# Investigación en teoría de autómatas

Clase 31

IIC2223 / IIC2224

Prof. Cristian Riveros

# ¿para qué sirve teoría de autómatas?

1. Fundamentos de la computación
2. Herramientas de software como:
   - Regex
   - Compiladores
   - Bases de datos
   - Verificación automática de software
3. Investigación

# Outline

Motivación

REQL

Detras de REmatch

# Encontrar todos los matches

**Problema:** Dado un patrón y un documento, encontrar todos los matches.

| t | h | a | t | h | a | t | h | a | t | h | a | t | h | a | t |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

*"Encontrar todas las apariciones de la palabra* **that***"*

# Encontrar todos los matches

**Problema:** Dado un patrón y un documento, encontrar todos los matches.


*"Encontrar todas las apariciones de la palabra* **that***"*

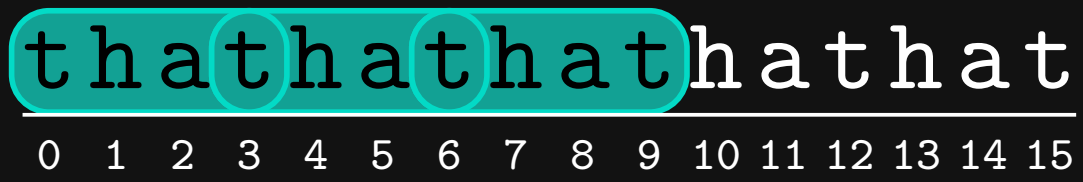
# Encontrar todos los matches

**Problema:** Dado un patrón y un documento, encontrar todos los matches.


*"Encontrar todas las apariciones de la palabra* **that***"*

# Encontrar todos los matches

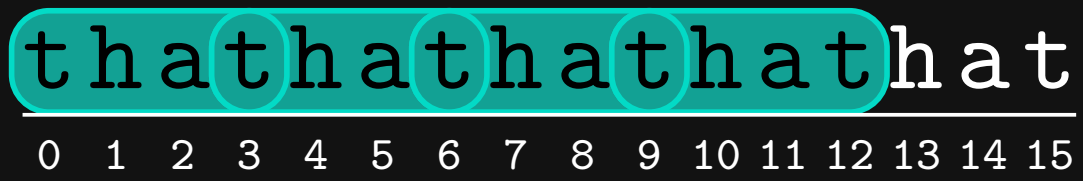
**Problema:** Dado un patrón y un documento, encontrar todos los matches.




*"Encontrar todas las apariciones de la palabra* **that***"*

# Encontrar todos los matches

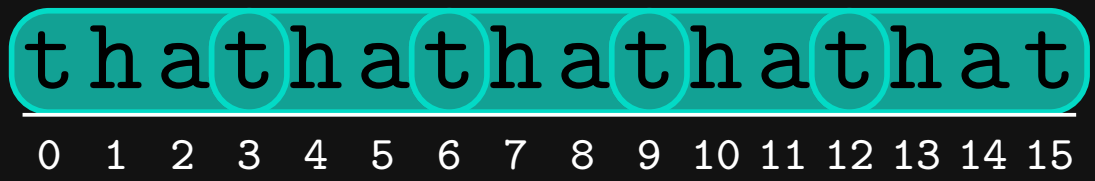
**Problema:** Dado un patrón y un documento, encontrar todos los matches.




*"Encontrar todas las apariciones de la palabra* **that***"*

# Encontrar todos los matches

**Problema:** Dado un patrón y un documento, encontrar todos los matches.




*"Encontrar todas las apariciones de la palabra* **that***"*

# Encontrar todos los matches con regex

**Problema:** Dado un patrón y un documento, encontrar todos los matches.

| t | h | a | t | h | a | t | h | a | t | h | a | t | h | a | t |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Regex := $\underbrace{\texttt{(that)}}_{\text{paréntesis de capture}}$

# Encontrar todos los matches con regex

**Problema:** Dado un patrón y un documento, encontrar todos los matches.


Regex := (that)

paréntesis de capture

# How to use regex to find all overlapping matches

Asked 13 years, 7 months ago Modified 10 days ago Viewed 113k times

170

I'm trying to find every 10 digit series of numbers within a larger series of numbers using re in Python 2.6.

I'm easily able to grab no overlapping matches, but I want every match in the number series. Eg.

in "12345678912345678 9"

I should get the following list:

```
[1234567891,2345678912,3456789123,4567891234,5678912345,6789123456,7891234567,8912
```

I've found references to a "lookahead", but the examples I've seen only show pairs of numbers rather than larger groupings and I haven't been able to convert them beyond the two digits.

python regex

Share Edit Follow

edited Sep 4, 2023 at 9:20

InSync

9,877 ● 4 ● 13 ● 49

asked Apr 11, 2011 at 4:41

danspants

3,377 ● 4 ● 29 ● 33

# How to use regex to find all overlapping matches

Asked 13 years, 7 months ago Modified 10 days ago Viewed 113k times

# Overlapping matches in Regex

Asked 15 years, 11 months ago Modified 2 years, 11 months ago Viewed 16k times

51

I can't seem to find an answer to this problem, and I'm wondering if one exists. Simplified example:

Consider a string "nnnn", where I want to find all matches of "nn" - but also those that overlap with each other. So the regex would provide the following 3 matches:

1. **nn**nn
2. n**nn**n
3. nn**nn**

I realize this is not exactly what regexes are meant for, but walking the string and parsing this manually seems like an awful lot of code, considering that in reality the matches would have to be done using a pattern, not a literal string.

c# regex overlap

# How to use regex to find all overlapping matches

Asked 13 years, 7 months ago Modified 10 days ago Viewed 113k times

# Overlapping matches in Regex

Asked 15 years, 11 months ago Modified 2 years, 11 months ago Viewed 16k times

## Regex Matches Overlapping Patterns

Learn

BR **brasheva** Brasheva Jul 2020

Hi,
How could I find overlapping patterns using Matches activity?
For example, I have the string “1-ABCD-AP.1234567” and I have 2 patterns which I have listed here: “1-[A-Z0-9-]{6,8}|AP.\d{6,30}”.
I need both “1-ABCD-AP” which matches the first pattern and “AP.1234567” which matched the second pattern.
But the Match Activity returns only “1-ABCD-AP” as the letters “AP” overlap in the 2 patterns.
I know I can do this using For Each activity for each pattern but I have multiple strings and multiple patterns which means that that I have to use For Each activity for each pattern within another For Each activity for each string, therefore I am wondering if there is a more simple solution.
Thank you

Jul 2020

1 / 4
Jul 2020

Jul 2020

# How to use regex to find all overlapping matches

Asked 13 years, 7 months ago Modified 10 days ago Viewed 113k times

# Overlapping matches in Regex

Asked 15 years, 11 months ago Modified 2 years, 11 months ago Viewed 16k times

## Regex Matches Overlapping Patterns

Learn

brasheva Brasheva Jul 2020 Jul 2020

Forums › Other Sciences › **Programming and Computer Science** ›

# How can I modify my regex to allow for overlapping matches in Python?

Python · TylerH · May 11, 2013 · Overlapping Python

In summary, the conversation discusses creating a function to generate a regex that can be used in algebraic manipulations. The regex is meant to match specific patterns, but the current version is not as flexible as desired. The speaker is considering using lex to handle the task instead.

