

# **SISTEMAS EVOLUTIVOS DE REESCRITURA**

**Fernando Galindo Soria**

**IPN - ESCOM**

**fgalindo@ipn.mx**

## **RESUMEN**

**Tal vez una de las herramientas más poderosas y desconocidas de la Matemática actual sean la reglas de reescritura, su fuerza es tan grande que valdría la pena que su uso se introdujera sin mucho formalismo desde el nivel de educación primaria y secundaria, sin embargo y a pesar de que su campo de aplicación es enorme, la potencia de esta herramienta se ha diluido dentro del concepto de gramática generativa. Por lo que en este documento se presentaran las reglas de reescritura como una herramienta independiente de las gramáticas y se mostrara su campo de aplicación**

**Como primer punto se dará una breve introducción al tema y se vera como un sistema de reescritura se puede representar fácilmente mediante un sistema de información integrado por un programa de control y un archivo donde se almacenan las reglas de reescritura. Con esta arquitectura el conocimiento queda independiente del programa y el proceso de mantenimiento y actualización del sistema se vuelve simple, ya que si surgen nuevas reglas solo se requiere almacenarlas en el archivo y el programa no se modifica. Aun más el sistema puede empezar a construir la base de conocimiento desde cero, ya que el archivo puede estar vacío y el sistema en forma interativa lo puede empezar ha llenar, con lo que se tiene la base de un Sistema Evolutivo.**

**A continuación se mostrara que problemas aparentemente diferentes como el reconocimiento de imágenes, señales, sistemas experto, tutores, traductores, lenguaje natural elemental, y prácticamente cualquier problema que se pueda representar como una cadena de bits o caracteres, son casos particulares de un problema mas general atacable con reglas de reescritura.**

**Finalmente se vera que cualquier programa o algoritmo X se puede representar como un conjunto de reglas de reescritura, que demostrar un teorema y hacer un programa son actividades equivalentes y que las reglas de reescritura permiten representar directamente a las gramáticas tipo cero.**

## **PALABRAS CLAVES**

**Reglas de Reescritura, Sistemas Evolutivos, Gramáticas y Autómatas, Sistemas expertos, Lenguaje Natural, Reconocimiento de Imágenes, Voz y Señales.**

# INTRODUCCIÓN

Una de las herramientas más poderosas y desconocidas de la matemática actual son las reglas de reescritura, descubiertas dentro de la lingüística matemática y aplicadas inicialmente en esta área y más adelante para la construcción de compiladores y en general de sistemas dirigidos por sintaxis.

La fuerza de esta herramienta se ha diluido, porque normalmente solo se aplica como parte de las gramáticas generativas y en particular como reglas de producción, a pesar de que cualquier sistema, sistema formal o sistema evolutivo puede manejar como herramienta de representación a estas reglas. En general cualquier problema de Informática, Computación o Inteligencia Artificial se puede representar en término de reglas de reescritura (es como si el operador de suma sólo se usara para sumar números naturales, hasta que se libera del entorno de los números naturales adquiere toda su importancia ).

Su fuerza es tan grande que valdría la pena que el uso de estas reglas se introdujera sin mucho formalismo desde el nivel de primaria y secundaria, como una herramienta mas de la Matemática, tan importante como la suma y la resta, ya que abre a los alumnos todo un nuevo universo.

## 1.) CONCEPTOS GENERALES

Como primer punto partiremos de que un **sistema de reescritura**  $\langle A, B, R \rangle$  esta formado por un conjunto de elementos de entrada **A**, un conjunto de elementos de salida **B** y un conjunto de reglas de reescritura **R**. (Donde **A** y **B** pueden ser iguales o diferentes).

Una **regla de reescritura** es de la forma:

**X**--> **Y**

Donde  $X \in A^*$ ,  $Y \in B^*$  ( $A^*$  y  $B^*$  representan respectivamente las cerradura de A y B ). Y lo anterior se lee como: **X se puede reescribir como Y** y significa que **X** se puede sustituir por **Y**.

