



**Universidad Autónoma de Chiapas**  
**Facultad de Contaduría y Administración, C-I**

---



Licenciatura en  
INGENIERIA EN DESARROLLO Y TECNOLOGIAS DE SOFTWARE

Unidad Académica  
**TEORÍA MATEMÁTICA DE LA COMPUTACIÓN**

## **APUNTES**

Elaborado por:

**DR. JUAN JOSÉ TEVERA MANDUJANO**

Ciclo escolar: ENE-JUN 2023

## Índice

<b>Presentación</b>	3
<b>Objetivo General</b>	4
1. Conceptos básicos de la teoría de lenguaje.	5
2. Expresiones regulares	9
3. Autómatas Finitos.	11
3.1 Autómata Finito No Determinístico AFN	12
3.2 Autómata Finito Determinístico AFD	15
3.3 Conversión de una expresión regular a un AFD	16
4. Gramáticas	19
4.1 Derivaciones y Árbol de Análisis Sintáctico	21
4.2 Generadores de Analizadores Sintácticos (Parser)	23
4.3 Gold Parser Builder	26
4.4 Ejemplo en Gold Parser Builder	28
4.5 Construir la aplicación con Visual Studio C#.NET	33
<b>Conclusión</b>	39
<b>Referencias</b>	39
<b>Programa de Estudios de unidad de competencia</b>	40

## **Presentación**

Estudiar la teoría matemática de la computación es iniciar en la interesante disciplina de las ciencias de la computación que ha sido tema de estudio como materia vocacional para una diversidad de carreras profesionales enfocadas a las tecnologías de la información, sistemas computacionales, ciencias de la computación, electrónica, cibernética entre otras.

La teoría matemática de la computación está formada por modelos matemáticos que desde la década de los 50's del siglo pasado las investigaciones han contribuido para formar una sólida ciencia de la computación que se aplicada a todas actividades que el mundo actual realiza, por tal motivo ha sido el avance de la ciencia de la computación que las investigaciones van desde sus inicios de la máquinas de Turing, teoría de autómatas, estudio de los lenguajes y las gramáticas libres del contexto, problemas intratables y el estudio de la computabilidad.

Los temas tratados en este apunte contiene los conocimientos para que el estudiante desarrolle las competencias disciplinares que son : a) Aplica habilidades de abstracción y expresión matemática para la solución de problemas.<sup>[1]</sup> b) Formula modelos matemáticos para la solución de problemas mediante el desarrollo de aplicaciones de software para diversos entornos. c) Posee conocimientos formales sobre las bases matemáticas de la computación y los aplica en la solución de problemas. Para desarrollar éstas competencias, es indispensable que el estudiante tenga los conocimientos previos en la teoría de conjuntos, programación, estructura de datos y matemáticas discretas.

El apunte está dividido en cuatro capítulos fundamentales de la teoría de la computación que en un lenguaje sencillo proporciona la teoría básica con ejemplos ilustrativos y ejercicios. En el capítulo 1 se aborda los conceptos básicos de la teoría de lenguaje como principio rector del resto del documento, conceptos como el alfabeto, las cadenas, las cerraduras, el lenguaje son tratados. En el capítulo 2 es el estudio de las expresiones regulares como una notación para especificar el conjuntos de cadenas de un lenguaje. El capítulo 3 comprende la teoría de los autómatas, en este capítulo el estudiante obtiene la competencia de construir un autómata finito determinístico AFD a partir una expresión regular, para ello es necesarios estudiar el algoritmo denominado Construcción de Thompson que a partir de una expresión regular se contruye el autómata finito no

determinístico AFN, posteriormente se estudia el algoritmo de Construcción de Subconjuntos para obtener el AFD. El capítulo 4 es referente al estudio de las gramáticas y representa un tema interesante que va desde el estudio de las gramáticas, diferentes constructores de analizadores sintácticos, el estudio de Gold Parser Builder y la creación de una aplicación en Visual Studio .Net con C#.

Por último, manifiesto que es la primera versión realizada y considero una satisfacción la realización del presente apunte que será de gran utilidad a los estudiantes interesados en la Teoría Matemática de la Computación.