**May 11, 2013** #1

**TylerH** :729

# ¿Por qué regex no encuentra todos los matches?

**Problema:** Dado un patrón y un documento, encontrar todos los matches.

| t | h | a | t | h | a | t | h | a | t | h | a | t | h | a | t |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Regex := $\underbrace{\texttt{(that)}}_{\text{paréntesis de capture}}$

Semántica **leftmost-longest**

# ¿Por qué regex no encuentra todos los matches?

**Problema:** Dado un patrón y un documento, encontrar todos los matches.


Regex := (that)

paréntesis de capture

Semántica **leftmost-longest**

# ¿Por qué regex no encuentra todos los matches?

**Problema:** Dado un patrón y un documento, encontrar todos los matches.


Regex := (that)

paréntesis de capture

Semántica **leftmost-longest**

# ¿Por qué regex no encuentra todos los matches?

**Problema:** Dado un patrón y un documento, encontrar todos los matches.


Regex := (that)

paréntesis de capture

Semántica **leftmost-longest**

# ¿Por qué regex no encuentra todos los matches?

**Problema:** Dado un patrón y un documento, encontrar todos los matches.


Regex := (that)

paréntesis de capture

Semántica **leftmost-longest**

# ¿Por qué regex no encuentra todos los matches?

**Problema:** Dado un patrón y un documento, encontrar todos los matches.


Regex := (that)

paréntesis de capture

Semántica **leftmost-longest**

# ¿Por qué regex no encuentra todos los matches?

**Problema:** Dado un patrón y un documento, encontrar todos los matches.

| t | h | a | t | h | a | t | h | a | t | h | a | t | h | a | t |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

$$\texttt{Regex} := \underbrace{\texttt{(that)}}_{\text{paréntesis de capture}}$$

Semántica **leftmost-longest**

# {REmatch}

**REmatch** es un motor de regex que **siempre encuentra todos los matches**.

1. Basado en una nueva teoría y evaluación de regex con capture.
2. Open source e implementado en C++.
3. Librería disponible en C++, Python, y Javascript.

Pueden probarlo online en **www.rematch.cl**

REQL Query

FINISHED (FOUND 5)

```
!x{that}
```

MULTI

Document

```
1  thathathathathat
```

Matches

Match 0

x (0-4) that

Match 1

x (3-7) that

Match 2

x (6-10) that

Match 3

x (9-13) that

Match 4

x (12-16) that

# Outline

Motivación

REQL

Detras de REmatch

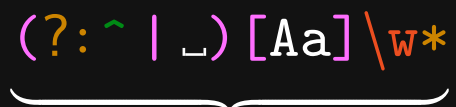
# **REQL**: RegEx Query Language

| | | | |
|---|---|---|---|
| e | := | a | (carácter en UTF-8) |
| | \| | . | (cualquier carácter) |
| | \| | [w] | (character set) |
| | \| | [^w] | (set negado) |
| | \| | ^ | (comienzo de documento) |
| | \| | $ | (término de documento) |
| | \| | ee | (concatenación) |
| | \| | e\|e | (alternación) |
| | \| | e? | (opcional) |
| | \| | e* | (cero o más) |
| | \| | e+ | (uno o más) |
| | \| | e{n,m} | (al menos n y a los más m ) |
| | \| | (e) | (paréntesis normales) |
| | \| | !x{e} | (variables de captura) |

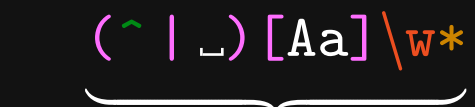
# REQL: su semántica a partir de ejemplos


*"Encontrar todas las palabras que* **empiezan** *con una* **'a'***."*

**Sin** capturar matches:




regex




REQL

REQL es equivalente a regex cuando **no hay que capturar**

. . . pero se pueden usar los paréntesis libremente.

# REQL: su semántica a partir de ejemplos

```
The ant is an amazing architect.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

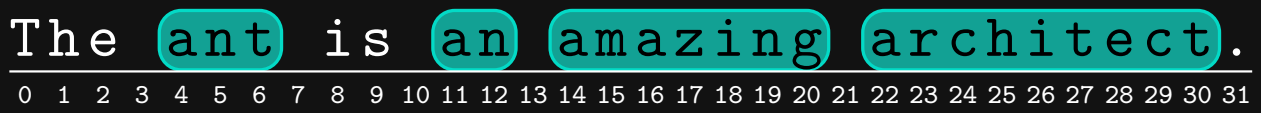
*"Encontrar todas las palabras que* **empiezan** *con una* **'a'***."*

**Con** capturar matches:

```
(?:^ | )([Aa]\w*)
```

regex

# REQL: su semántica a partir de ejemplos




*"Encontrar todas las palabras que* **empiezan** *con una* **'a'***."*

**Con** capturar matches:

```
(?:^ | )([Aa]\w*)
```
regex

```
(^ | )!word{[Aa]\w*}
```
REQL

REQL y regex solo difieren en la **sintaxis y semántica de las capturas**

. . . especialmente en la semántica.

REQL Query

FINISHED (FOUND 21)

