) = \begin{cases} \emptyset & \textit{se } \neg\epsilon(\alpha) \\ \{\epsilon\} & \textit{se } \epsilon(\alpha) \end{cases}$$

Antimirov e Mosses[13] constataram que para qualquer expressão regular α

$$\alpha = \phi(\alpha) \bigcup_{a \in \Sigma} a(\sum \partial_a(\alpha))$$

assim verificar se duas expressões regulares α e β são equivalentes basta verificar

$$\phi(\alpha) \bigcup_{a \in \Sigma} a(\sum \partial_a(\alpha)) = \phi(\beta) \bigcup_{a \in \Sigma} a(\sum \partial_a(\beta))$$

[13] V.M. Antimirov, P.D. Mosses, Rewriting extended regular expressions, in: G. Rozenberg, A. Salomaa (Eds.), DLT, World Scientific, 1994, pp.195-209.

a equação

$$\phi(\alpha) \bigcup_{a \in \Sigma} a(\sum \partial_a(\alpha)) = \phi(\beta) \bigcup_{a \in \Sigma} a(\sum \partial_a(\beta))$$

é um ingrediente essencial para o algoritmo que decide a equivalência de expressões regulares que vamos apresentar em seguida

de facto, à luz desta equação, determinar se $\alpha = \beta$ pode resumir-se em perceber se $\phi(\alpha) = \phi(\beta)$ e se $\forall a \in \Sigma, \partial_a(\alpha) = \partial_a(\beta)$

se contarmos também com o facto de os conjuntos de derivadas parciais de expressões regulares serem **finitos**

e que o problema da pertença de uma palavra w a uma linguagem $L(\alpha)$ se poder resumir a testar se $\epsilon(\partial_w(\alpha))$

então obtemos igualmente:

$$(\forall w \in \Sigma^*, \epsilon(\partial_w(\alpha)) = \epsilon(\partial_w(\beta))) \longleftrightarrow \alpha = \beta$$

em contraposição, provar que $\alpha \neq \beta$ pode fazer-se ao provar que para uma palavra $w \in \Sigma^*$

$$\epsilon(\partial_w(\alpha)) \neq \epsilon(\partial_w(\beta)) \longrightarrow \alpha \neq \beta$$

estas duas propriedades sugerem um algoritmo iterativo/recursivo: calcular derivadas e testar a equi-anulabilidade até chegar a um teste negativo (resposta $\bot$) ou então ao fim do conjunto de derivadas existentes (resposta $\top$).

um algoritmo para calcular se $\alpha = \beta$

Require: $S = \{(\{\alpha\}, \{\beta\})\} \ \wedge \ H = \emptyset$
Ensure: $\top$ se e só se $\alpha = \beta$, senão $\bot$

function EquivP(S, H)
  while $S \neq \emptyset$ **do**
    $(S_\alpha, S_\beta) \leftarrow pop(S)$
    if $\epsilon(S_\alpha) \neq \epsilon(S_\beta)$ **then**
      return $\bot$
    end if
    $H \leftarrow H \cup \{(S_\alpha, S_\beta)\}$
    for all $a \in \Sigma$ **do**
      $(S'_\alpha, S'_\beta) \leftarrow \partial_a(S_\alpha), \partial_a(S_\beta)$
      if $(S'_\alpha, S'_\beta) \notin H$ **then**
        $S \leftarrow S \cup \{S'_\alpha, S'_\beta\}$
      end if
    end for
  end while
  return $\top$
end function

```
(* symbols_set r = set of symbols of the alphabet *)
(* used in the regular expression r *)
let rec symbol_set = function
 | C  a                -> CS.singleton a
 | E         | V       -> CS.empty
 | S a                 -> symbol_set a
 | U (a, b) | P (a, b)-> CS.union (symbol_set a)(symbol_set b)

(* extract an element e from s (i.e. e is removed from s) *)
let pop s = let el = RePS.choose s in (el, RePS.remove el s)

(* let the initial s be {(q,r)}. Are the two  *)
(* regular expressions q and r equivalent?    *)
let rec equivP (s: RePS.t) h symbols =
  if RePS.is_empty s then true
  else let ((gamma,delta),new_s) = pop s in
       if ewps gamma <> ewps delta then false
       else
         let new_h = RePS.add (gamma,delta) h in
         (equivP
           (CS.fold
             (fun a acc ->
               let dg,dd = pds a gamma, pds a delta in
                 if not (RePS.mem (dg,dd) new_h)
                 then RePS.add (dg,dd) acc
                 else acc)
             symbols new_s)
           new_h symbols)

(* (equivP (RePS.singleton (ReS.singleton r, ReS.singleton s))
          RePS.empty (symbol_set (U (r,s))))*)
```