*Juan José Tevera Mandujano  
Profesor de Tiempo Completo*

## **Objetivo General**

Ofrecer a los estudiantes los conocimientos básicos en la Teoría Matemática de la Computación para aplicar los modelos matemáticos en solución de problemas de reconocimiento de cadenas tarea que realiza el analizador léxico, así también aplicar la herramienta Gold Parser Builder para construir gramáticas para desarrollar el análisis sintáctico.

## **1. Conceptos básicos de la teoría de lenguaje.**

En esta sección se presentan las definiciones de los términos más importantes utilizados en la teoría de lenguaje, el término lenguaje es de origen latín *lingua* y es el sistema o recurso a través del cual se establece la comunicación entre los seres humanos, los animales, así también para la comunicación entre los seres humanos y las máquinas en el mundo de la computación se utiliza el lenguaje formal que se enfocará esta sección entre la comunicación de seres humanos y las máquinas.

Para que se produzca la comunicación debe existir un transmisor, un receptor y lo que se desea comunicar; el transmisor y receptor puede el ser el humano-máquina o máquina-máquina y lo que se desea comunicar son códigos estándar en computación como ejemplo el código ASCII acrónimo inglés de American Standard Code for Information Interchange — Código Estándar Estadounidense para el Intercambio de Información compuesto por 128 símbolos compuesto por letras minúsculas y mayúsculas, dígitos, signos de puntuación, de admiración, de interrogación, llaves, corchetes, caracteres de control y demás símbolos; fue publicado en 1963 por el Instituto Nacional Estadounidense de Estándares (ANSI).

A medida que la tecnología informática se difundió a lo largo del mundo, se desarrollaron diferentes estándares y las empresas desarrollaron muchas variaciones del código ASCII para facilitar la escritura de lenguas diferentes al inglés que usaran alfabetos latinos. Se pueden encontrar algunas de esas variaciones clasificadas como "ASCII Extendido", aunque en ocasiones el término se aplica erróneamente para cubrir todas las variantes, incluso las que no preservan el conjunto de códigos de caracteres original ASCII de siete bits. La ISO 646 (1972), el primer intento de remediar el sesgo pro-inglés de la codificación de caracteres, creó problemas de compatibilidad, pues también era un código de caracteres de 7 bits. No especificó códigos adicionales, así que reasignó algunos específicamente para los nuevos lenguajes. De esta forma se volvió imposible saber en qué variante se encontraba codificado el texto, y, consecuentemente, los procesadores de texto podían tratar una sola variante. La tecnología mejoró y aportó medios para representar la información codificada en el octavo bit de cada byte, liberando este bit, lo que añadió otros 128 códigos

de carácter adicionales que quedaron disponibles para nuevas asignaciones. Por ejemplo, IBM desarrolló páginas de código de 8 bits, como la página de códigos 437, que reemplazaba los caracteres de control con símbolos gráficos como sonrisas, y asignó otros caracteres gráficos adicionales a los 128 bytes superiores de la página de códigos. Algunos sistemas operativos como DOS, podían trabajar con esas páginas de código, y los fabricantes de computadoras personales incluyeron soporte para dichas páginas en su hardware. Los estándares de ocho bits como ISO 8859 y Mac OS Roman fueron desarrollados como verdaderas extensiones de ASCII, dejando los primeros 127 caracteres intactos y añadiendo únicamente valores adicionales por encima de los 7-bits. Esto permitió la representación de un abanico mayor de lenguajes, pero estos estándares continuaron sufriendo incompatibilidades y limitaciones. Todavía hoy, ISO-8859-1 y su variante Windows-1252 (a veces llamada erróneamente ISO-8859-1) y el código ASCII original de 7 bits son los códigos de carácter más comúnmente utilizados.

Unicode y Conjunto de Caracteres Universal (UCS) ISO/IEC 10646 definen un conjunto de caracteres mucho mayor, y sus diferentes formas de codificación han empezado a reemplazar ISO 8859 y ASCII rápidamente en muchos entornos. Mientras que ASCII básicamente usa códigos de 7-bits, Unicode y UCS usan "code points" o apuntadores relativamente abstractos: números positivos (incluyendo el cero) que asignan secuencias de 8 o más bits a caracteres. Para permitir la compatibilidad, Unicode y UCS asignan los primeros 128 apuntadores a los mismos caracteres que el código ASCII. De esta forma se puede pensar en ASCII como un subconjunto muy pequeño de Unicode y UCS. La popular codificación UTF-8 recomienda el uso de uno a cuatro valores de 8 bits para cada apuntador, donde los primeros 128 valores apuntan a los mismos caracteres que ASCII. Otras codificaciones de caracteres como UTF-16 se parece a ASCII en cómo representan los primeros 128 caracteres de Unicode, pero tienden a usar 16 a 32 bits por carácter, así que requieren de una conversión adecuada para que haya compatibilidad entre ambos códigos de carácter. La palabra *ASCIIbético* (o, más habitualmente, la palabra "inglesa" *ASCIIbetical*) describe la ordenación según el orden de los códigos ASCII en lugar del orden alfabético.