```
(^|·)!word{[Aa]\w*}
```

~~MULTI~~

Document

```
1 The ant is an amazing architect.
```

Matches

Match 0

word (4-5) a

Match 1

word (4-6) an

Match 2

word (4-7) ant

Match 3

word (11-12) a

Match 4

word (11-13) an

Match 5

word (14-15) a

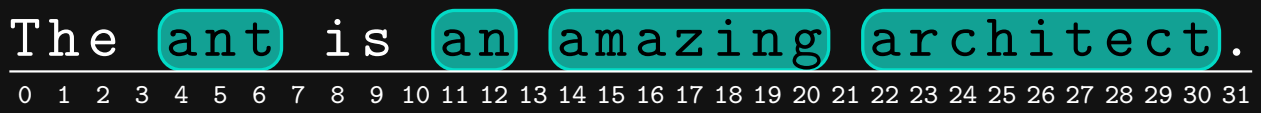
Match 6

word (14-16) am

Match 7

‹ 1 ›

# REQL: su semántica a partir de ejemplos



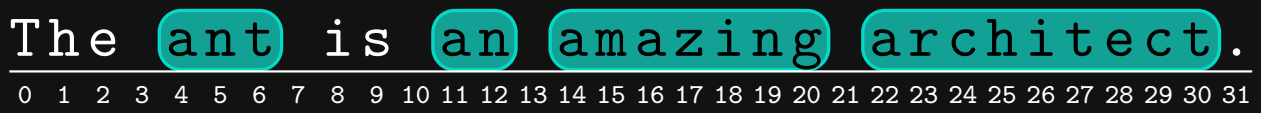

*"Encontrar todas las palabras que* **empiezan** *con una* **'a'***."*

**Con** capturar matches:

```
(?:^ | ␣)([Aa]\w*)
```
regex

```
(^ | ␣)!word{[Aa]\w*}
```
REQL

# REQL: su semántica a partir de ejemplos




*"Encontrar todas las palabras que* **empiezan** *con una* **'a'***."*

**Con** capturar matches:

```
(?:^ | )([Aa]\w*)
```
regex

```
(^ | )!word{[Aa]\w*}[ .]
```
REQL

En REQL los usuarios deben ser más precisos al definir patrones

# REQL: su semántica a partir de ejemplos (2)

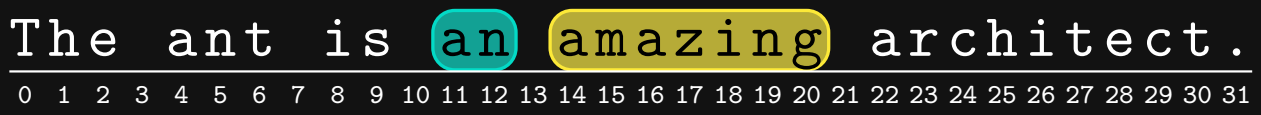
```
The ant is an amazing architect.
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

*"Encontrar todos los* **pares de palabras consecutivas** *(2-gram) donde cada palabra* **empieza** *con una* **'a'***."*

```
(^ | ␣)!w1{[Aa]\w*}␣!w2{[Aa]\w*}[␣.]
```

REQL

# REQL: su semántica a partir de ejemplos (2)




*"Encontrar todos los* **pares de palabras consecutivas** *(2-gram) donde cada palabra* **empieza** *con una* **'a'***."*

```
(^ | ␣)!w1{[Aa]\w*}␣!w2{[Aa]\w*}[␣.]
```

REQL

# REQL: su semántica a partir de ejemplos (2)


*"Encontrar todos los* **pares de palabras consecutivas** *(2-gram) donde cada palabra* **empieza** *con una* **'a'***."*

```
(^ | ␣)!w1{[Aa]\w*}␣!w2{[Aa]\w*}[␣.]
```

REQL

```
?
```

regex

¿cómo podemos escribir esta consulta **en regex**?

# REQL: su semántica a partir de ejemplos (2)


*"Encontrar todos los* **pares de palabras consecutivas** *(2-gram) donde cada palabra* **empieza** *con una* **'a'***."*


REmatch **no necesita** estos operadores especiales con lookahead

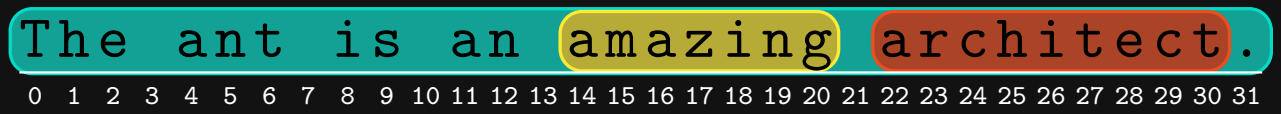
# REQL: su semántica a partir de ejemplos (3)

| The | ant | is | an | amazing | architect. |
|---|---|---|---|---|---|
| 0 1 2 3 | 4 5 6 7 | 8 9 | 10 11 | 12 13 14 15 16 17 18 | 19 20 21 22 23 24 25 26 27 28 29 30 31 |

*"Encontrar todos los* **pares de palabras en una misma oración**
*donde cada palabra* **empieza** *con una* **'a'**
*y, adicionalmente, la* **oración** *donde aparecen."*

```
(^ |\.␣)!sent{([^.]*␣)?
        !w1{[Aa]\w*}␣([^.]*␣)?!w2{[Aa]\w*}
                                (␣[^.]*)?\.}
```

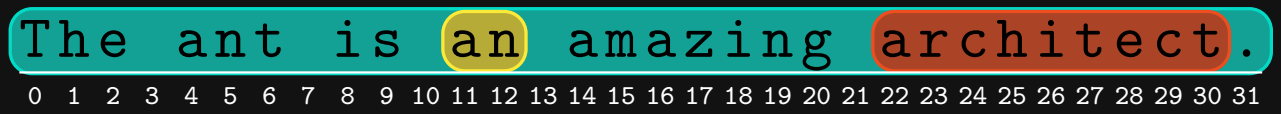
# REQL: su semántica a partir de ejemplos (3)




*"Encontrar todos los* **pares de palabras en una misma oración** *donde cada palabra* **empieza** *con una* **'a'** *y, adicionalmente, la* **oración** *donde aparecen."*

```
(^ |\.␣)!sent{([^.]*␣)?
        !w1{[Aa]\w*}␣([^.]*␣)?!w2{[Aa]\w*}
                                (␣[^.]*)?\.}
```

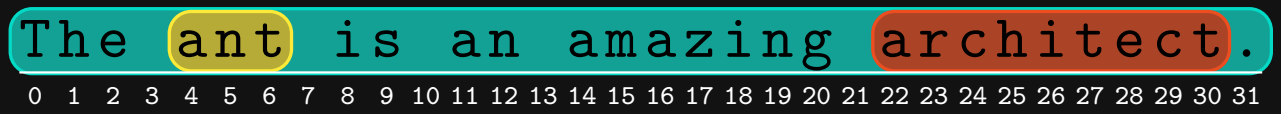
# REQL: su semántica a partir de ejemplos (3)




*"Encontrar todos los* **pares de palabras en una misma oración** *donde cada palabra* **empieza** *con una* **'a'** *y, adicionalmente, la* **oración** *donde aparecen."*

```
(^ |\.␣)!sent{([^.]*␣)?
        !w1{[Aa]\w*}␣([^.]*␣)?!w2{[Aa]\w*}
                                (␣[^.]*)?\.}
```

