

# **Uma Implementação de Expressões Regulares em Tempo Polinomial**

Juan Pedro Alves Lopes

Orientador

Prof. Paulo Eustáquio Duarte Pinto

Coorientadora

Prof<sup>a</sup>. Maria Alice Silveira de Brito

# Agenda

- Motivação
- Construção do autômato
- Implementação
- Backreferences em NP-difícil
- Conclusão e trabalhos futuros

186.247.10.187 - - GET

/gnews/gnews\_600/gnews\_1374508780437\_1374508780437\_1470.ts

HTTP/1.1 "200" 346672 "-" 0.232 "AppleCoreMedia/1.0.0.10B329 (iPad; U; CPU OS 6\_1\_3 like Mac OS X; pt\_br)" "-" - [MISS] 0.006 ms

201.51.232.169 - - GET

/gnews/gnews\_1000/gnews\_1374508779820\_1374508779820\_1466.ts

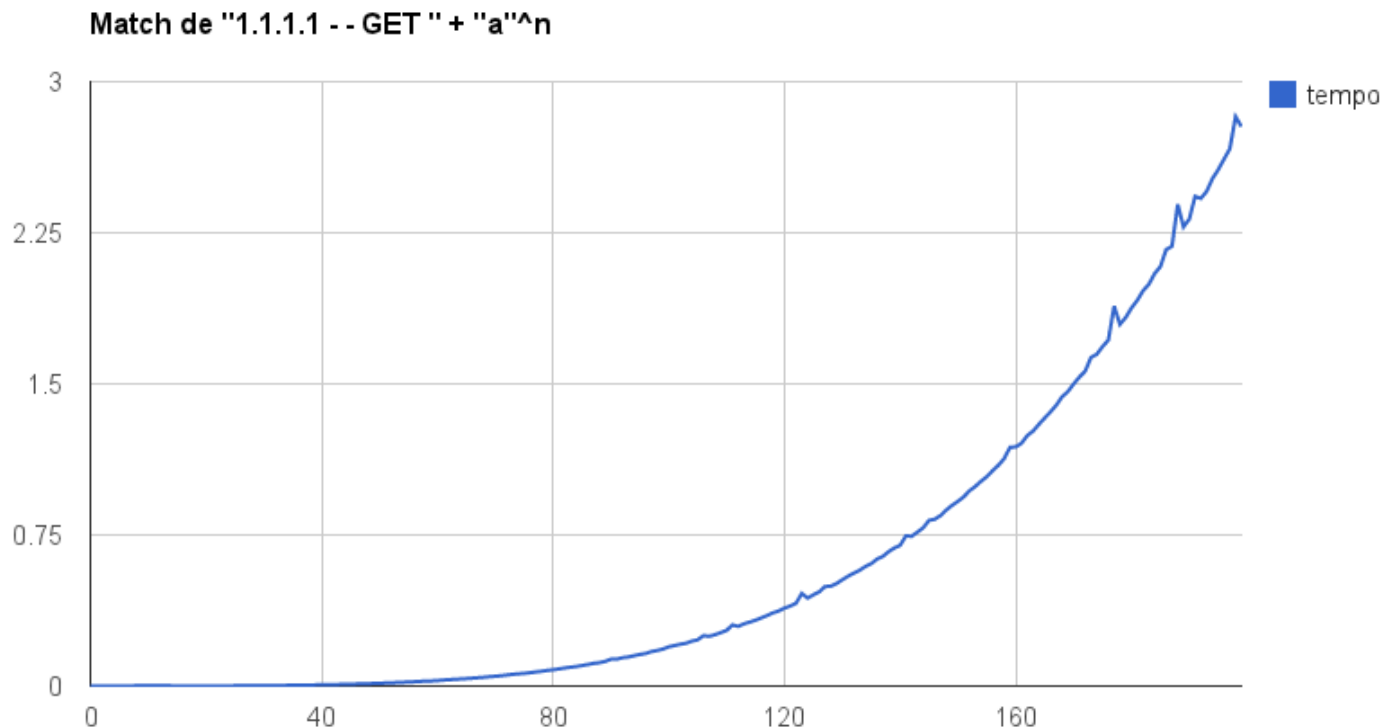
HTTP/1.1 "200" 552156 "http://globotv.globo.com/globo-news/globo-news-ao-vivo/v/globonews-ao-vivo/61910/" 0.480 "Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/28.0.1500.72 Safari/537.36" "-" - [MISS] 0.008 ms

201.7.186.60 - - GET /ber2/ber2\_400/playlist.m3u8 HTTP/1.1 "200" 538

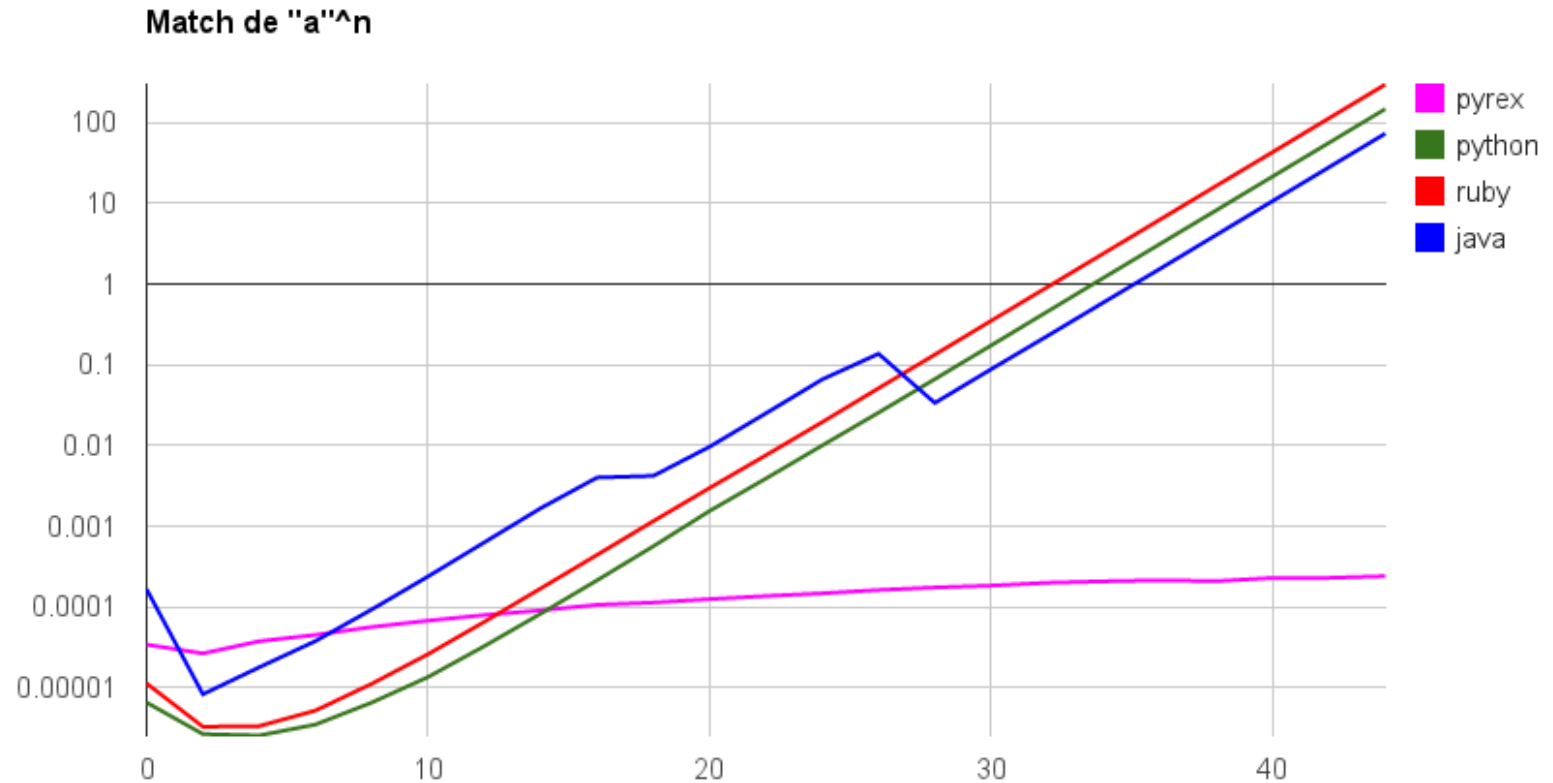
"http://g1.globo.com/jornal-hoje/ao-vivo.html" 0.001 "Mozilla/5.0

(Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/28.0.1500.72 Safari/537.36" "-" - [EXPIRED] 0.001 ms

```
(?<ip>[\d\.]+\s-\s-\sGET\s(?<path>/?(?<stream>[^\|]*)/?[^\.]*.?(?<extension>[^\s\?]*))?(?<querystring>[^\s]*\s[^\s]*\s"(?<status>\d*)"\s(?<body_bytes_sent>[\d]*)\s"(?<http_referer>[^\"]*)"\s[\d\.]+\s"(?<user_agent>[^\"]*)"\s"(?<forwarded_for>[^\"]*)"\s-\s\s[ (?<cache_status>[\w-]*)\]\s(?<upstream_response_time>[^\s]*)\sms.*
```



# $(a?a)+b$



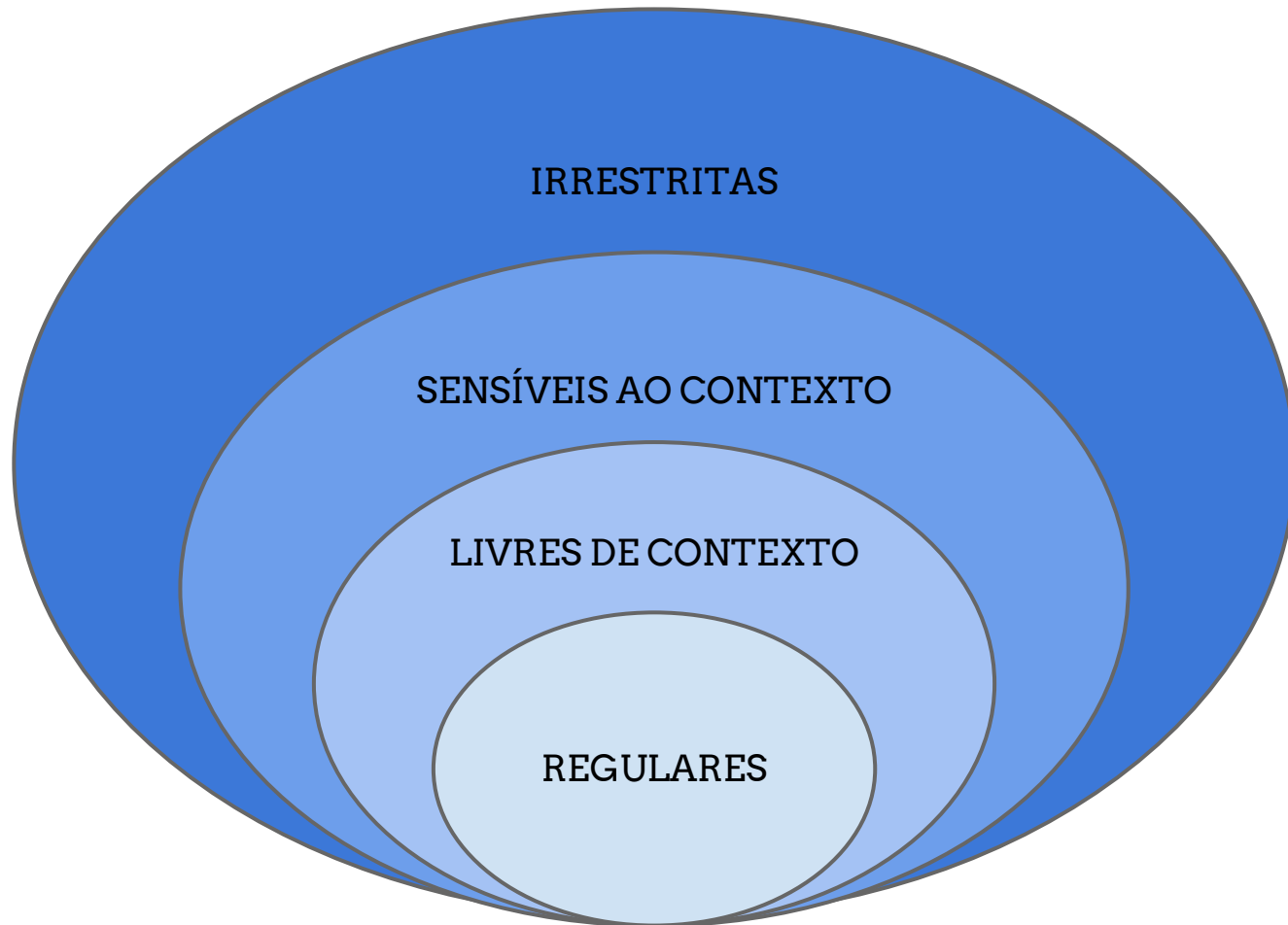
Expressões regulares em  
tempo linear?