El alfabeto es el término inicial en la teoría de lenguaje, un **alfabeto** es un conjunto finito de símbolos y no vacío que convencionalmente se utiliza la letra griega sigma  $\Sigma$  para designar un alfabeto. Entre los alfabetos más comunes se incluyen los siguientes:

1. El alfabeto binario  $\Sigma = \{0,1\}$
2. El alfabeto hexadecimal  $\Sigma = \{0,1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$
3.  $\Sigma = \{a,b,...,z\}$ , el conjunto de todas las letras minúsculas.
4. El conjunto de todos los caracteres ASCII o el conjunto de todos los caracteres ASCII imprimibles.

Los símbolos de un alfabeto  $\Sigma$  se pueden unir formando secuencias finitas de símbolos, a estas secuencias de se les llama **cadena** o palabra, por ejemplo para el alfabeto binario  $\Sigma = \{0,1\}$ , algunas cadenas son 01101, 101, 101010111; para el alfabeto hexadecimal  $\Sigma = \{0,1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ , algunas cadenas son 367FE, 8AF0, FFA100, A8C. Obsérvese que las cadenas pueden tener diferentes números de símbolos y se pueden repetir, lo que nos lleva a la definición de **longitud** que indica la cantidad de caracteres de la cadena y se denota por  $|w|$  donde  $w$  es la cadena, por ejemplo las longitudes de las cadenas 01101, 101010111 y FFA100 son  $|01101| = 5$ ,  $|101010111| = 9$ ,  $|FFA100| = 6$ .

La cadena vacía es aquella cadena que presenta cero apariciones de símbolos, es una cadena que no contiene símbolos y su longitud es cero. Esta cadena, designada por  $\epsilon$ , es una cadena que puede construirse en cualquier alfabeto.

Si  $\Sigma$  es un alfabeto, podemos expresar el conjunto de todas las cadenas de una determinada longitud de dicho alfabeto utilizando una notación exponencial. Definimos  $\Sigma^k$  para que sea el conjunto de las cadenas de longitud  $k$ , tales que cada uno de los símbolos de las mismas pertenece a  $\Sigma$ .

Por ejemplo si el alfabeto  $\Sigma = \{0,1\}$  entonces tenemos que

$$\Sigma^0 = \{ \epsilon \} \text{ cadena de longitud 0}$$

$$\Sigma^1 = \{ 0,1 \} \text{ cadenas de longitud 1}$$

$$\Sigma^2 = \{ 00, 01, 10, 11 \} \text{ cadenas de longitud 2}$$

$$\Sigma^3 = \{ 000, 001, 010, 011, 100, 101, 110, 111 \} \text{ cadenas de longitud 3}$$

$\Sigma^4 = \{ 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 \}$  cadenas de longitud 4

Y así sucesivamente para  $k = 5, 6, 7, \dots$

Si  $\Sigma$  es un alfabeto, se define  $\Sigma^*$  al conjunto de todas las cadenas de cualquier longitud, esto es, todas las cadenas iniciando desde la longitud 0, 1, 2, 3, ... expresado en forma de conjunto

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$  Se le llama cerradura vacía o clausura de Kleene y

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$  Se le llama cerradura positiva

Si  $x$  e  $y$  son dos cadenas de un alfabeto  $\Sigma$ , entonces  $xy$  es la concatenación de la cadena  $x$  seguida de la cadena  $y$  con longitud igual a la suma de ambas cadenas, así si  $z = xy$  entonces  $|z| = |x| + |y|$ . La concatenación se representa con el símbolo punto, así  $x.y$  a veces se omite el punto y se escribe  $xy$  solamente.