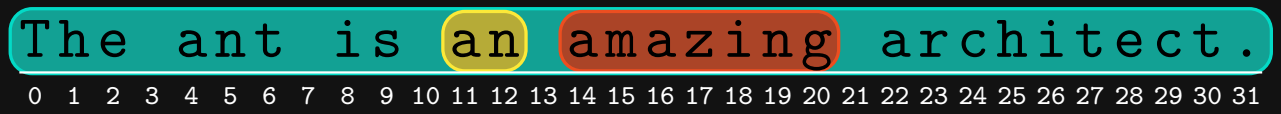
# REQL: su semántica a partir de ejemplos (3)




*"Encontrar todos los* **pares de palabras en una misma oración** *donde cada palabra* **empieza** *con una* **'a'** *y, adicionalmente, la* **oración** *donde aparecen."*

```
(^ |\.␣)!sent{([^.]*␣)?
        !w1{[Aa]\w*}␣([^.]*␣)?!w2{[Aa]\w*}
                                (␣[^.]*)?\.}
```

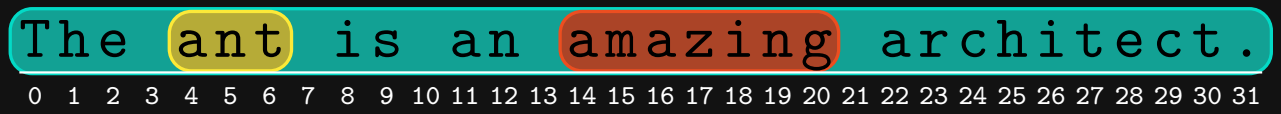
# REQL: su semántica a partir de ejemplos (3)




*"Encontrar todos los* **pares de palabras en una misma oración** *donde cada palabra* **empieza** *con una* **'a'** *y, adicionalmente, la* **oración** *donde aparecen."*

```
(^ |\.␣)!sent{([^.]*␣)?
        !w1{[Aa]\w*}␣([^.]*␣)?!w2{[Aa]\w*}
                                (␣[^.]*)?\.}
```

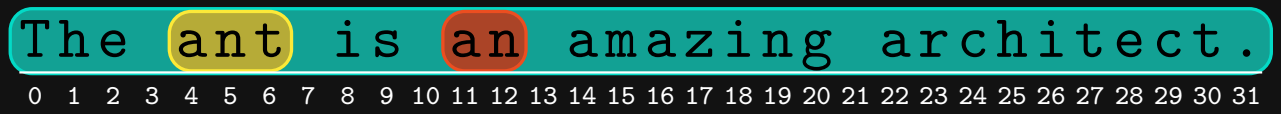
# REQL: su semántica a partir de ejemplos (3)




*"Encontrar todos los* **pares de palabras en una misma oración** *donde cada palabra* **empieza** *con una* **'a'** *y, adicionalmente, la* **oración** *donde aparecen."*

```
(^ |\.␣)!sent{([^.]*␣)?
        !w1{[Aa]\w*}␣([^.]*␣)?!w2{[Aa]\w*}
                                (␣[^.]*)?\.}
```

# REQL: su semántica a partir de ejemplos (3)

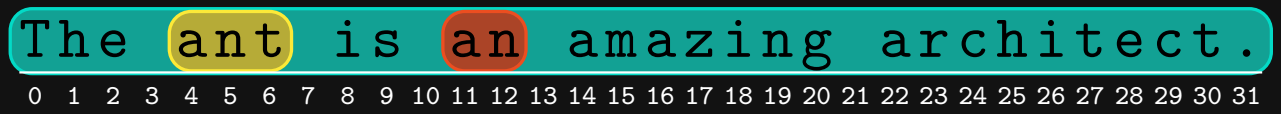



*"Encontrar todos los* **pares de palabras en una misma oración** *donde cada palabra* **empieza** *con una* **'a'** *y, adicionalmente, la* **oración** *donde aparecen."*

```
(^ |\.␣)!sent{([^.]*␣)?
        !w1{[Aa]\w*}␣([^.]*␣)?!w2{[Aa]\w*}
                                (␣[^.]*)?\.}
```

Regex?

# REQL: su semántica a partir de ejemplos (3)




*"Encontrar todos los* **pares de palabras en una misma oración** *donde cada palabra* **empieza** *con una* **'a'** *y, adicionalmente, la* **oración** *donde aparecen."*

```
(^ |\.␣)!sent{([^.]*␣)?
        !w1{[Aa]\w*}␣([^.]*␣)?!w2{[Aa]\w*}
                                (␣[^.]*)?\.}
```

¡**No es posible** con regex o cualquier operador de lookahead!

# Outline

Motivación

REQL

Detras de REmatch

# Document Spanners: A Formal Approach to Information Extraction

RONALD FAGIN, BENNY KIMELFELD,and FREDERICK REISS, IBM Research – Almaden
STIJN VANSUMMEREN, Université Libre de Bruxelles (ULB)



An intrinsic part of information extraction is the creation and manipulation of relations extracted from text. In this article, we develop a foundational framework where the central construct is what we call a *document spanner* (or just *spanner* for short). A spanner maps an input string into a relation over the spans (intervals specified by bounding indices) of the string. The focus of this article is on the representation of spanners. Conceptually, there are two kinds of such representations. Spanners defined in a *primitive* representation extract relations directly from the input string; those defined in an *algebra* apply algebraic operations to the primitively represented spanners. This framework is driven by SystemT, an IBM commercial product for text analysis, where the primitive representation is that of regular expressions with capture variables.

We define additional types of primitive spanner representations by means of two kinds of automata that assign spans to variables. We prove that the first kind has the same expressive power as regular expressions with capture variables; the second kind expresses precisely the algebra of the *regular* spanners—the closure of the first kind under standard relational operators. The *core* spanners extend the regular ones by string-equality selection (an extension used in SystemT). We give some fundamental results on the expressiveness of regular and core spanners. As an example, we prove that regular spanners are closed under difference (and complement), but core spanners are not. Finally, we establish connections with related notions in the literature.

Categories and Subject Descriptors: F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata (e.g., finite, push-down, resource-bounded), relations between models*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Algebraic language theory, classes defined by grammars or automata (e.g., context-free languages, regular sets, recursive sets), operations on languages*; H.2.1 [**Database Management**]: Logical Design—*Data models*; H.2.4 [**Database Management**]: Systems—

# Investigación en document spanners

Ronald Fagin, Benny Kimelfeld, Frederick Reiss, Stijn Vansummeren
Document Spanners: A Formal Approach to Information Extraction. JACM 2015.