SIM e NÃO

# Expressões regulares em tempo linear?

**SIM** e NÃO

# Hierarquia de Chomsky





# Hierarquia de Chomsky

	Gramática	Máquina	Regras
0	Irrestrita	Máquina de Turing	$\alpha \rightarrow \beta$
1	Sensível ao contexto	Autômato linearmente limitado	$\alpha S \beta \rightarrow \alpha \gamma \beta$
2	Livre de contexto	Autômato de pilha	$S \rightarrow \gamma$
3	Regular	Autômato finito	$S \rightarrow \varepsilon$ $S \rightarrow a$ $S \rightarrow aT$

# Gramática Regular

$$S \rightarrow \varepsilon$$

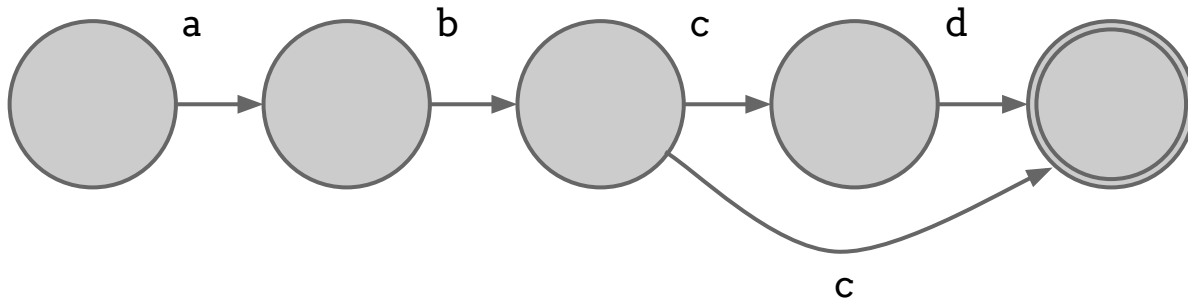
$$S \rightarrow a$$

$$S \rightarrow aT$$

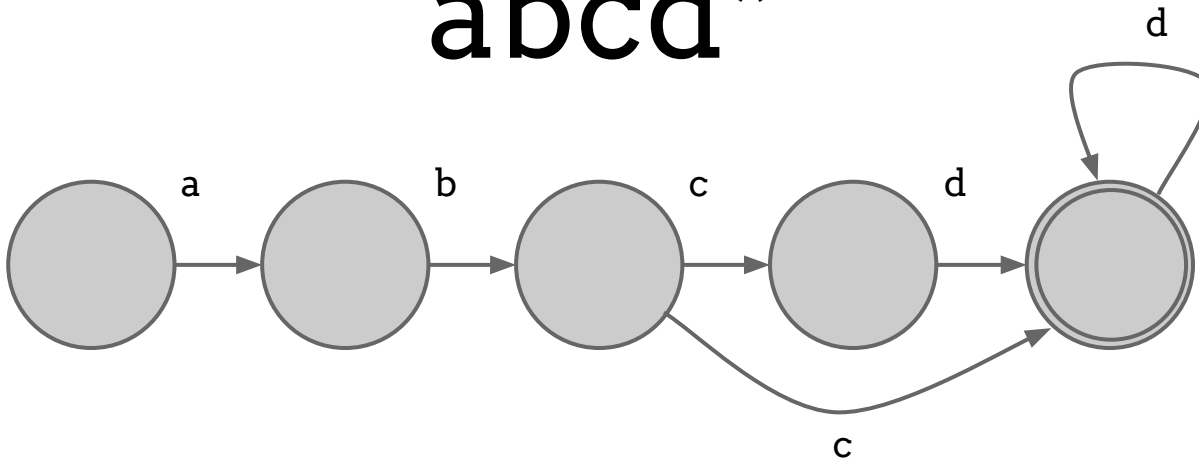
# Expressões Regulares

	Forma	Descrição
ER1	$\emptyset$	Linguagem vazia
ER2	$\varepsilon$	Linguagem contendo apenas cadeia vazia
ER3	$a$	Linguagem contendo apenas cadeia $a$
ER4	$\alpha \beta$	União entre $L_\alpha$ e $L_\beta$
ER5	$\alpha\beta$	Concatenação entre $L_\alpha$ e $L_\beta$
ER6	$\alpha^*$	Fecho Kleene sobre $L_\alpha$

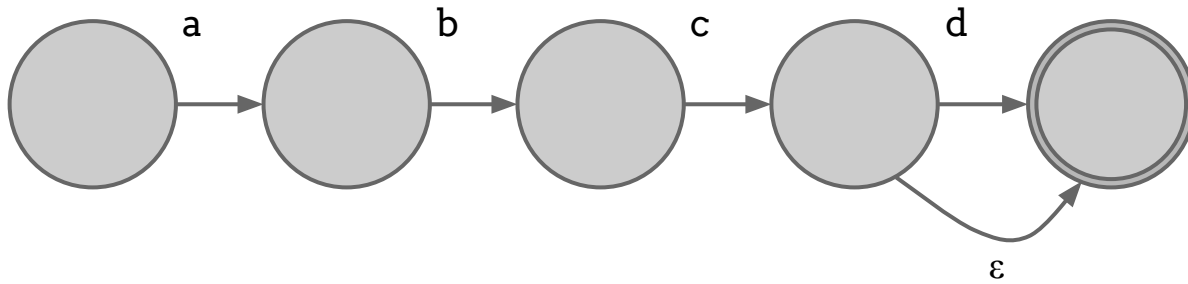
abc?d



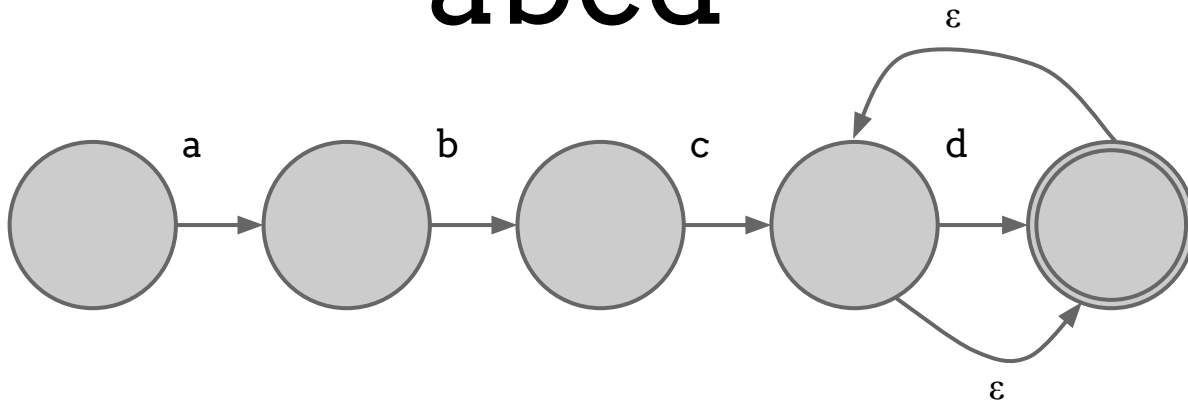
abcd\*



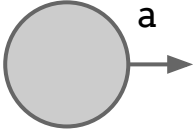
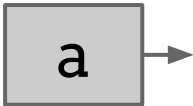
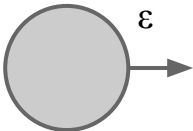
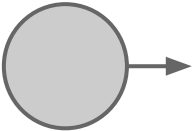
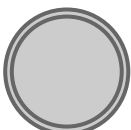
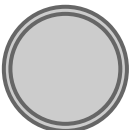
abc?d