Así también, se define al **lenguaje**  $L$  como un subconjunto de cadenas de  $\Sigma^*$  y donde  $\Sigma$  es un determinado alfabeto, así  $L \subseteq \Sigma^*$

Algunos ejemplos de lenguajes son:

1. El lenguaje  $L$  del conjunto de todas las cadenas de 0's y 1's que inician con 1 y terminan con 00, donde el alfabeto  $\Sigma = \{0,1\}$ , entonces  
 $L = \{100, 1000, 1100, 10000, 10100, 11000, 11100, \dots\}$
2. El lenguaje  $L$  del conjunto de todas las cadenas de números positivos que son divisibles entre 5, donde el alfabeto  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ , entonces  $L = \{5, 10, 15, 20, 25, 30, 35, 40, 45, \dots\}$
3. El lenguaje  $L$  de todas las palabras en mayúsculas del español que tienen longitud 4, donde el alfabeto  $\Sigma = \{A, B, C, D, E, F, \dots, Z\}$ , entonces  $L = \{CASA, COMO, MESA, CINE, CAMA, AZUL, ROJO, CAFÉ, UTIL, RAUL, JUAN, ROPA, AUTO, NADA, LUNA, TRES, OCHO, OTRO, SUBE, \dots\}$



La única restricción importante de un lenguaje es que todos los alfabetos  $\Sigma$  son finitos, de este modo, los lenguajes aunque pueden tener un número infinito de cadenas están formadas por los símbolos que definen un alfabeto finito y prefijado.

### EJERCICIOS.

1. Si  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$  proporcione 10 ejemplos de cadenas de  $\Sigma$
2. Si  $\Sigma = \{\&, \grave{,} ?, =, \#, (, ), <\}$  ¿Qué longitud tiene las siguientes cadenas?  
 a)  $|\#)\&\#?|=|$       b)  $|\<\grave{)}\&|$       c)  $|\&\&=( )=\&\&|$       d)  $|( )(\grave{)}\&=\#=?()(|$
3. Si  $\Sigma = \{A, B, C, D, E, F, \dots, Z\}$  y L es el lenguaje de todas las palabras en mayúsculas del español, proporcione 10 ejemplos de palabras de longitud 5
4. Si  $\Sigma = \{0, 1\}$  y L es el lenguaje de todas las cadenas que sean palíndromos, es decir que indiquen los mismos al leer de izquierda a derecha y también de derecha a izquierda. Proporcione 10 ejemplos de cadenas.
5. Si  $\Sigma = \{A, B, C, D, E, F, \dots, Z\}$  y L es el lenguaje de todas las palabras en mayúsculas del español, proporcione 10 ejemplos de palabras de longitud 4 que inicien con la letra M

## 2. Expresiones regulares.

Una expresión regular es una notación que sirve para describir patrones de cadenas de símbolos de un lenguaje L, por ejemplo si quisieramos describir el patrón de todas las cadenas o palabras del lenguaje español que inician con la letra *c* y la antepenúltima letra *i*, el conjunto de esas palabras es la siguiente  $\{cima, cine, cita, clima, canica, canción, celebración, \dots\}$ , sería conveniente expresar el patrón de todas las palabras mediante una expresión regular. En ocasiones describir un patrón de cadenas no es fácil porque las cadenas pueden iniciar o terminar con muchas alternativas de símbolos como ejemplo todas las cadenas de los numeros con punto decimal, son cadenas de símbolos que puede iniciar con  $+$  ó  $-$  ó  $.$  ó dígito, donde dígito es un símbolo de conjunto  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$  así en el caso que inicia con  $+$  ó  $-$  el siguiente símbolo puede ser  $.$  ó dígito, pero si inicia con  $.$  los siguientes símbolos solamente serán sítigos, en el caso de que inicie con dígito los siguientes símbolos deben ser dígitos y debe existir un  $.$  en cualquier posición. Ejemplo de

números con punto decimal son: +23.56, -450., 10.01, -.78, 2017.50, +.789 , por eso es conveniente utilizar las **expresiones regulares** que definen una notación estructural que permite describir todas las cadenas de ese lenguaje. La sintaxis de las expresiones regulares está compuestas por un alfabeto de símbolos  $\Sigma$ , los operadores  $*$   $+$   $|$  y símbolos  $()$  y  $[]$  para alterar la precedencia.