Dominik D. Freydenberger, Mario Holldack
Document Spanners: From Expressive Power to Decision Problems. ICDT 2016.

**Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, Domagoj Vrgoc**
**Constant Delay Algorithms for Regular Document Spanners. PODS 2018.**

Antoine Amarilli, Pierre Bourhis, Stefan Mengel, Matthias Niewerth
Constant-Delay Enumeration for Nondeterministic Document Spanners. ICDT 2019

Liat Peterfreund, Balder ten Cate, Ronald Fagin, Benny Kimelfeld
Recursive Programs for Document Spanners. ICDT 2019,

Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, Frank Neven
Split-Correctness in Information Extraction. PODS 2019.

Markus L. Schmid, Nicole Schweikardt
Spanner Evaluation over SLP-Compressed Documents. PODS 2021.

(. . . )

# Ideas principales en investigación de REmatch

1. Expresiones regulares con variables de capturas
2. Variable-set automata
3. Algoritmos de enumeración con delay constante

# 1. Expresiones regulares con variables de capturas

Sintaxis de **variable regex**

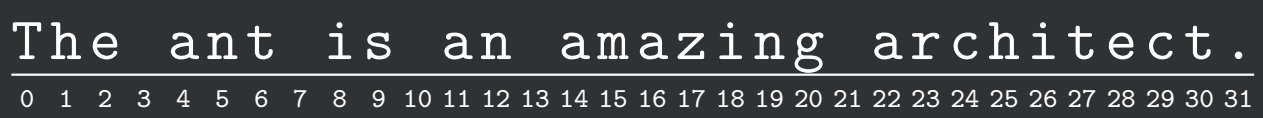
$$
\begin{array}{rcll}
R & := & a & \text{(letra en } \Sigma\text{)} \\
 & | & \varepsilon & \text{(palabra vacía)} \\
 & | & \varnothing & \text{(conjunto vacío)} \\
 & | & R + R & \text{(disyunción)} \\
 & | & R \cdot R & \text{(concatenación)} \\
 & | & R^* & \text{(clausura de Kleene)} \\
 & | & \mathbf{x}\{R\} & \text{(\textbf{variable de captura} de un conjunto } \mathbf{X}\text{)}
\end{array}
$$

# 1. Expresiones regulares con variables de capturas

## Spans y mappings

- Un **span** $s$ de un documento $d = a_0 \dots a_{n-1}$ es un par $[i, j\rangle$ con $0 \leq i \leq j \leq n$ que representa a la subpalabra $a_i \dots a_{j-1}$.
- Un **mapping** $\mu$ de $\mathbf{X}$ sobre $d$ es una función $\mu$ de $\mathbf{X}$ a spans de $d$.

### Ejemplo

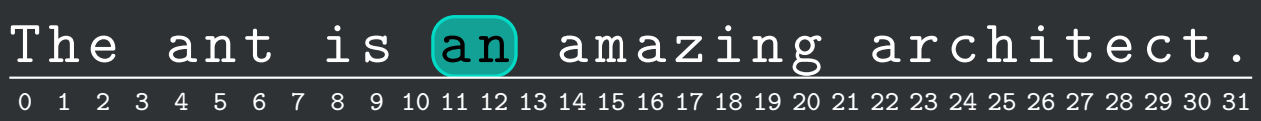


# 1. Expresiones regulares con variables de capturas

## Spans y mappings

- Un **span** $s$ de un documento $d = a_0 \dots a_{n-1}$ es un par $[i, j\rangle$ con $0 \leq i \leq j \leq n$ que representa a la subpalabra $a_i \dots a_{j-1}$.
- Un **mapping** $\mu$ de $\mathbf{X}$ sobre $d$ es una función $\mu$ de $\mathbf{X}$ a spans de $d$.

### Ejemplo



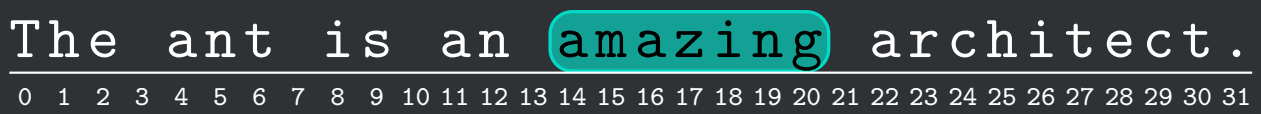

Ejemplos de **spans**: $s_1 = [11, 13\rangle$

# 1. Expresiones regulares con variables de capturas

## Spans y mappings

- Un **span** $s$ de un documento $d = a_0 \dots a_{n-1}$ es un par $[i, j\rangle$ con $0 \leq i \leq j \leq n$ que representa a la subpalabra $a_i \dots a_{j-1}$.
- Un **mapping** $\mu$ de $\mathbf{X}$ sobre $d$ es una función $\mu$ de $\mathbf{X}$ a spans de $d$.

### Ejemplo



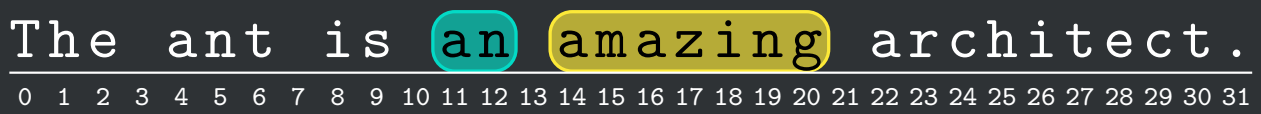

Ejemplos de **spans**: $s_1 = [11, 13\rangle$ $s_2 = [14, 21\rangle$

# 1. Expresiones regulares con variables de capturas

## Spans y mappings

- Un **span** $s$ de un documento $d = a_0 \dots a_{n-1}$ es un par $[i, j\rangle$ con $0 \leq i \leq j \leq n$ que representa a la subpalabra $a_i \dots a_{j-1}$.
- Un **mapping** $\mu$ de $\mathbf{X}$ sobre $d$ es una función $\mu$ de $\mathbf{X}$ a spans de $d$.