Es relativamente fácil construir un sistema de información que representa a los sistemas de reescritura, por ejemplo el siguiente sistema formado por un programa y un archivo con dos columnas :

A1	B1
A2	B2
""	""
An	Bn

```
Programa()  
{  
  i=1  
  lee Ax  
  mientras ((Ax != Ai) y (no fin de archivo))i++
```

```
si (no fin de archivo) escribe Bi  
sino escribe "no reconozco Ax"  
}
```

El archivo representa al conjunto de reglas de reescritura:

```
A1 --> B1  
A2 --> B2  
""  
An --> Bn
```

Y el programa representa a un proceso que reescribe  $A_i$  como  $B_i$ .

## 2.) PRESENTACIÓN DE LA HERRAMIENTA GENERAL.

Esta idea se puede aplicar por ejemplo para hacer un traductor automático, donde se puede tener una tabla que contiene en la primera columna el texto en un idioma y en la segunda su traducción a otro idioma.

texto	traducción
este es un perro	this is a dog
el perro blanco	the white dog
""	""
el perro negro	the black dog

Y un pequeño programa que realiza la sustitución de reglas.

```
Programa()  
{  
  i=1  
  lee Textox  
  mientras ((Textox != Textoj) y (no fin de archivo))i++  
  si (no fin de archivo) escribe Traducciónj;  
  sino escribe "no conozco el texto"  
}
```

Como se podrá ver el sistema es pequeño y en general trivial de modificar, ya que el conocimiento queda separado del programa, al estar almacenado en registros, por lo que, si se encuentra una nueva regla lo único que se requiere es aumentar un registro en el archivo y el programa no se toca.

Aun mas, se puede hacer que el mismo programa actualice el archivo, de tal manera que si entra un texto desconocido, el programa pueda preguntar (al usuario o a un experto) la traducción asociada, con lo que se tendría un pequeño **sistema evolutivo**.

texto	traducción
este es un perro	this is a dog
el perro blanco	the white dog
""	""
el perro negro	the black dog

```

Programa()
{
  i=1
  lee Textox
  mientras ((Textox != Textoj) y (no fin de archivo))i++
  si (no fin de archivo) escribe Traduccióni;
  sino //entra al dialogo
    escribe "desconozco Textox dame su traducción"
    lee Traducciónx
    almacena en el archivo Textox , Traducciónx
}

```

Como se podrá ver este programa es muy simple de hacer y tiene la ventaja de que no requiere tener almacenado el conocimiento previamente, ya que si el archivo estuviera vacío y se preguntara por **Texto<sub>x</sub>** el programa pediría **Traducción<sub>x</sub>** y ya tendría la regla **Texto<sub>x</sub> -> Traducción<sub>x</sub>** con lo que, **el sistema evolutivo puede empezar ha construir la base de conocimiento y 'aprender' desde cero, en tiempo real y fácilmente.**

Aunque se tienen que tomar en cuenta algunas consideraciones prácticas, por ejemplo que sólo se permita 'enseñar' al sistema a personas con clave de autorización, estos motivos no afectan ni la idea ni la estructura del sistema.

Si seguimos con la misma idea el programa se puede generalizar aun mas y aplicar para atacar otros problemas aparentemente diferentes, como es la **construcción de sistemas experto, tutores automatizados y en general al tratamiento restringido de lenguaje natural.**

Por ejemplo si se analizan algunos sistemas experto se observa que es común que los programas en su forma más simple tengan una estructura de cascadas de IF's de la forma:

```

Programa()
{
  lee Sx
  si Sx = S1 entonces D1 sino
  si Sx = S2 entonces D2 sino
  "" ""
  si Sx = Sn entonces Dn sino
  escribe "no reconoce los síntomas"
}

```

Donde el programa lee los síntomas **Sx** que da el usuario y los compara con los primeros síntomas almacenados **S1**, si corresponden entonces envía el diagnóstico **D1**, sino los compara con los síntomas **S2** y así sucesivamente.

El problema de este enfoque es que si se quieren aumentar las reglas o modificar al sistema se tiene que reestructurar el programa. Ahora bien, usando prácticamente el mismo sistema evolutivo desarrollado para la traducción es relativamente fácil construir un programa que generaliza al anterior.

Para lo cual sustituiremos la cascada de IF's por un sistema formado por un archivo de reglas de reescritura de la forma **Si --> Di**, donde se almacenan los síntomas y diagnósticos.