Los operadores en una expresión regular son a) La cerradura de Kleene que se representa por el símbolo  $*$ , b) La cerradura positiva que se representa por el símbolo  $+$ , c) La concatenación por el símbolo  $.$  y d) La disyunción por el símbolo  $|$  y el mayor orden de precedencia lo tiene el  $*$  y  $+$ , luego el  $.$  y menor el  $|$ , así también los parentesis  $()$  y los corchetes  $[]$  sirven para alterar la precedencia en una expresión regular. Por ejemplo, la expresión regular  $a|b.c^*$  primero se aplica el  $*$  luego el  $.$  y al último la  $|$ , esta expresión regular es equivalente a  $a|(b.(c^*))$  en muchas ocasiones el operador concatenación que es el punto se omite escribirlo y se escribe  $a|bc^*$  y su equivalente  $a|(b(c^*))$  que indica el conjunto de todas las cadenas del alfabeto  $\Sigma = \{a, b, c\}$  donde el lenguaje  $L = \{a, b, bc, bcc, bccc, \dots\}$ , sin embargo si cambiamos la precedencia a la expresión regular  $(a|b)c^*$  entonces ahora esta expresión regular tiene un nuevo lenguaje  $L = \{a, b, ac, bc, acc, bcc, accc, bccc, \dots\}$ .

Considerando de que los números enteros son todas las cadenas de símbolos donde el primer símbolo es  $+$  o  $-$  o ninguno de ellos y luego sigue uno o más dígitos entonces la expresión regular es  $(+|-)^?d^+$  donde el símbolo  $d$  representa los dígitos del 0 al 9. El lenguaje  $L$  formado por todas las cadenas es  $L = \{\dots -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \dots\}$  los números 1, 2, 3 ... también pueden representarse como  $+1, +2, +3, +4, +5, \dots$

Si bien, describir el patrón de todas las cadenas o palabras del lenguaje español que inician con la letra  $c$  y la antepenúltima letra  $i$ , el conjunto de esas palabras es la siguiente  $\{cima, cine, cita, clima, canica, canción, celebración, \dots\}$ , la expresión regular  $c\hat{o}^*i\hat{o}\hat{o}$  donde  $\hat{o} = \{a, \dots, z, A, \dots, Z\}$ , describe a todas las que inician con la letra  $c$ , luego  $\hat{o}^*$  indica que después de la letra  $c$  hay cero o más símbolos  $\hat{o}$  que son letras  $\{a, \dots, z, A, \dots, Z\}$ , después una letra  $i$  y los dos últimos símbolos son  $\hat{o}$ , en esta expresión hay cadenas que no tienen significado como  $ciaa, cizz, caiaa, cbibb$  y demás, para tal caso se necesita de un vocabulario de las cadenas de este patrón que tienen significado.

## EJERCICIOS.

1. Realice una expresión regular de todas las cadenas con símbolos  $a$  y  $b$ , que terminen con el sufijo  $abb$ . Ejemplo de éstas cadenas son:  $abb$ ,  $aabb$ ,  $babb$ ,  $aaabb$ ,  $ababb$ ,  $baabb$ ,  $bbabb$ , ...
2. Realice una expresión regular de todas las cadenas de con símbolos  $0$  y  $1$ , que primero tengan los símbolos  $1$ 's con longitud impar y después aparezcan los  $0$ 's con longitud par. Ejemplo de éstas cadenas son:  $100$ ,  $10000$ ,  $1000000$ ,  $11100$ ,  $1110000$ ,  $111110000$ , ...
3. Para la expresión regular  $(+|-)^?d^+.d^+$  indique las cadenas correctas de los siguientes incisos. (Nota. En esta expresión el  $.$  es un símbolo no el operador concatenación y  $d$  representa los dígitos del 0 al 9).  
a) -20.43    b) 0.3216    c) 329.    d) 217.92    e) +2019    f) +.762    g) -.4555

Las expresiones regulares son un lenguaje que permite simbolizar conjuntos de cadenas de texto formadas por la concatenación de otras cadenas. Es decir, permite buscar subcadenas de texto dentro de una cadena de texto.

La definición de un patrón de búsqueda de expresión regular se establece a través de un intrincado conjunto de axiomas de tipo matemático, (que por ahora, ni espero que nunca, entrare a detallar).