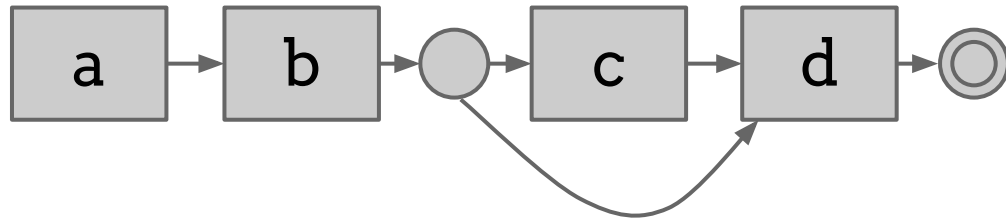
abcd\*



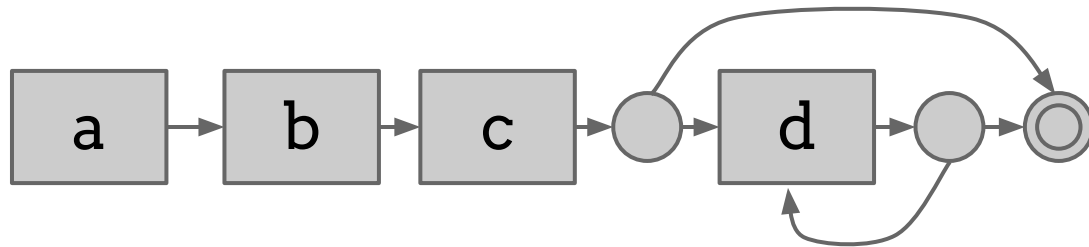
# Notação

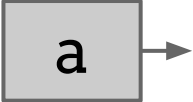
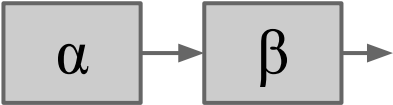
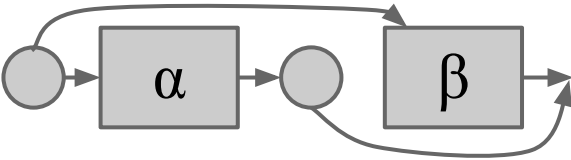
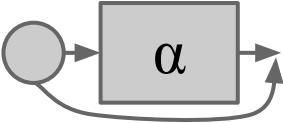
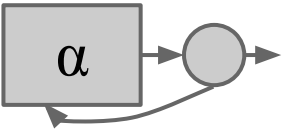
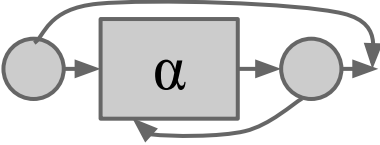
Autômato	Notação	Instrução
		CONSUME
		JUMP
		MATCH

abc?d



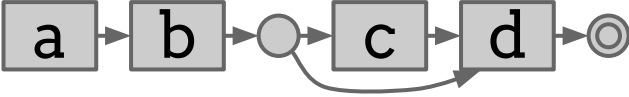
abcd\*



Expressão	Autômato
$a$	
$\alpha\beta$	
$\alpha \beta$	
$\alpha?$	
$\alpha^+$	
$\alpha^*$	



Expressão	Autômato	Array
abc?d		'a', 'b', [1, 2], 'c', 'd'
abcd+		'a', 'b', 'c', 'd', [1, -1]
(a?a)+b		[1, 2], 'a', 'a', [1, -3], 'b'

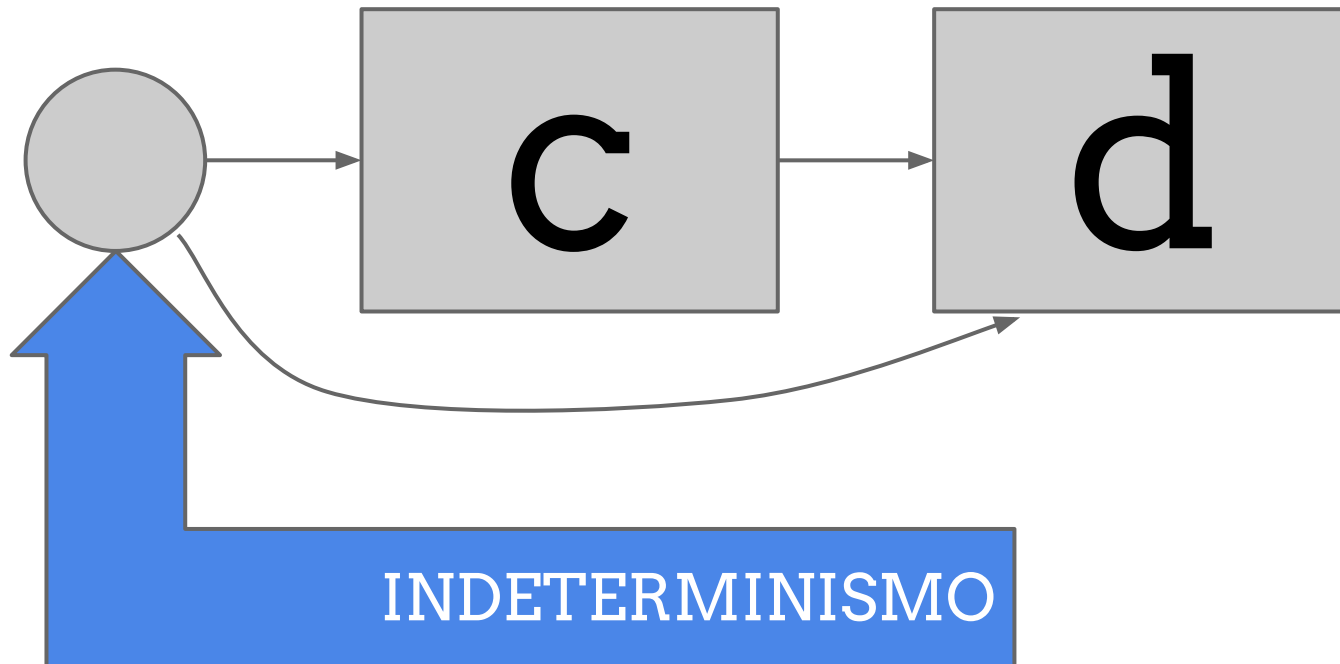
Expressão	Autômato	Array
abc?d		'a', 'b', [1, 2], 'c', 'd'

```

0000: CONSUME 'a'
0001: CONSUME 'b'
0002: JUMP (1, 2)
0003: CONSUME 'c'
0004: CONSUME 'd'
0005: MATCH!