## Ejemplo




Ejemplos de **spans**: $s_1 = [11, 13\rangle$ $s_2 = [14, 21\rangle$

Ejemplo de un **mapping**: $\{\mathbf{x} \mapsto [11, 13\rangle, \mathbf{y} \mapsto [14, 21\rangle\}$

# 1. Expresiones regulares con variables de capturas

## Semántica de **variable regex**

$$
\begin{aligned}
[a]_d &= \{(s, \varnothing) \mid s \in \text{spans}(d) \text{ and } d(s) = a\} \\
[R_1 + R_2]_d &= [R_1]_d \cup [R_2]_d \\
[R_1 \cdot R_2]_d &= \{(s, \mu) \mid \exists (s_1, \mu_1) \in [R_1]_d . \\
&\qquad \exists (s_2, \mu_2) \in [R_2]_d : s = s_1 \cdot s_2, \\
&\qquad \text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \varnothing, \text{ and } \mu = \mu_1 \cup \mu_2\} \\
[R^*]_d &= [\varepsilon]_d \cup [R]_d \cup [R^2]_d \cup [R^3]_d \cup \cdots \\
[\mathbf{x}\{r\}]_d &= \{(s, \mu) \mid \exists (s, \mu') \in [R]_d : \\
&\qquad \mathbf{x} \notin \text{dom}(\mu') \text{ and } \mu = [\mathbf{x} \to s] \cup \mu'\} \\
[\![R]\!]_d &= \{\mu \mid ([0, |d|\rangle, \mu) \in [R]_d\}
\end{aligned}
$$

Una **variable regex** retorna un **conjunto de mappings** de $d$

# 2. Variable-set automata

## Definition

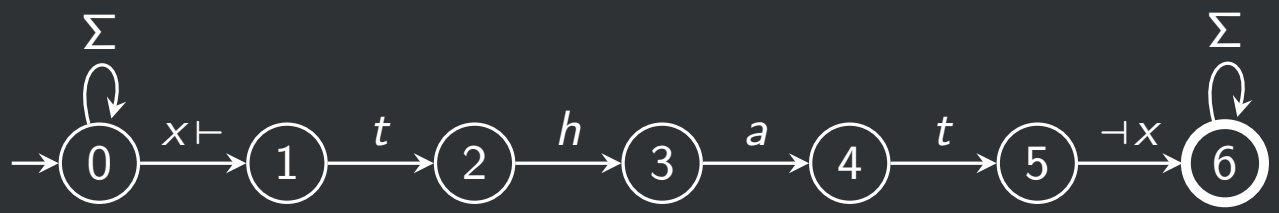
Un variable set automata (**vset automata**) es una tupla:

$$\mathcal{A} \quad = \quad (Q, \Sigma, \mathbf{X}, \Delta, I, F)$$

- $Q$ es un conjunto finito de estados
- $\Sigma$ es un alfabeto finito
- $I \subseteq Q$ son los estados iniciales
- $F \subseteq Q$ son los estados finales
- $\mathbf{X}$ es un conjunto finito de **variables**
- $\Delta \subseteq (Q \times \Sigma \times Q) \cup (Q \times \{x\vdash, \dashv x \mid x \in \mathbf{X}\} \times Q)$
  es la **relación de transición**

# 2. Variable-set automata

## El funcionamiento de un vset automata **con un ejemplo**




$$d = \underset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9}{\texttt{thathathat}}$$

Algunas **ejecuciones de aceptación** sobre $d$:

$$\rho_1 : 0 \xrightarrow{x\vdash} 1 \xrightarrow{t} 2 \xrightarrow{h} 3 \xrightarrow{a} 4 \xrightarrow{t} 5 \xrightarrow{\dashv x} 6 \xrightarrow{h} 6 \xrightarrow{a} 6 \xrightarrow{t} 6 \xrightarrow{h} 6 \xrightarrow{a} 6 \xrightarrow{t} 6$$

$$\rho_2 : 0 \xrightarrow{t} 0 \xrightarrow{h} 0 \xrightarrow{a} 0 \xrightarrow{x\vdash} 1 \xrightarrow{t} 2 \xrightarrow{h} 3 \xrightarrow{a} 4 \xrightarrow{t} 5 \xrightarrow{\dashv x} 6 \xrightarrow{h} 6 \xrightarrow{a} 6 \xrightarrow{t} 6$$

$$\mu(\rho_1) = \{x \mapsto [0,4\rangle\} \qquad \mu(\rho_2) = \{x \mapsto [3,7\rangle\}$$

$$\underbrace{\texttt{that}}_{x}\texttt{hathat} \qquad \texttt{tha}\underbrace{\texttt{that}}_{x}\texttt{hat}$$

# 3. Algoritmos de enumeración con delay constante

## Problem de evaluación principal

`INPUT:` Una variable regex $R$ y un documento $d$

`OUTPUT:` Enumerar $[\![R]\!]_d = \{\mu_1, \mu_2, \ldots, \mu_k\}$

## Teorema

Para cada variable regex $R$,
uno puede construir un vset automata **equivalente** $\mathcal{A}_R$ en tiempo lineal.

# 3. Algoritmos de enumeración con delay constante

## Problem de evaluación principal

`INPUT:` Un **vset automata** $\mathcal{A}$ y un documento $d$

`OUTPUT:` Enumerar $[\![\mathcal{A}]\!]_d = \{\mu_1, \mu_2, \ldots, \mu_k\}$

## Problema con la eficiencia

`!x1{.+}!x2{.+}!x3{.+}...!xk{.+}`

- ¿Qué hace esta consulta REQL?
- Si el documento tiene tamaño $n$, ¿cuántos resultados tendremos?

¿cómo diseñar un algoritmo eficiente cuando tenemos **muchos resultados**?

# 3. Algoritmos de enumeración con delay constante

## Problem de evaluación principal

`INPUT:` Un **vset automata** $\mathcal{A}$ y un documento $d$

`OUTPUT:` Enumerar $[\![\mathcal{A}]\!]_d = \{\mu_1, \mu_2, \ldots, \mu_k\}$

## Algoritmo en dos fases

- **Preprocessing**: construir un índice $\mathcal{I}$ en tiempo $\mathcal{O}(f(\mathcal{A}) \cdot |d|)$.
- **Enumerar**: usando $\mathcal{I}$, el algoritmo entrega los resultados:

$$\underset{\uparrow 0}{\#}\ \mu_1\ \underset{\uparrow 1}{\#}\ \mu_2\ \underset{\uparrow 2}{\#}\ \cdots\ \underset{\uparrow k-1}{\#}\ \mu_k\ \underset{\uparrow k}{\#}$$

tal que existe una constante $C \in \mathbb{N}$ donde para todo $i \leq k$ el tiempo entre los #-simbolos $(i-1)$ e $i$ es menor a $C \cdot |\mu_i|$.