Foi demonstrado[14] que a função *EquivP* **termina** (lembre-se do limite superior ao cardinal do conjunto das derivadas parciais)

e que **respeita a propriedade de correcção**

$$\mathit{EquivP}(\{(\{\alpha\},\{\beta\})\},\emptyset) = \begin{cases} \top & \textbf{se } \alpha = \beta, \\ \bot & \textbf{senão (i.e. } \alpha \neq \beta) \end{cases}$$

[14] Para uma demonstração feita por computador, pode consultar: N. Moreira, D. Pereira, S. Melo de Sousa. Deciding Kleene Algebra Terms Equivalence in Coq. Journal of Logic and Algebraic Methods in Programming, Elsevier. 28 pages. 2015.

com $\alpha = (ab)^*a$, $\beta = a(ba)^*$, $s_0 = (\{(\alpha, \beta)\})$
considere igualmente $s_1 = (\{\epsilon, b(ab)^*a\}, \{(ba)^*\})$ e $s_2 = (\emptyset, \emptyset)$

executemos $EquivP(s_0, \emptyset)$

i	S_i	H_i	*derivadas*
0	$\{s_0\}$	$\emptyset$	$\partial_a(s_0) = s_1, \partial_b(s_0) = s_2$
1	$\{s_1, s_2\}$	$\{s_0\}$	$\partial_a(s_1) = s_2, \partial_b(s_1) = s_0$
2	$\{s_2\}$	$\{s_0, s_1\}$	$\partial_a(s_2) = s_2, \partial_b(s_2) = s_2$
3	$\emptyset$	$\{s_0, s_1, s_2\}$	$\top$

onde i é o contador de execução do ciclo *while* principal, S_i e H_i são os argumento de *EquivP* na iteração i

o algoritmo termina a sua execução com $S_3 = \emptyset$, logo $EquivP(s_0, \emptyset) = \top$
e assim $\boxed{(ab)^*a \;=\; a(ba)^*}$

a implementação OCaml dada ao algoritmo *EquivP* é puramente funcional

usa recursividade no lugar dos ciclos (aqui “while”, “for all”)

exercício: dar uma implementação imperativa (i.e. fiel ao algoritmo) da função *EquivP*

exercício: executar este algoritmo para demonstrar a equivalência

- da regra de ***denesting***, i.e. $(a+b)^* = a^*(ba^*)^*$ e
- da regra $\mathbb{1} + xx^* = x^*$

Teorema de Kleene, again

Lemma (de Arden)

Consideremos R e S dois termos de uma álgebra de Kleene $(K, +, \times, *, \mathbb{0}, \mathbb{1})$ e x uma incógnita sobre o conjunto K. A equação

$$x = Rx + S$$

*admite como **menor** solução $x = R^*S$*

*Mais, se $\neg\epsilon(R)$, R^*S é a **única** solução.*

(demonstração: dada mais acima! secção "álgebras de Kleene")

este lema permite a construção de um método sistemático de construção de uma expressão regular a partir de um sistema de equações sobre termos de uma álgebra de Kleene

em particular, permite redescobrir o algoritmo por eliminação de estado que transforma um autómato numa expressão regular

porque podemos reduzir um autómato a um sistema de equação deste tipo

considere o seguinte autómato:

extraí-se o sistema de equações seguinte:

$$E_A = (0+1)E_A + 1E_B, \quad E_B = (0+1)E_C, \quad E_C = \epsilon + (0+1)E_D, \quad E_D = \epsilon$$

onde E_i representa a expressão (regular) que o estado i reconhece

pelo Lema de Arden, e tratando os estados pela ordem D, C, B e A:

$E_D = \epsilon, \quad E_C = \epsilon + 0 + 1, \quad E_B = 0 + 1 + (0+1)^2,$

$$E_A = (0+1)^*(10 + 11 + 1(0+1)^2)$$

consideremos o sistema

$$E_1 = bE_2, \quad E_2 = aE_1 + bE_3, \quad E_3 = \epsilon + bE_1$$

então, este pode ser simplificado em

$$E_1 = bE_2, \quad E_3 = \epsilon + bbE_2 \quad e \quad E_2 = (ab + bbb)E_2 + b$$

logo $E_2 = (ab + bbb)^* b$ e $E_1 = b(ab + bbb)^* b$

exercício:

considere o sistema $X = aX + bY, \quad Y = \epsilon + cY + dX$

que expressão regular obtém para X

- se eliminar X seguido de Y?
- se eliminar Y seguido de X?

o que pode concluir sobre as duas soluções obtidas?

o método usando o lema de Arden é fundamentalmente o mesmo que os métodos de eliminação de estados e o método de Kleene ou ainda de Mac Naughton - Yamada

construir um autómato determinista a partir de uma expressão regular

tirando proveito do facto que o conjunto das derivadas de uma expressão regular r é finito, a ideia é usar as derivadas deste conjunto como estados do autómato em questão e considerar as transições da forma $r \xrightarrow{a} \partial_a(r)$.

Definition

seja r uma expressão regular sobre um alfabeto Σ, seja $B = (Q, \Sigma, \delta, \{r\}, F)$ o autómato cujo o estado inicial é $\{r\}$, cujo conjunto de transições é fechado pela operação $r \xrightarrow{a} \partial_a(r)$ para todo o $a \in \Sigma$.

O fecho desta operação define assim Q e δ.

Os estados finais f são os estados de Q para os quais $\epsilon(f)$ é verdade.

Theorem

B é determinista, completo e reconhece $L(r)$

(demonstração, no livro de J. Sakarovitch - ver bibliografia)

consideremos $r = a^* + ab$, designemos r_0 como $\{r\}$, temos:

$$\delta = \begin{cases} r_0 \xrightarrow{a} \partial_a(\{r\}) = \{a^*, b\} = \underline{r_1} \\ r_0 \xrightarrow{b} \partial_b(\{r\}) = \emptyset = r_2 \\ \hline r_1 \xrightarrow{a} \partial_a(r_1) = \{a^*\} = \underline{r_3} \\ r_1 \xrightarrow{b} \partial_b(r_1) = \{\epsilon\} = \underline{r_4} \\ \hline r_2 \xrightarrow{a} \partial_a(r_2) = \emptyset = r_2 \\ r_2 \xrightarrow{b} \partial_b(r_2) = \emptyset = r_2 \\ \hline r_3 \xrightarrow{a} \partial_a(r_3) = \{a^*\} = r_3 \\ r_3 \xrightarrow{b} \partial_b(r_3) = \emptyset = r_2 \\ \hline r_4 \xrightarrow{a} \partial_a(r_4) = \emptyset = r_2 \\ r_4 \xrightarrow{b} \partial_b(r_4) = \emptyset = r_2 \end{cases}$$