```

# NFA: Autômato Finito Não Determinístico



Expressão	Autômato	Array
$(a?a)+b$	<pre> graph LR     Start(( )) --&gt; A1[a]     Start --&gt; A2[a]     A1 --&gt; A3[a]     A2 --&gt; B[b]     A3 --&gt; B     B --&gt; End((( ))) </pre>	$[1, 2], 'a', 'a', [1, -3], 'b'$

```

0000: JUMP (1, 2)
0001: CONSUME 'a'
0002: CONSUME 'a'
0003: JUMP (1, -3)
0004: CONSUME 'b'
0005: MATCH!

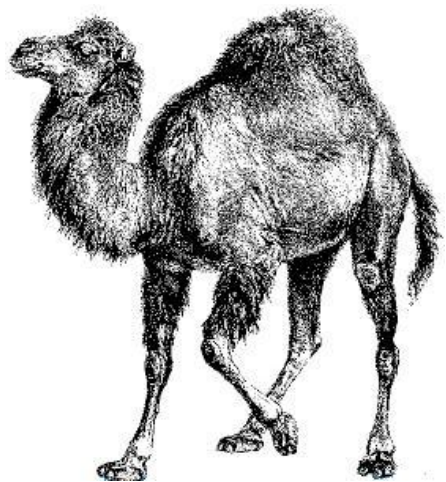
```

# Simular NFA

Método	Vantagem	Desvantagem
Backtracking	Implementação simples	Simulação exponencial
Backtracking com memorização	Implementação simples; complexidade polinomial	Chamadas recursivas; acesso aleatório à entrada
Converter para DFA	Simulação linear	Conversão exponencial
Simulação paralela	Sem chamadas recursivas; acesso sequencial à entrada	Simulação superlinear
Conversão preguiçosa para DFA	Boa localidade de referência; melhor caso linear	Pior caso ainda superlinear

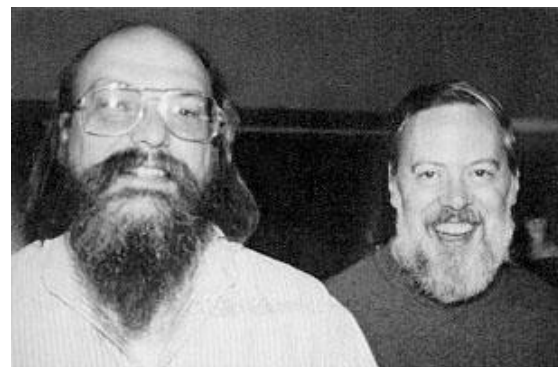
# Simular NFA

Método	Vantagem	Desvantagem
Backtracking	Implementação simples	Simulação exponencial
Backtracking com memorização	Implementação simples; complexidade polinomial	Chamadas recursivas; acesso aleatório à entrada
Converter para DFA	Simulação linear	Conversão exponencial
Simulação paralela	Sem chamadas recursivas; acesso sequencial à entrada	Simulação superlinear
Conversão preguiçosa para DFA	Boa localidade de referência; melhor caso linear	Pior caso ainda superlinear



PCRE

POSIX



# PyRex

```
# -*- coding: utf-8 -*-
from collections import deque
from functools import reduce

def rex(pattern):
    tokens = deque(pattern)

    def walk(chars):
        while tokens and tokens[0] in chars:
            yield tokens.popleft()

    def option():
        e = sequence()
        for token in walk('|'):
            e2 = sequence()
            e = [(1, len(e)+2)] + e + [(len(e2)+1,)] + e2
        return e

    def sequence():
        e = []
        while tokens and tokens[0] not in '|)':
            e += repetition()
        return e

    def repetition():
        e = primary()
        for token in walk('?*+'):
            if token in '+': e = e + [(1, -len(e))]
            if token in '?*': e = [(1, len(e)+1)] + e
        return e

    def primary():
        token = tokens.popleft()
        if token == '.': return [None]
        if token == '(': return [option(), tokens.popleft()][0]
        if token not in '?*+|)': return [token]
        raise Exception('Not expected: "{}".format(token))

    e = option()
    if tokens:
        raise Exception('Not expected: "{}".format(''.join(tokens)))

    return Machine(e)
```

```
class Machine(object):
    def __init__(self, states):
        self.states = states
        self.n = len(states)

    def matcher(self, string):
        A, B, V = list(), list(), [-1]*len(self.states)

        def addnext(start, i, j):
            if j==self.n: return 1
            if V[j] == i: return 0
            V[j] = i

            if isinstance(self.states[j], tuple):
                return sum(addnext(start, i, j+k) for k in self.states[j])

            B.append((start, j))
            return 0

        def key(a): return (a[1]-a[0], -a[0]) if a else (0, 0)

        answer = None
        for i, c in enumerate(string):
            addnext(i, i, 0)
            yield i, answer, B

            A, B = B, A
            del B[:]

        for start, j in A:
            if self.states[j] in (None, c) and addnext(start, i+1, j+1):
                answer = max(answer, (start, i+1), key=key)

        yield len(string), answer, B

    def match(self, string):
        return reduce(lambda answer, s: s[1], self.matcher(string), None)

    def source(self):
        for s in self.states:
            yield ('JUMP ' if isinstance(s, tuple) else 'CONSUME ') + str(s)
        yield 'MATCH!'

    def __repr__(self):
        return '\n'.join('{:04d}: {}'.format(i, s) for i, s in enumerate(self.source()))
```