Síntomas	Diagnostico
S1	D1
S2	D2
""	""
Sn	Dn

Y por un pequeño programa que realiza la sustitución de reglas.

```

Programa()
{
  i=1
  lee Sx
  mientras ((Sx != Si) y (no fin de archivo))i++
  si (no fin de archivo) escribe Di
  sino //entra al dialogo
    escribe "desconozco Sx, dame su diagnostico"
    lee Dx
    almacena en el archivo Sx , Dx
}

```

Nuevamente con este sistema se puede tener el archivo vacío y conforme se realiza el proceso de dialogo las reglas del sistema experto se van integrando en forma natural, con lo que se tiene un sistema evolutivo que genera las reglas de inferencia de un sistemas expertos. Lo único que cambia entre el traductor y el sistema experto es que en lugar de escribir:

desconozco texto, dame su traducción  
 escribe  
 desconozco **Sx**, dame su diagnostico

Aun mas, si se cambian las reglas, con el mismo programa se puede tener un sistema de reconocimiento de lenguaje natural elemental. Por ejemplo, en lugar de poner los síntomas y diagnósticos de un sistema experto se pueden poner las preguntas y respuestas de un sistema tutor. En este caso se puede tener una lista de preguntas y respuestas como las siguientes:

¿quien descubrió América?	Colon
¿cual es la capital de Francia?	París
""	""
¿en que continente esta Rusia?	en Europa

Entonces **el mismo programa** funciona como un sistema tutor, donde el alumno pregunta y el sistema responde, y **en lugar de síntomas y diagnósticos** el sistema **tiene preguntas y respuestas**, lo cual es interesante ya que **el mismo sistema se puede ver como un sistema experto o como un tutor automatizado**. Y lo único que cambia es que en lugar de escribir

desconozco Sx, dame su diagnostico  
 escribiría  
 no conozco la respuesta, dámela

### 3.) GENERALIZACIÓN PARA MANEJO DE SEÑALES

Como se puede ver una gran cantidad de problemas aparentemente diferentes se pueden reducir en primera instancia al mismo sistema, sin embargo no son los únicos casos que se pueden manejar con esta herramienta, ya que prácticamente se puede atacar cualquier problema que se pueda representar como una cadena de bits o caracteres.

Para poder ver lo anterior se trabajara primero con algo muy simple, ya que lo importante no es resolver el problema en toda su complejidad, sino **mostrar una de las cosas mas hermosas de la Informática**, que **el reconocimiento de imágenes, texto, traductores, sistemas experto, señales biofísica, voz, etc., que normalmente se ven como cuestiones completamente diferentes son en realidad casos particulares de un problema general**.

Como primer ejemplo se vera un sistema que entrega una imagen a partir de su nombre. Este sistema consta de un archivo con dos columnas, en la primera columna se encuentra el nombre de la imagen y en la segunda su imagen correspondiente -almacenada por ejemplo como cadena de bits-, ademas se cuenta con un programa construido de tal manera que si alguien da el nombre de la imagen **Nx** regresa su imagen correspondiente **Ix** y si no la tiene entra en un proceso de dialogo donde la pide y almacena.

Nombre	Imagen
N1	I1
N2	I2
""	""
Nn	In

Programa()

```
{
i=1
lee Nx
mientras ((Nx != Ni) y (no fin de nombres))i++
```

```

si (no fin de nombres) escribe Ii
sino //entra al dialogo
    escribe "no conozco la imagen dámela"
    lee Ix
    almacena Ix y Nx
}

```

Como se puede ver este sistema es prácticamente idéntico al que se ha estado manejando desde el principio para texto con la diferencia de que ahora un lado de la regla de reescritura no es texto.

Ha partir de una idea tan simple es fácil construir un **programa que pronuncie palabras ha partir de texto escrito**, en este caso en la primera columna se tiene el texto y en la segunda el sonido asociado, en caso de que la computadora no encontrara la cadena escrita puede entrar en un proceso de dialogo donde se grabaría el texto y su sonido.

A continuación se puede guardar en la primera columna una serie de imágenes y en la segunda sus nombres, entonces si se presenta una nueva imagen, el sistema puede compararla con las que ya tiene, si la encuentra regresa su nombre, sino lo pide y almacena la imagen y el nombre.