$Q = \{r_0, r_1, r_2, r_3, r_4\}$
$I = r_0$
$F = \{r_1, r_3, r_4\}$

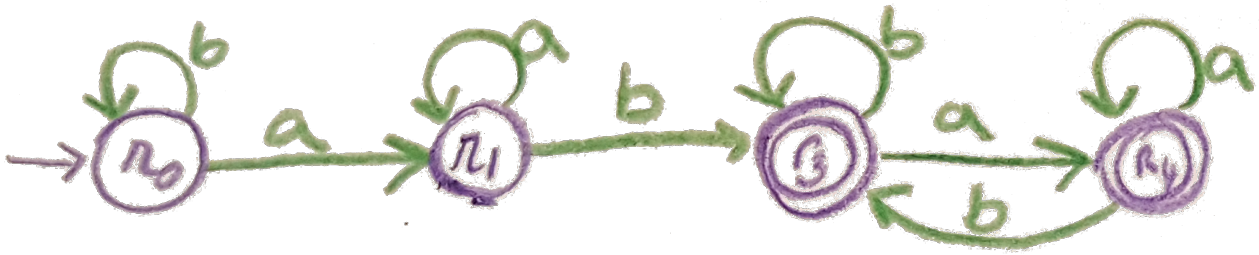
aplicar o algoritmo de Brzozowski sobre a expressão regular

$$r = (a + b)^* ab(a + b)^*$$

aplicar o algoritmo de Brzozowski sobre a expressão regular

$$r = (a + b)^* ab(a + b)^*$$

o detalhe da resolução (por facultar) deve resultar neste autómato



implementar em OCaml o algoritmo de Brzozowski

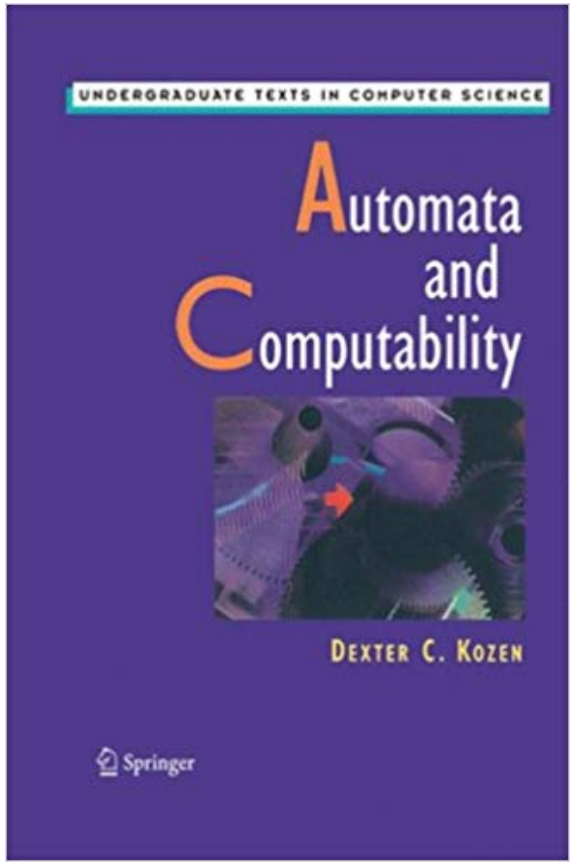
Conclusão. Quer saber mais?

este material apresentado foi realizado com base nas seguintes referências

Automata and Complexity

Dexter Kozen
Undergraduate Texts in Computer Science, 400 p. Springer (August 1997)