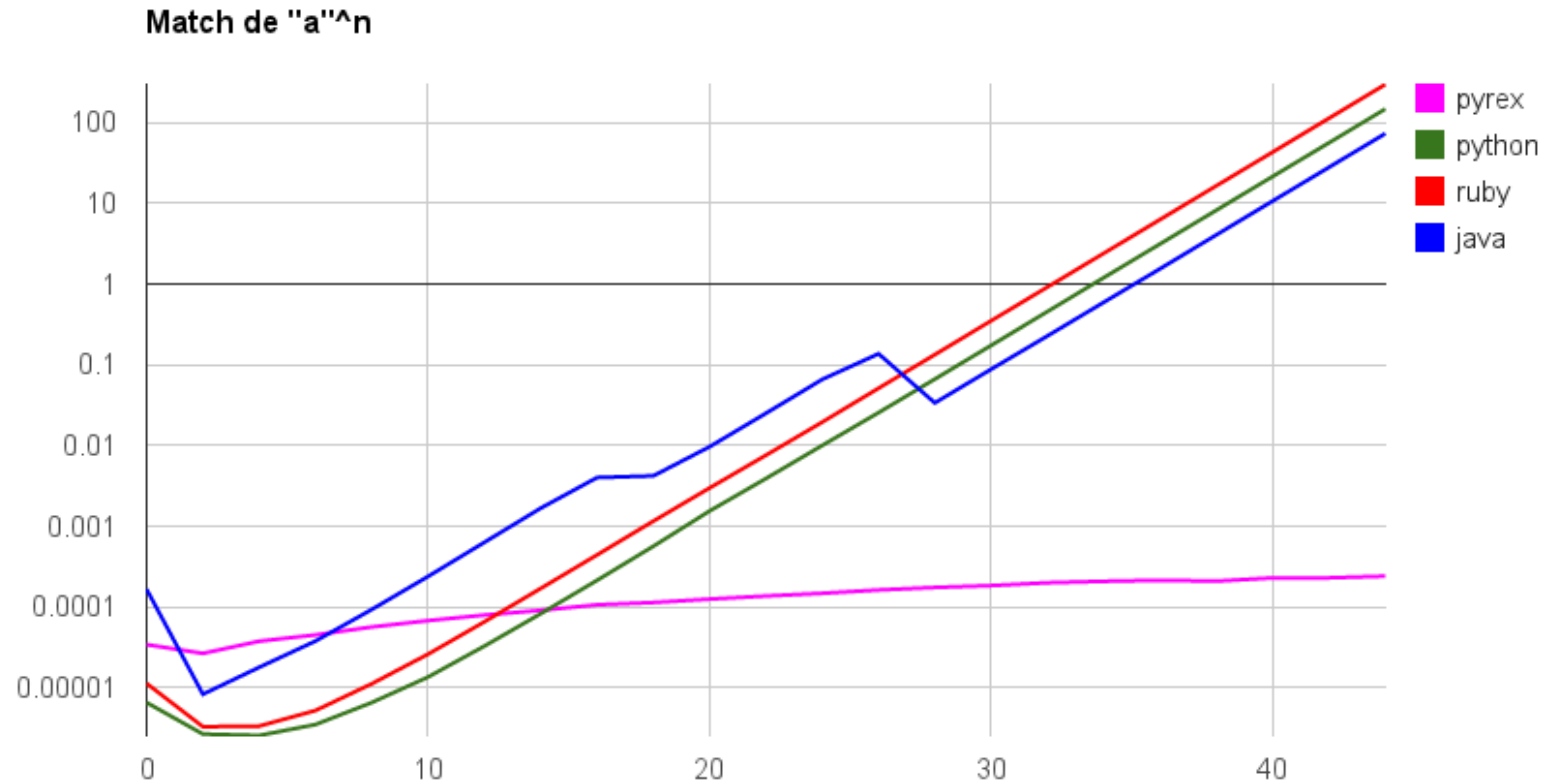
# PyRex Viewer

```
/bin/bash
/bin/bash 74x33
juanlop@juanlop-pc ~/gh/monografia/pyrex $ ./view.py a+b+c+ aabbcc
-----
Matching pattern "a+b+c+" against input "aabbcc"
-----
Best answer: <none>
Input: aabbcc
      ^ (0)

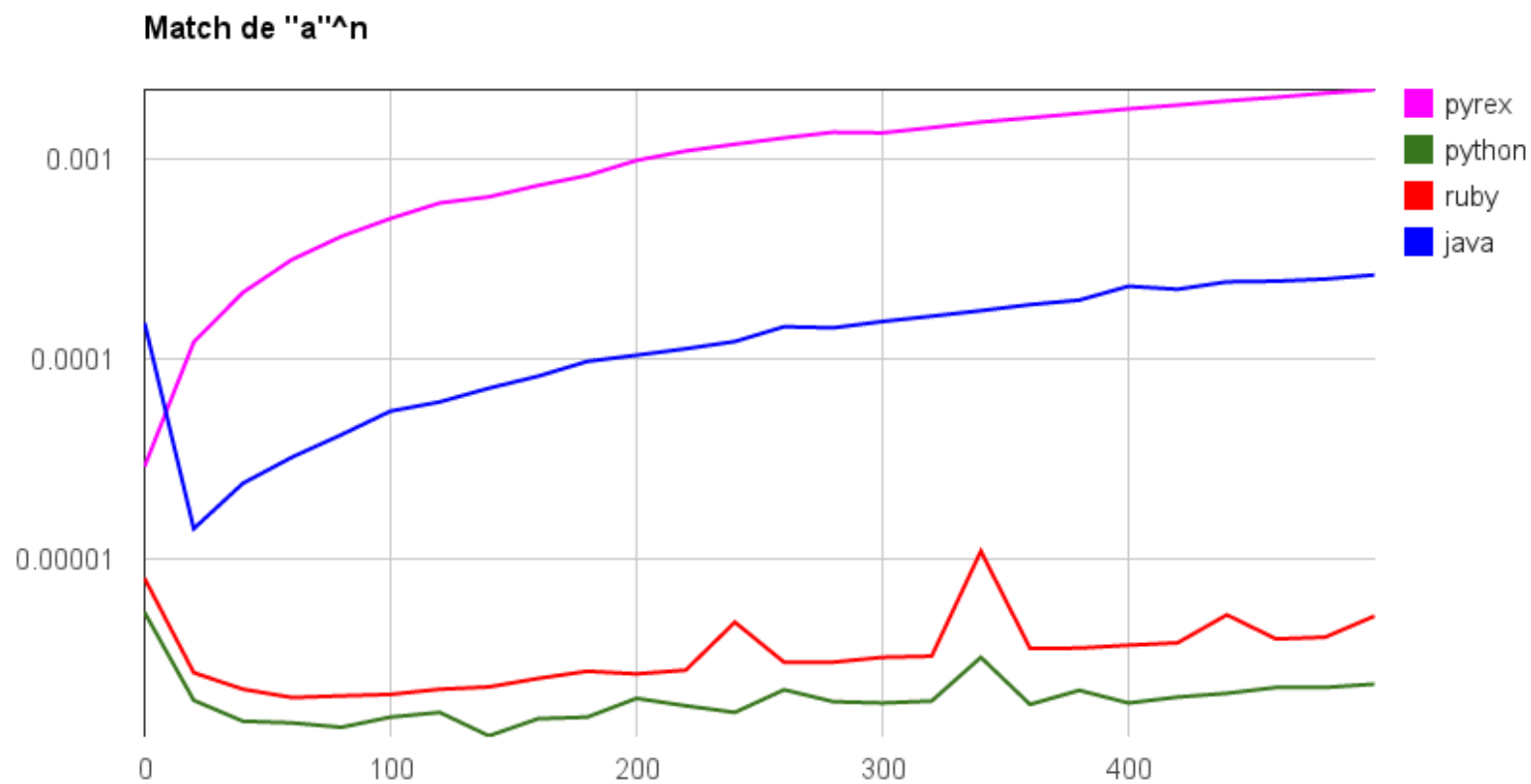
0 >0000: CONSUME a
      0001: JUMP (1, -1)
      0002: CONSUME b
      0003: JUMP (1, -1)
      0004: CONSUME c
      0005: JUMP (1, -1)
      0006: MATCH!
-----
Best answer: <none>
Input: aabbcc
      ^ (1)

0 >0000: CONSUME a
      0001: JUMP (1, -1)
0 >0002: CONSUME b
      0003: JUMP (1, -1)
      0004: CONSUME c
      0005: JUMP (1, -1)
      0006: MATCH!
-----
```