El tiempo de entregar el próximo resultado **depende solo de su tamaño**

# 3. Algoritmos de enumeración con delay constante

## Problem de evaluación principal

`INPUT:` Un **vset automata** $\mathcal{A}$ y un documento $d$

`OUTPUT:` Enumerar $[\![\mathcal{A}]\!]_d = \{\mu_1, \mu_2, \ldots, \mu_k\}$

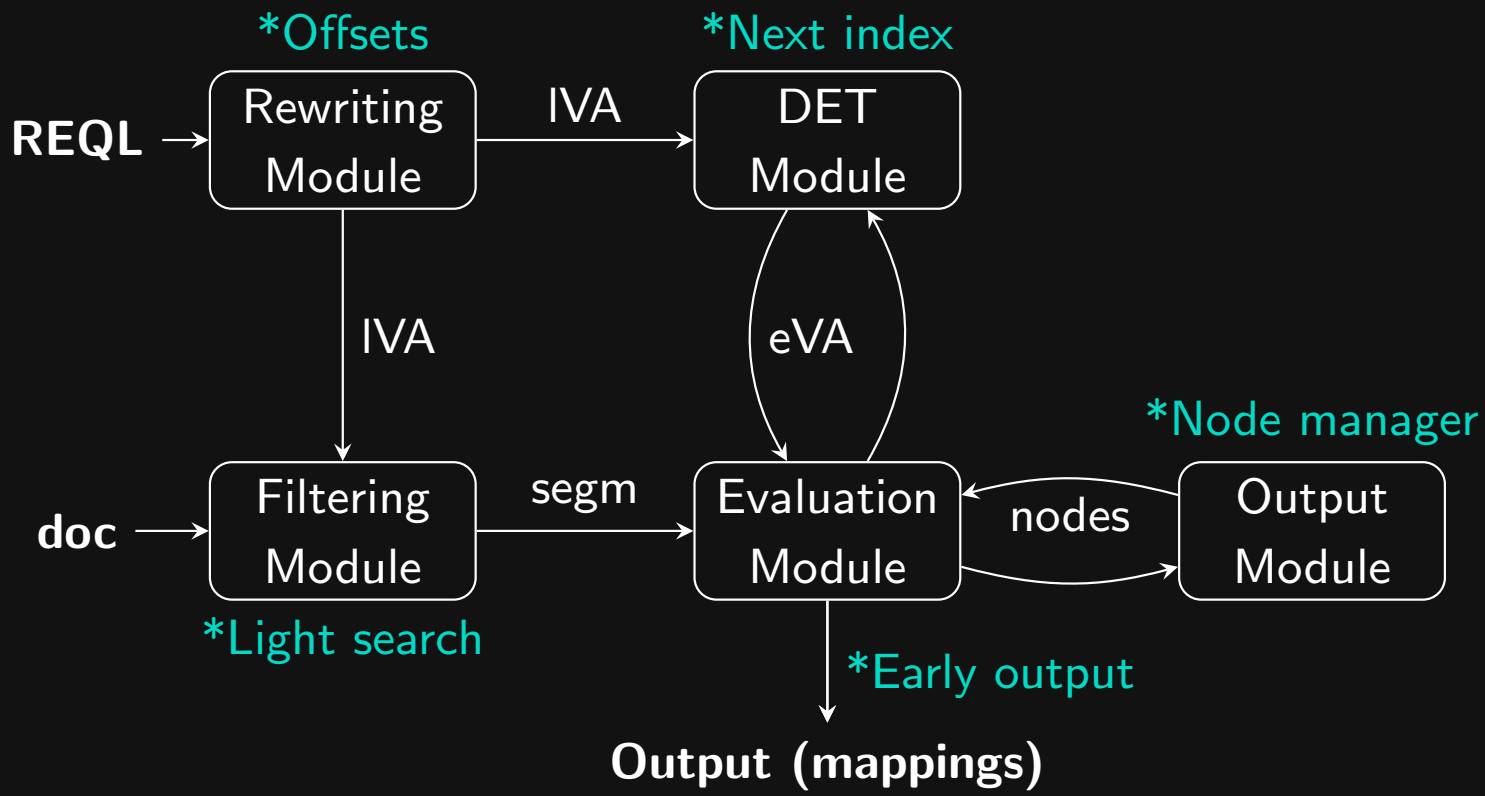
## Teorema

Existe un algoritmo de enumeración constante que dado un **vset automata determinista** $\mathcal{A}$ y un documento$d$, enumera $[\![\mathcal{A}]\!]_d$.

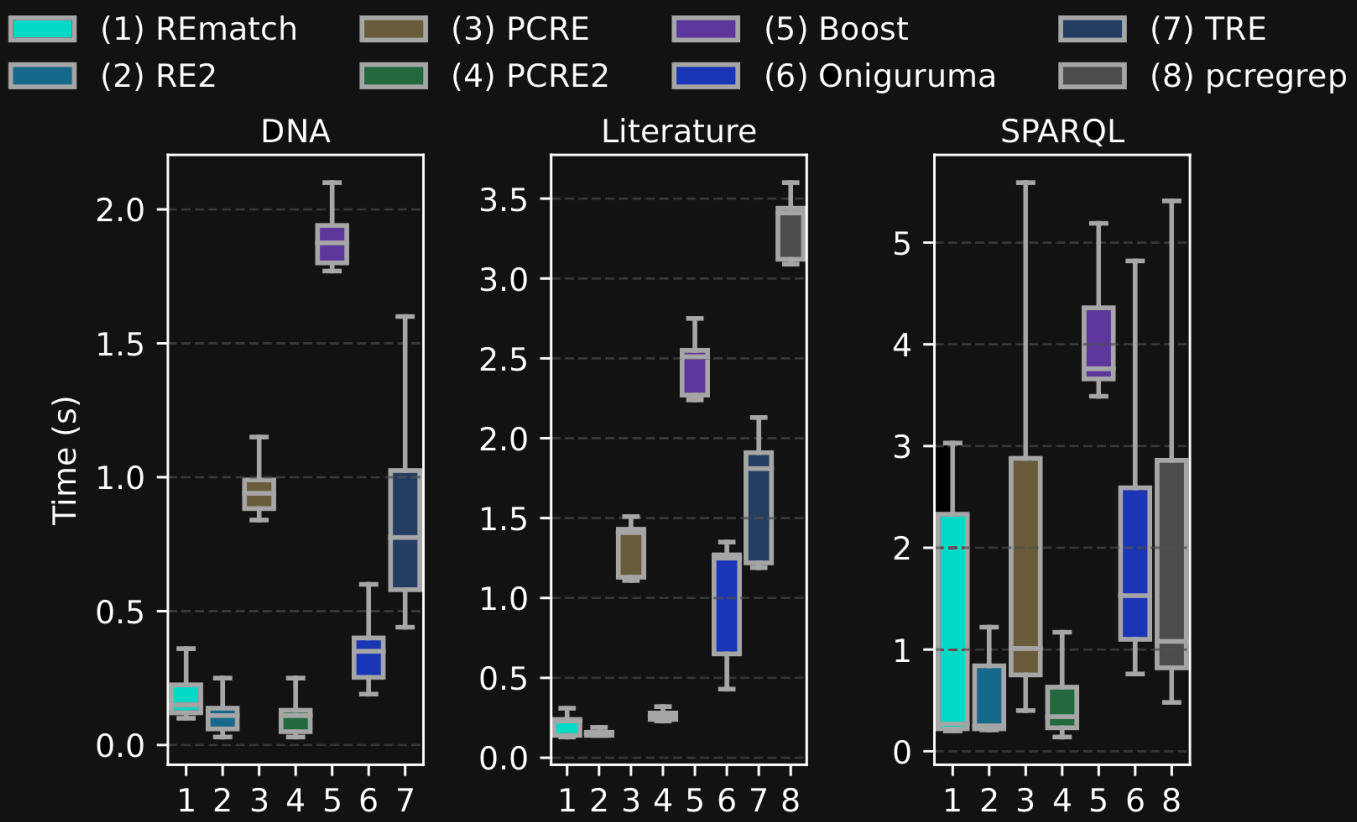
El **algoritmo de evaluación de REmatch** está basado en este resultado