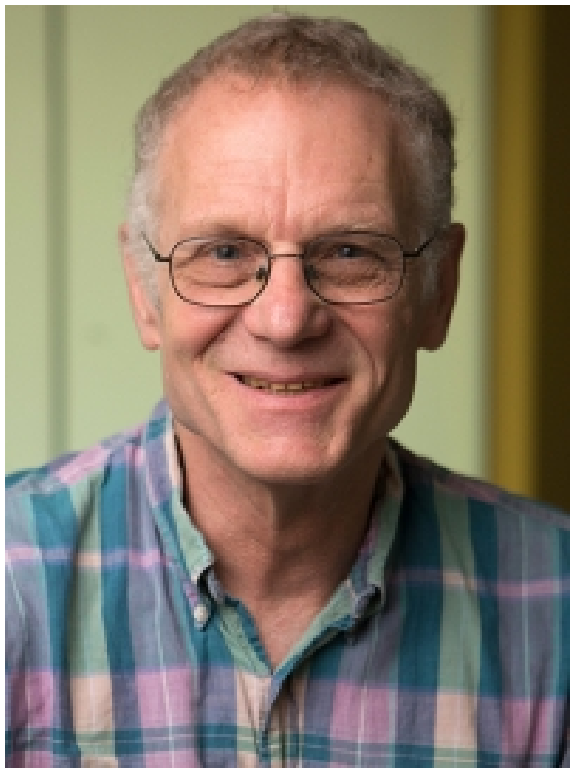
as Álgebra de Kleene na primeira pessoa e apresentadas no contexto de um curso como o presente curso



este material apresentado foi realizado com base nas seguintes referências

Introduction to Kleene Algebra (CS786) (link)

curso proferido por Dexter Kozen



as Álgebra de Kleene na primeira pessoa e apresentadas no contexto de um curso avançado mas alinhado com a perspectiva do presente curso

este material apresentado foi realizado com base nas seguintes referências

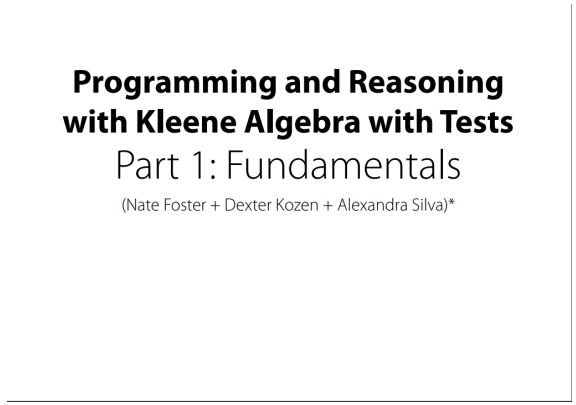
a course on Finite Automata and Formal Languages

uma aula (link) do Thierry Coquand sobre o tema desta UC onde as álgebras de Kleene e as expressões regulares têm um tratamento próximo do que aqui se apresentou.

este material apresentado foi realizado com base nas seguintes referências

Programming and Reasoning with Kleene Algebra with Tests

Nate Foster, Dexter Kozen, Alexandra Silva
47th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2020).
New Orleans, Louisiana, United States



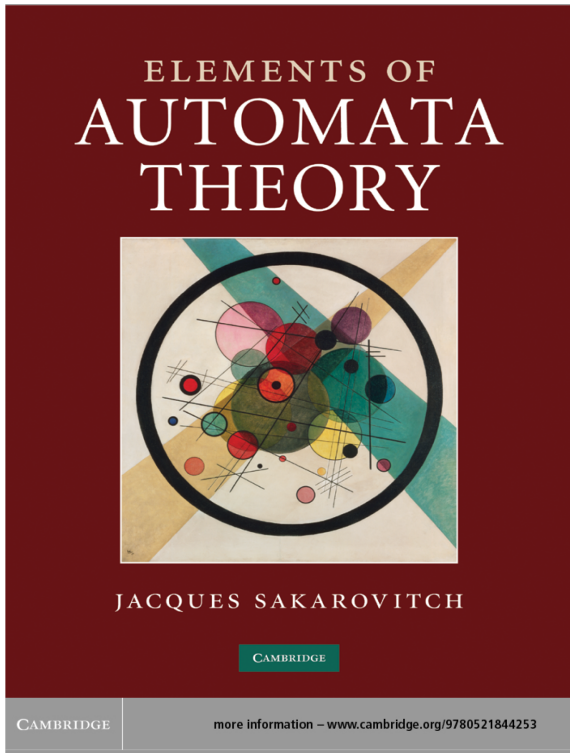
as Álgebra de Kleene no contexto dum tutorial recente numa conferência sobre a ciência e a engenharia das linguagens de programação (a prestigiada POPL)

este material apresentado foi realizado com base nas seguintes referências

Elements of Automata Theory

de Jacques Sakarotich

um livro muito detalhado e sólido sobre o assunto, *mathematically oriented*.