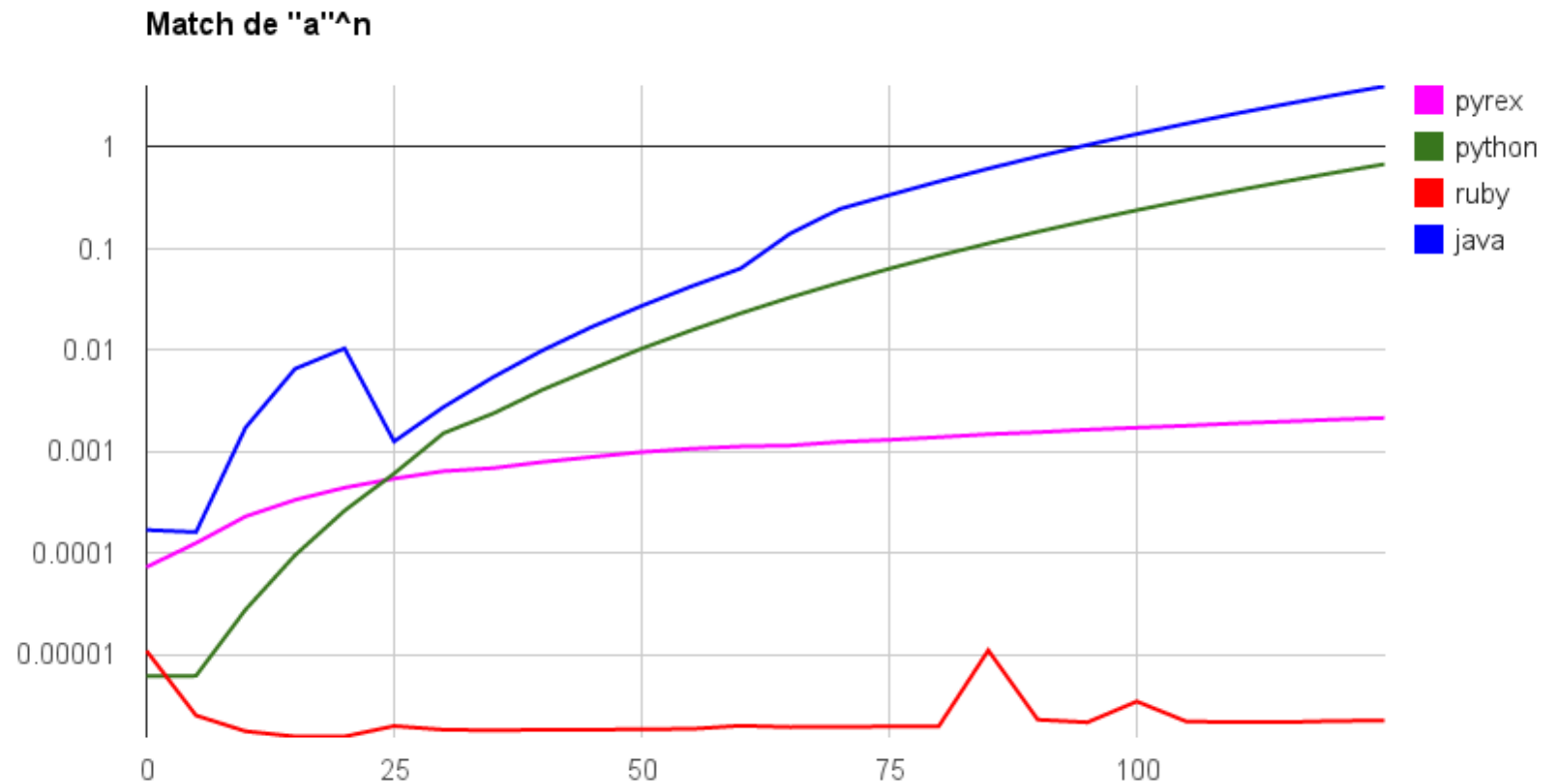
# $(a?a)+b$



# $a^*b$



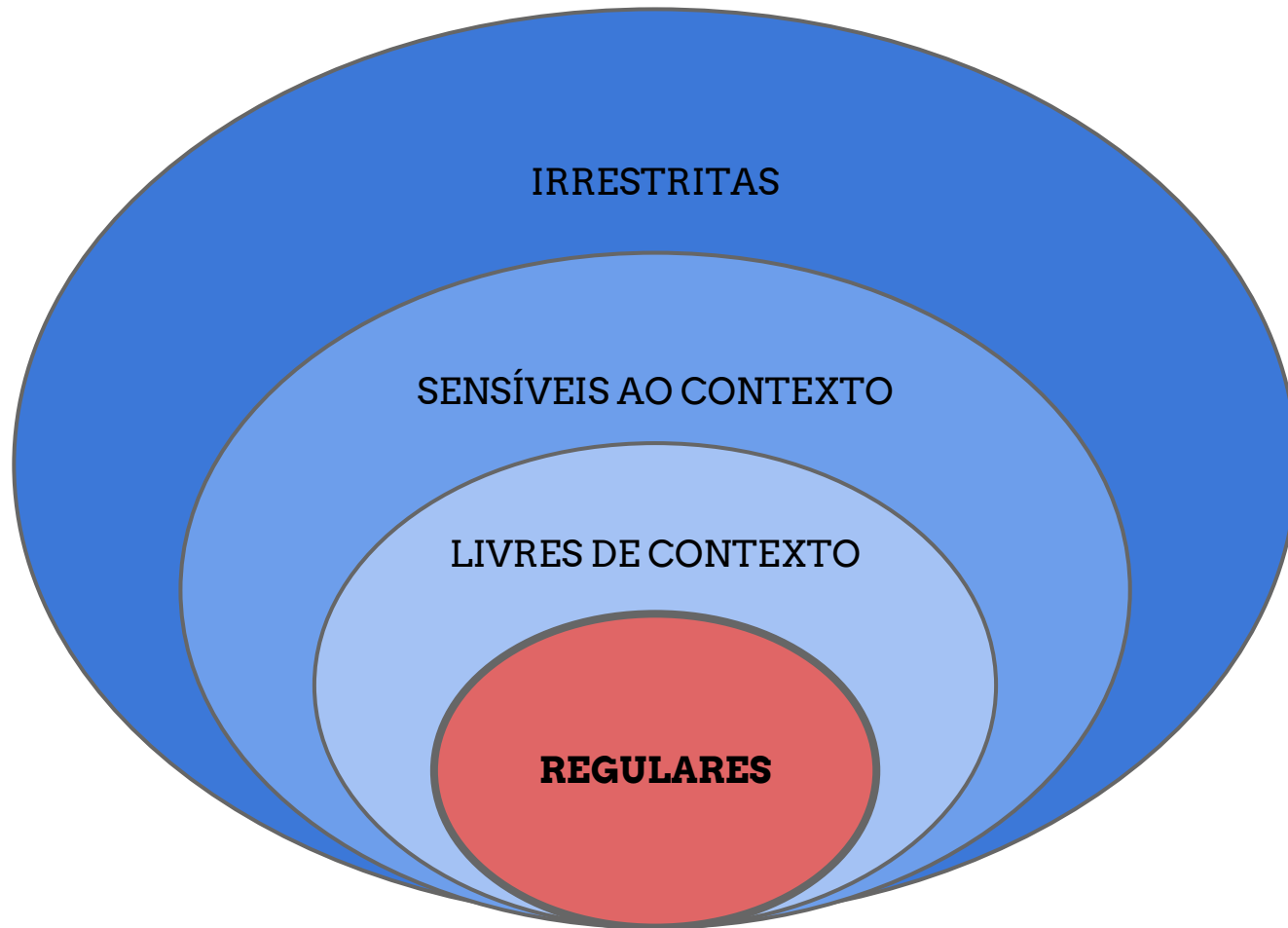
# $a^*a^*a^*a^*a^*b$



# Expressões regulares em tempo linear?

SIM e NÃO

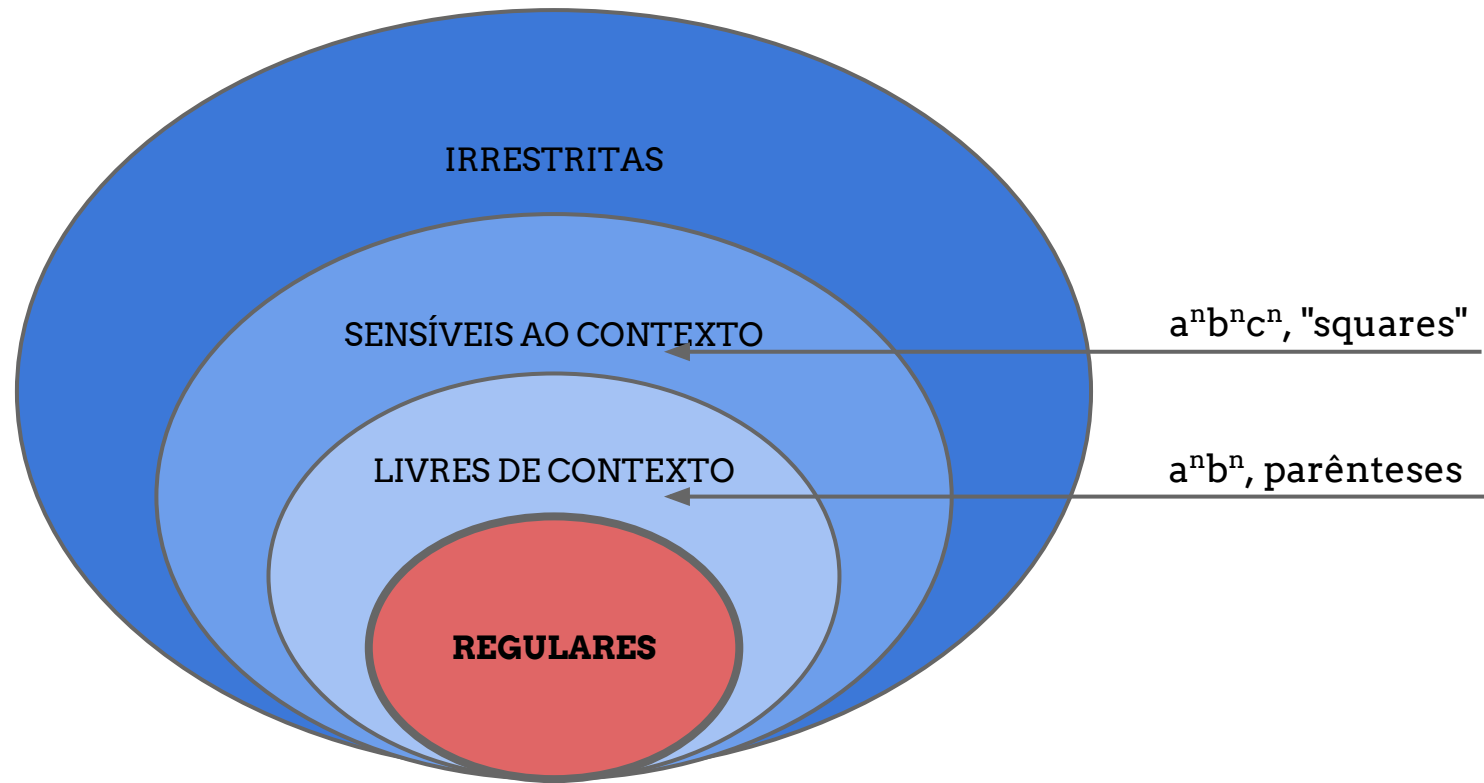
# Linguagens não-regulares



# Linguagens não-regulares

	Exemplos
Parênteses aninhados	$((())(())), (((()))), \dots$
"Squares"	DogDog, CatCat, WikiWiki, ...
$a^n b^n$	ab, aabb, aaabbb, ...

# Linguagens não-regulares





# Na prática...

	Expressão
Parênteses aninhados	$(\backslash((?1)^*\backslash))^1$
"Squares"	$(.*)\backslash 1$
$a^n b^n$	$(a(?1)?b)^1$

<sup>1</sup> sintaxe do Perl para recursive capture buffers

Match de regexps com  
backreferences é um  
problema **NP-difícil**.

# 3-SAT (NP-completo)

$$\begin{aligned} &(\neg x_1 \vee x_2 \vee x_4) \wedge \\ &(x_1 \vee \neg x_3 \vee x_4) \wedge \\ &(x_1 \vee \neg x_2 \vee \neg x_4) \wedge \\ &(x_2 \vee \neg x_3 \vee \neg x_4) \wedge \\ &(\neg x_1 \vee x_3 \vee \neg x_4) \wedge \\ &(x_1 \vee x_2 \vee x_3) \wedge \\ &(\neg x_1 \vee \neg x_2 \vee \neg x_3) \end{aligned}$$

3-SAT com Regexp	
variáveis $x_1, x_2, x_3$ e $x_4$	$^(x?)(x?)(x?)(x?).*;$