Imagen	Nombre
I1	N1
I2	N2
""	""
In	Nn

```

Programa()
{
i=1
lee Ix
mientras ((Ix != Ii) y (no fin de imágenes))i++
si (no fin de imágenes) escribe Ni
sino //entra al dialogo
    escribe "no conozco la imagen dame su nombre"
    lee Nx
    almacena Ix y Nx
}

```

**Con lo que se tiene un sistema elemental de reconocimiento de imágenes usando el mismo que se utilizo para tratamiento de texto**, donde en lugar de manejar cadenas de caracteres, se manejan imágenes, con lo que la programación especifica se puede complicar, porque no es lo mismo manejar registros de unos cuantos caracteres a manejar arreglos de miles de palabras, sin embargo la estructura general se conserva.

Aunque buscar imágenes idénticas se podría considerar como una "trampa", este sistema es un buen ejemplo para empezar con el tratamiento de imágenes, ya que el manejo de cadenas

idénticas es común en computación, por ejemplo en un compilador las palabras claves son iguales ha palabras ya almacenadas.

A partir de lo anterior se pueden atacar problemas de mayor complejidad, por ejemplo modificando el sistema para que encuentre la distancia entre la imagen de entrada y la almacenada y en su caso nos indique si son idénticas o su grado de semejanza o diferencia, con lo que se tiene las bases de un sistema de reconocimiento de formas.

El tratamiento de señales en general, se vuelve prácticamente idéntico, ya que en principio, lo que se tiene son archivos donde se guardan las señales y se comparan unas con otras.

Al almacenar la información **se ha construido una mina de información** sobre la cual se pueden aplicar una gran cantidad de técnicas y métodos de reconocimiento de formas, sistemas evolutivos, lingüística matemática e inferencia gramatical, redes neuronales y otras herramientas para el manejo de información.

#### 4.)GENERALIZACIÓN A CUALQUIER SISTEMA

Al proceso de sustituir **A<sub>i</sub>** por **B<sub>i</sub>** se le conoce como derivación directa y se denota como:  
**A<sub>i</sub> => B<sub>i</sub>**

Si **A = B**  
o **A => B<sub>1</sub> => B<sub>2</sub> => B<sub>3</sub> =>...=> B**  
entonces se dice que **A deriva B** y se denota como **A =>\* B**

En general cualquier sistema informático se puede ver como un proceso de derivación, ya que sí se tiene:



El sistema sustituye la entrada por la salida (a la mejor realizando millones de operaciones ), o sea

**Entrada => \* Salida**

Y la secuencia de pasos que llevan de la entrada a la salida se puede visualizar como:

**Entrada =>a1=>a2 =>...=> Salida**

Donde para cada paso de la forma **a<sub>i</sub> => a<sub>j</sub>** existe una regla de reescritura de la forma **a<sub>i</sub> --> a<sub>j</sub>** , por lo que el algoritmo o programa se puede ver como un sistema de reescritura.

De la misma forma un teorema se puede ver como

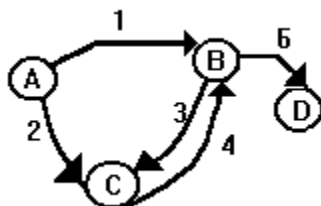
**A =>\* B**

Y la demostración del teorema se puede ver como:

$$A \Rightarrow a1 \Rightarrow a2 \Rightarrow \dots \Rightarrow B$$

O sea que, desde el punto de vista de los sistemas de reescritura demostrar un teorema y hacer un programa son equivalentes, lo cual es interesante, ya que normalmente se considera que es más fácil programar que demostrar teoremas, por lo que, si se adquiere conciencia de que son procesos equivalentes, se podría facilitar el desarrollo de la capacidad de demostración de teoremas ha partir de la capacidad de programar.

Aun mas esta herramienta se puede aplicar dentro del entorno de los sistemas axiomáticos (gramáticas, autómatas, sistemas de Post, etc. ). Por ejemplo si se tiene el siguiente diagrama de estados.