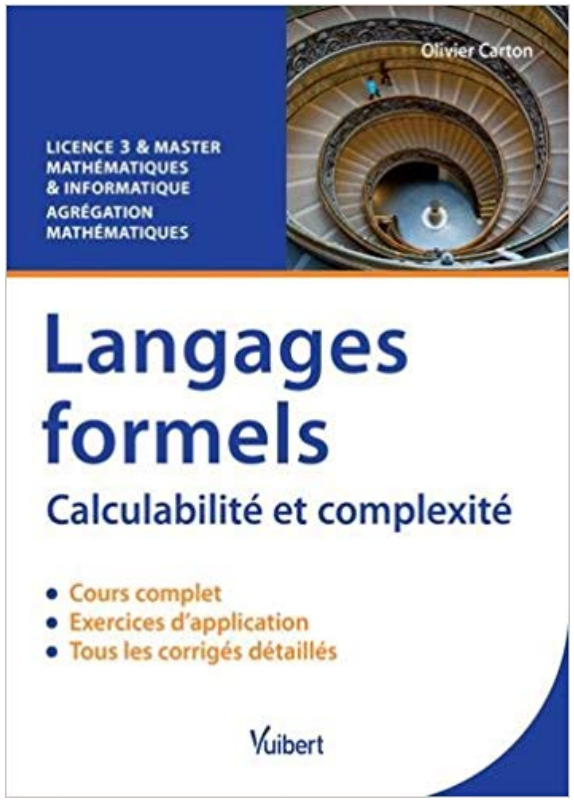


este material apresentado foi realizado com base nas seguintes referências

Langages formels - Calculabilité et complexité

de Oliver Carton

um livro moderno, completo e sólido sobre o assunto, embora em francês.

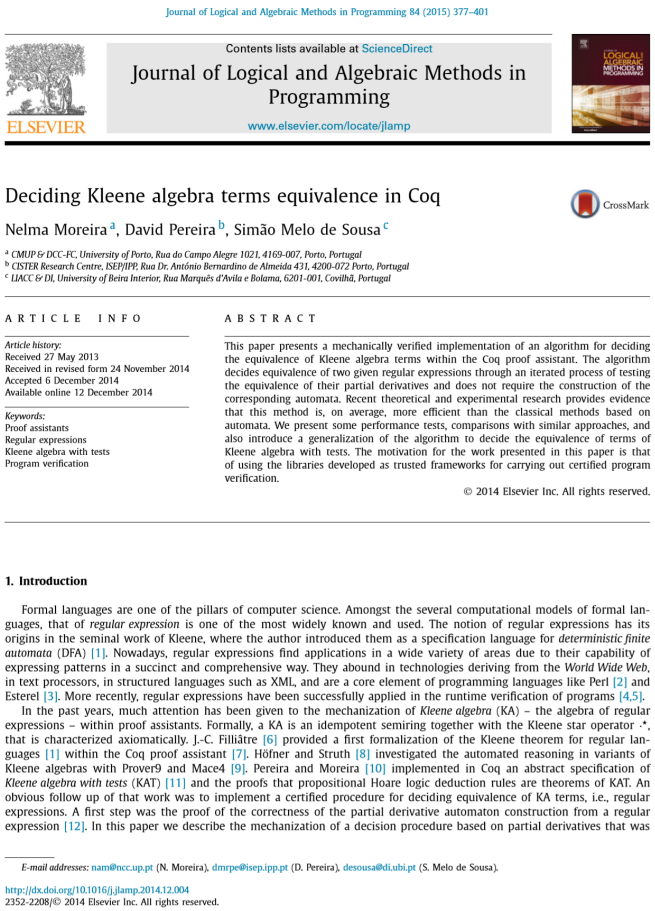


este material apresentado foi realizado com base nas seguintes referências

Deciding Kleene algebra terms equivalence in COQ

N. .Moreira, D. Pereira, S. Melo de Sousa. Journal of Logic and Algebraic Methods in Programming, Elsevier. 28 pages. 2015.