$(\neg x_1 \vee x_2 \vee x_4) \wedge$	$(?:x\1 \2 \4),$
$(x_1 \vee \neg x_3 \vee x_4) \wedge$	$(?:\1 x\3 \4),$
$(x_1 \vee \neg x_2 \vee \neg x_4) \wedge$	$(?:\1 x\2 x\4),$
$(x_2 \vee \neg x_3 \vee \neg x_4) \wedge$	$(?:\2 x\3 x\4),$
$(\neg x_1 \vee x_3 \vee \neg x_4) \wedge$	$(?:x\1 \3 x\4),$
$(x_1 \vee x_2 \vee x_3) \wedge$	$(?:\1 \2 \3),$
$(\neg x_1 \vee \neg x_2 \vee \neg x_3)$	$(?:x\1 x\2 x\3),\$$
	match 'xxxx;x,x,x,x,x,x,x,'

Grupos capturados: ['x', 'x', "", ""]

# Conclusão

- Match de expressões regulares com backreferences é um problema NP-difícil.
- É possível reconhecer cadeias em linguagens regulares com algoritmo polinomial.
- Implementações como a da JDK podem levar a tempos de execução exponenciais mesmo com expressões que denotam linguagens regulares.
- O uso de expressões regulares modernas pode levar a falhas de segurança em aplicações online.

# Trabalhos futuros

- Implementação em outras linguagens
- Implementar funcionalidades
  - Classes de caracteres
  - Complemento
  - Captura de grupos
  - Suporte a Unicode
  - Repetições
  - Zero-width assertions
- Mapear o impacto em tempo de execução das funcionalidades em diversas implementações