La idea más importante es que una expresión regular es un patrón de búsqueda en una cadena, es algo parecido a los caracteres comodines del sistema operativo. Por ejemplo, si queremos que el sistema nos muestre todos los archivos fuentes de C# podemos hacerlo a través del patrón `*.cs`. Esta es una forma de decirle al sistema operativo que muestre solo los archivos cuyo nombre termine en los caracteres `.cs`. Podríamos decir que para el sistema operativo la cadena `*.cs` es una expresión regular.

De forma parecida, si queremos buscar un determinado grupo de caracteres dentro de una cadena, escribiremos un patrón de búsqueda de expresión regular, y a continuación, (de alguna forma arcana e incomprensible□) pondremos en marcha el motor de búsqueda de expresiones regulares, le pasaremos la cadena y el patrón de búsqueda y nos devolverá una colección de objetos con todas las veces que ha encontrado ese patrón de búsqueda, o bien podemos interrogarle para ver si hay alguna coincidencia, etc.

A partir de aquí, y en lo que resta de este documento te sugiero que olvides todo lo que sabes sobre el significado de algunos caracteres especiales, tales como \* y +, y manejes únicamente el significado que aquí se describe.

Para comenzar, veamos algunas expresiones regulares básicas. Supongamos que tenemos un carácter 'a', entonces se representa:

- a Representa a la cadena formada por a
- a+ Representa todas las cadenas formadas por la concatenación de a tales como a, aa, aaa, .... etc. El calificador + indica que el elemento precedente (la letra a) puede aparecer una o más veces seguidas en la cadena
- a\* Representa Todas las cadenas formadas por la concatenación de a, incluyendo a la cadena vacía. Es decir, el calificador \* indica que el elemento precedente (la letra a) puede aparecer ninguna, una o más veces vez en la cadena
- a? El calificador ? indica que el elemento precedente (la letra a) puede aparecer ninguna, o una vez en la cadena Ejemplo:
- La expresión regular 01\* representa a todas las cadenas que empiezan por el carácter cero (0) seguido de ninguno o cualquier cantidad de unos. Aquí, están representadas cadenas como 0, 01, 011, 01111, etc.
- La expresión regular (ab)+c, representa todas las cadenas que repiten la cadena ab, una o más veces y terminan en el carácter c, tales como abc, ababc, abababc, ... etc. En este último ejemplo no se incluyen la cadena abcab, ni tampoco la cadena c.

Para escribir ese ejemplo necesitamos conocer algo más sobre caracteres especiales:

[ ] Los corchetes, permiten determinar una lista de caracteres, de los cuales se escogerá SOLAMENTE uno. Por ejemplo, [0123] pone a disposición cualquiera de estos dígitos para hacerlo coincidir con la cadena analizada.

( ) Los paréntesis pueden usarse para definir un grupo de caracteres sobre los que se aplicaran otros operadores. Permiten establecer alguna subcadena que se hará coincidir con la cadena analizada. Por ejemplo (01)\* representa a todas las cadenas que son una repetición de la subcadena 01, tales como 01, 0101, 010101,... etc.

| Una barra vertical separa las alternativas posibles. Por ejemplo, "(marrón|castaño)" quiere decir que se da por bueno si encuentra la palabra marrón o

castaño.

$\backslash A$  Establece que la coincidencia debe cumplirse desde el principio de la cadena  $\backslash Z$  Establece que la coincidencia debe establecerse hasta el final de la cadena

$\backslash w$  Representa a cualquier carácter que sea un carácter alfanumérico o el carácter de subrayado. También se puede representar como  $[a-zA-Zo-9]$

$\{N\}$  Las llaves son un elemento de repetición. Indica que el elemento que le antecede debe repetirse exactamente N veces, por ejemplo  $w\{3\}$  ,  $(w)\{3\}$  y  $[w]\{3\}$  representa (las tres) a la cadena *www*

### 3. Autómatas Finitos

Los autómatas finitos representan un modelo útil para desarrollar software por ejemplo para diseñar y probar el comportamiento de circuitos digitales, para el análisis léxico en los compiladores, para explorar textos o patrones en páginas web, para verificar sistemas de todo tipo que tengan un número finito de estados diferentes tales como protocolos de comunicaciones o protocolo para el intercambio seguro de información.