La gramática regular equivalente tiene reglas de reescritura de la forma	El autómata finito equivalente se representa con las herramientas tradicionales como	El autómata finito equivalente tiene reglas de reescritura de la forma
A --> 1B	$\delta(A,1) = B$	A1 --> B
A --> 2C	$\delta(A,2) = C$	A2 --> C
B --> 3C	$\delta(B,3) = C$	B3 --> C
B --> 5D	$\delta(B,5) = D$	B5 --> D
C --> 4B	$\delta(C,4) = B$	C4 --> B

En este ejemplo se ve la fuerza de las reglas de reescritura, ya que, con un cambio notacional se reduce drásticamente la complejidad del manejo de los autómatas.

Por ejemplo, si se quiere demostrar que  $A245 \Rightarrow^* D$  -o sea que ha partir de A se llega a D con las señales 2, 4 y 5- con la notación tradicional quedaría:

$$\delta(A,245) = \delta(\delta(A,24),5) = \delta(\delta(\delta(A,2),4),5) = \delta(\delta(C,4),5) = \delta(B,5) = D$$

Que desde el "punto de vista matemático" es poesía, pero desde el punto de vista aplicativo complica la vida (como un numero romano se puede ver más elegante que uno arábigo, pero es menos práctico ).

Usando reglas de reescritura queda:

$$A245 \Rightarrow C45 \Rightarrow B5 \Rightarrow D$$

que como se puede ver es mucho más simple. Por lo que sin entrar en mas detalle quiero comentar que, los teoremas de equivalencia entre gramáticas y autómatas se vuelven más simples usando reglas de reescritura en lugar de la notación tradicional.

Las reglas de reescritura tienen un rango de aplicación enorme y están en el núcleo de solución de muchos problemas aparentemente complejos. Lo anterior no es gratis, ya que si se recuerda la jerarquía de Chomsky, las gramáticas generativas tipo cero son las más generales y equivalen a las máquinas de Turing que pueden representar el algoritmo de solución de cualquier problema computable.

Ahora bien a pesar de su fuerza la máquina de Turing normalmente no se aplica a la solución de problemas reales por la dificultad notacional y operativa que presenta.

Sin embargo las gramáticas tipo 0 tienen reglas de la forma  $\alpha \rightarrow \beta$  donde  $\alpha$  y  $\beta$  son cadenas de elementos de algún conjunto. O sea que las gramáticas tipo cero están formadas por reglas de reescritura.

**Por lo que no es raro que un sistema de reescritura sea tan fuerte ya que representa a una gramática tipo cero y por equivalencia a una máquina de Turing.**

## CONCLUSIÓN

En este trabajo se mostró la fuerza de las reglas de reescritura, como se pueden manejar en forma independiente de las gramáticas y aplicar a áreas tan diferentes como el reconocimiento de imágenes, tratamiento de voz, traducción de lenguajes, sistemas expertos, reconocimiento de señales y muchas otras.

Se vio a estas áreas como casos particulares del mismo problema, por lo que se está perdiendo una cantidad enorme de energía al atacarlos como problemas independientes, ya que, muchas de las herramientas que se han encontrado para atacar un tipo de problema se podrían usar para atacar los otros y en su momento se deben desarrollar herramientas generales a las que les sean indistintas las aplicaciones particulares y aun mas que llegue un momento en el que **ya no se vea cada problema en forma particular sino que se ataque el problema general.**

Finalmente se vio que las gramáticas tipo cero se representan mediante reglas de reescritura, de donde se concluyo que **se puede construir un sistema con la fuerza de una máquina de Turing y la facilidad de un programa de unas cuantas líneas.**

## BIBLIOGRAFÍA

- 1.- Estructuras Sintácticas. *Noam Chomsky*. Ed. Siglo XXI
- 2.- Teoría Sintáctica. *Emmon Bach*. Ed. Anagrama.
- 3.- Formal Languages. *Salomaa*. Ed. Academic Press.
- 4.- Sistemas Evolutivos. *Fernando Galindo Soria*.  
Boletín de Política Informática. México, Septiembre de 1986.
- 5.- Sistemas Evolutivos: Nuevo Paradigma de la Informática. *Fernando Galindo Soria*.  
Memorias XVII Conf. Latinoamericana de Informática. Caracas Ven., julio de 1991.
- 6.- Syntactic Pattern Recognition. *Rafael C. Gonzalez y Michael C. Thomason*.  
Ed. Addison-Wesley.