Para além de um resumo do conteúdo desta aula, este artigo introduz o algoritmo de equivalência de termos KA (e.g. expressões regulares) apresentado na aula, com a sua prova de correção (realizada por computador).



Journal of Logical and Algebraic Methods in Programming 84 (2015) 377–401

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Deciding Kleene algebra terms equivalence in Coq

Nelma Moreira [a], David Pereira [b], Simão Melo de Sousa [c]

[a] CMUP & DCC-FC, University of Porto, Rua do Campo Alegre 1021, 4169-007, Porto, Portugal
[b] CISTER Research Centre, ISEP/IPP, Rua Dr. António Bernardino de Almeida 431, 4200-072 Porto, Portugal
[c] LIACC & DI, University of Beira Interior, Rua Marquês d'Avila e Bolama, 6201-001, Covilhã, Portugal

ARTICLE INFO

Article history:
Received 27 May 2013
Received in revised form 24 November 2014
Accepted 6 December 2014
Available online 12 December 2014

Keywords:
Proof assistants
Regular expressions
Kleene algebra with tests
Program verification

ABSTRACT

This paper presents a mechanically verified implementation of an algorithm for deciding the equivalence of Kleene algebra terms within the Coq proof assistant. The algorithm decides equivalence of two given regular expressions through an iterated process of testing the equivalence of their partial derivatives and does not require the construction of the corresponding automata. Recent theoretical and experimental research provides evidence that this method is, on average, more efficient than the classical methods based on automata. We present some performance tests, comparisons with similar approaches, and also introduce a generalization of the algorithm to decide the equivalence of terms of Kleene algebra with tests. The motivation for the work presented in this paper is that of using the libraries developed as trusted frameworks for carrying out certified program verification.

1. Introduction

Formal languages are one of the pillars of computer science. Amongst the several computational models of formal languages, that of *regular expression* is one of the most widely known and used. The notion of regular expressions has its origins in the seminal work of Kleene, where the author introduced them as a specification language for *deterministic finite automata* (DFA) [1]. Nowadays, regular expressions find applications in a wide variety of areas due to their capability of expressing patterns in a succinct and comprehensive way. They abound in technologies deriving from the *World Wide Web*, in text processors, in structured languages such as XML, and are a core element of programming languages like Perl [2] and Esterel [3]. More recently, regular expressions have been successfully applied in the runtime verification of programs [4,5].

In the past years, much attention has been given to the mechanization of *Kleene algebra* (KA) – the algebra of regular expressions – within proof assistants. Formally, a KA is an idempotent semiring together with the Kleene star operator $\cdot^*$, that is characterized axiomatically. J.-C. Filliâtre [6] provided a first formalization of the Kleene theorem for regular languages [1] within the Coq proof assistant [7]. Höfner and Struth [8] investigated the automated reasoning in variants of Kleene algebras with Prover9 and Mace4 [9]. Pereira and Moreira [10] implemented in Coq an abstract specification of *Kleene algebra with tests* (KAT) [11] and the proofs that propositional Hoare logic deduction rules are theorems of KAT. An obvious follow up of that work was to implement a certified procedure for deciding equivalence of KA terms, i.e., regular expressions. A first step was the proof of the correctness of the partial derivative automaton construction from a regular expression [12]. In this paper we describe the mechanization of a decision procedure based on partial derivatives that was

E-mail addresses: nam@ncc.up.pt (N. Moreira), dmrpe@isep.ipp.pt (D. Pereira), desousa@di.ubi.pt (S. Melo de Sousa).

http://dx.doi.org/10.1016/j.jlamp.2014.12.004
2352-2208/