Un autómata finito está compuesto por  $\langle S, \Sigma, \delta, s_0, F \rangle$  donde:

$S$  es un conjunto finitos de estados

$\Sigma$  es el conjunto de símbolos del alfabeto de entrada, la cadena vacía  $\epsilon$  no pertenece a  $\Sigma$

$\delta$  es la función de transición que relaciona un estado  $x$  con un conjunto de estados siguiente a través de  $\Sigma \cup \{ \epsilon \}$

$s_0$  es el estado inicial,  $s_0$  pertenece a  $S$

$F$  es el conjunto de estados finales o de aceptación,  $F$  es un subconjunto de  $S$

Un autómata finito se puede representar mediante un diagrama de transición con las siguientes características: el conjunto  $S$  de estados es representado por nodos  $s_0, s_1, s_2, s_3, \dots$ , la función de transición  $\delta$  en un conjunto de relaciones de trayectorias desde un estado  $s_x$  hacia otros estados de  $S$  y es representado por aristas o flechas donde indica el símbolo o

$\epsilon$  de la relación, el estado inicial  $s_0$  es representado el nodo anteponiendo una flecha, los estados finales  $F$  son representados por doble círculo.

Los autómatas finitos se clasifican en Autómatas finitos no determinísticos (AFN) y Autómatas finitos determinísticos (AFD).

### 3.1 Autómata Finito No Determinístico

Un AFN es un modelo matemático que consiste de:

- 1.- Un conjunto de estados  $S$ .
- 2.- Un conjunto de símbolos de entrada  $\Sigma$ .
- 3.- Una función de transición  $\delta$  que corresponde pares estado-símbolo a conjuntos de estados.
- 4.- Un estado  $s_0$  que denota como el estado inicial.
- 5.- Un conjunto de estados  $F$  que denotan los estados de aceptación o finales.

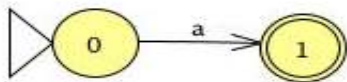
Un AFN no tiene restricciones para que exista más de una transición con el mismo nombre a diferentes estados, por lo que en una representación tabular no es posible determinar de manera única el estado destino para un símbolo determinado. No podemos determinar a qué estado se avanza del estado 0 con el símbolo  $a$ ; puede ser al propio 0 o al 1 pero ¿bajo qué condición? No está determinado. Incluso podría existir transiciones sin ninguna restricción, que se denominan transiciones epsilon o vacías porque no requieren un símbolo determinado ocurra para realizarse. Nuevamente, esto se vuelve indeterminado en una representación tabular.

Una expresión regular se puede representar de forma gráfica mediante un diagrama de transición.

Un **diagrama de transición** es una representación gráfica donde se tiene un conjunto de estados, los cuales pueden ser: iniciales, finales e intermedios: los cuales pueden tener una o más salidas hacia otro estado. Los estados se dibujan como círculos y se relacionan entre

sí con flechas con una etiqueta (el símbolo o conjunto de símbolos que provoca la transición de un estado a otro). Un estado final se representa con doble círculo y también recibe el nombre de estado de aceptación.

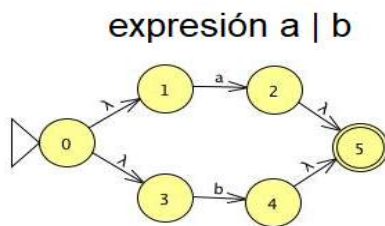
Para el símbolo **a**, la expresión regular es **a**, y el diagrama de transición es



El estado inicial es el estado 0 que está indicado con un triángulo y al ocurrir el símbolo **a** en el diagrama de transición indica que hay una transición del estado 0 al estado 1 mediante la etiqueta con el símbolo **a**. El estado 1 es final o de aceptación y está con doble círculo.

Los diagramas de transición mostradas en esta sección fueron creadas en el software gratuito JFLAP que se puede descargar de [www.jflap.org](http://www.jflap.org).