# 3. Algoritmos de enumeración con delay constante

## Dentro de REmatch



# 3. Algoritmos de enumeración con delay constante




El rendimiento de REmatch es **comparable** con motores estándares de regex

... y adicionalmente encuentra todos los matches.

# Ideas principales en investigación de REmatch

1. Expresiones regulares con variables de capturas ✓
2. Variable-set automata ✓
3. Algoritmos de enumeración con delay constante ✓

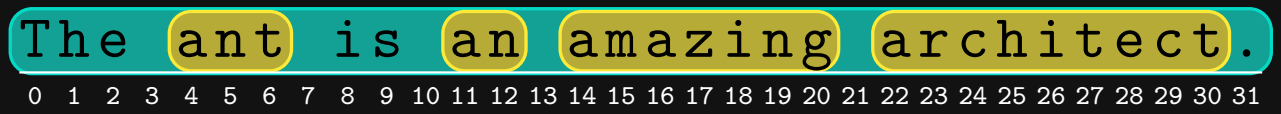
Estos resultados teóricos nos permiten ir **más allá** de regex

# Multimatch: beyond regex

```
The ant is an amazing architect.
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

*"Encontrar todas las* **oraciones**
*junto con* **todas sus palabras** *que* **empiezan** *con* **'a'**."

# Multimatch: beyond regex




*"Encontrar todas las* **oraciones**
*junto con* **todas sus palabras** *que* **empiezan** *con* **'a'**.*"*

```
(^ |\.␣)!sent{
    (([^Aa]\w*␣)*!ws{[Aa]\w*}␣)*
        ([^Aa]\w*␣)*([^Aa]\w*|!ws{[Aa]\w*})\.}
```

## REQL Query

FINISHED (FOUND 1)

```
(^|\.·)!sentence{(([^Aa]\w*·
)*!ws{[aA]\w*}·)*([^Aa]\w*·)*
([^Aa]\w*|!ws{[aA]\w*})?\.}
```

MULTI

## Document

```
1  The ant is an amazing architect.
```

## Matches

Match 0

| | |
|---|---|
| sentence | (0-32) The·ant·is·an·amazing·architect. |
| ws | (4-7) ant |
| | (11-13) an |
| | (14-21) amazing |
| | (22-31) architect |

‹ 1 ›


!RE{MATCH}

# Investigación en REmatch

Son un **resumen/ideas** de los siguientes artículos:

Maturana, Riveros, Vrgoc
"*Document Spanners for Extracting Incomplete Information.*" PODS 2018.

Florenzano, Riveros, Ugarte, Vansummeren, Vrgoc
"*Constant Delay Algorithms for Regular Document Spanners.*" PODS 2018.

Grez, Riveros, Ugarte
"*A Formal Framework for Complex Event Processing.*" ICDT 2019.

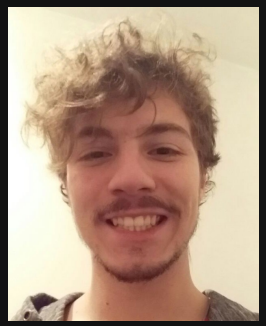
Muñoz, Riveros
"*Streaming Enumeration on Nested Documents.*" ICDT 2022.

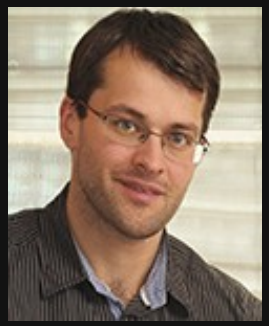
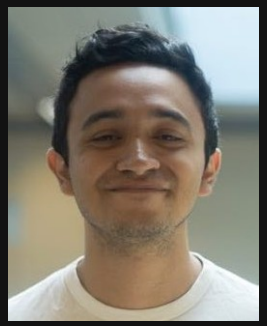
Amarilli, Jachiet, Muñoz, Riveros
"*Efficient Enumeration for Annotated Grammars.*" PODS 2022.

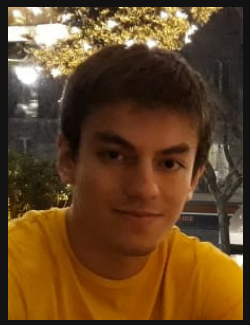
Riveros, Van Sint Jan, Vrgoc
"*REmatch: a novel regex engine for finding all matches.*" VLDB 2023.

$+$ artículos de otros grupos $+$ investigación en curso.

# Investigación en REmatch




(Faltan varios)

# Sobre teoría de bases de datos

Manejo de datos:

- Big data.
- Datos streaming.
- Extracción de información.

Grafos de datos:

- Web semántica.
- Base de datos de grafos.
- Centralidad de datos.

Lógica / Lenguajes formales:

- Teoría de modelos finitos.
- Teoría de automatas.

Teoría de la Computación:

- Complejidad computacional.
- Algoritmos aleatorios.

# Proyectos de implementación

1. REmatch — **Motor de extracción de información**
2. CORE — **Base de datos streaming**
3. MilleniumDB — **Base de datos de grafos**

Estan **invitados a colaborar** en cualquiera de estos proyectos

(solo escribanme y pregunten)

FIN