Para la expresión regular **a | b** el diagrama de transición es

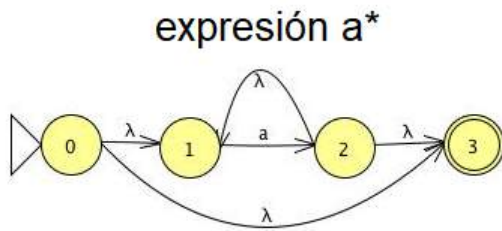


En este diagrama de transición existen 6 estados numerados del 0 al 5, El estado 0 es inicial y el estado 5 es final, las etiquetas con  $\lambda$  son transiciones vacías lo que indica que del estado 0 al estado 1 hay una transición libre con ningún símbolo, lo mismo ocurre con las transiciones  $0 \rightarrow 3$ ,  $2 \rightarrow 5$ ,  $4 \rightarrow 5$ . Por ejemplo, si ocurriera un símbolo **a**, las transiciones serían las siguientes  $0 \rightarrow 1 \rightarrow 2 \rightarrow 5$  y para el símbolo **b**, las transiciones serían las siguientes  $0 \rightarrow 3 \rightarrow 4 \rightarrow 5$ .

Esta representación del diagrama de transición se le conoce como la Construcción de Thompson, creador del algoritmo Construcción de Thompson que construye un diagrama de transición a partir de una expresión regular. Ken Thompson es conocido por ser el creador del lenguaje C y el sistema operativo Unix junto con Dennis Ritchie.

El **operador**  $*$  se le llama la cerradura vacío o de Kleene que indica la repetición de cero o más veces el símbolo que le antecede.

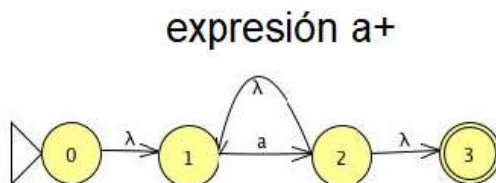
Por ejemplo la expresión regular  $a^*$  indica repetir cero o más veces el símbolo  $a$ , la expresión  $a^*$  es un conjunto de cadenas con el símbolo  $a$  que contiene los elementos  $a^* = \{ \epsilon, a, aa, aaa, aaa, aaaa, aaaaa, \dots \}$  y el diegrama de transición es el siguiente



El diagrama de transición esta compuesto por el conjunto de estados  $\{0, 1, 2, 3\}$  donde el estado 0 es el estado inicial y el estado final el 3, Como  $a^* = \{ \epsilon, a, aa, aaa, aaa, aaaa, aaaaa, \dots \}$  entonces para el caso de cero ocurrencia del símbolo  $a$  es con la transición  $0 \rightarrow 3$ , para 1 ocurrencia de  $a$  es con las transiciones  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ , para 2 ocurrencias de  $a$  es con las transiciones  $0 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 2 \rightarrow 3$ , para 3 ocurrencias de  $a$  es con las transiciones  $0 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 2 \rightarrow 3$ , y así sucesivamente para las siguientes ocurrencias de  $a$ .

El **operador**  $^+$  se le llama la cerradura positiva que indica la repetición de uno o más veces el símbolo que le antecede.

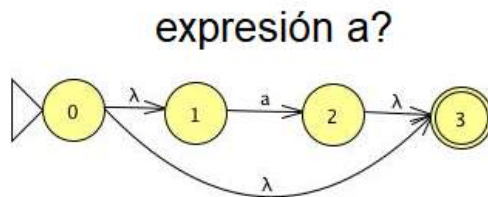
Por ejemplo la expresión regular  $a^+$  indica repetir uno o más veces el símbolo  $a$ , la expresión  $a^+$  es un conjunto de cadenas con el símbolo  $a$  que contiene los elementos  $a^+ = \{ a, aa, aaa, aaa, aaaa, aaaaa, \dots \}$  y el diegrama de transición es el siguiente





El **operador**  $^?$  se le operador cero o una vez el símbolo que le antecede.

Por ejemplo la expresión regular  $a^?$  indica cero o una vez el símbolo  $a$ , la expresión regular es el conjunto  $a^? = \{ \epsilon, a \}$  y el diegrama de transición es el siguiente



### 3.2 Autómata Finito Determinístico

Un AFD es un caso especial de AFN en el que ningún estado tiene transiciones para diversos estados bajo el mismo símbolo y no se permiten transiciones vacías. Los diagramas de transición son autómatas determinísticos. que podría ser muy aproximado al diagrama de transición que construiríamos para la expresión.

El término “determinista” hace referencia al hecho de que para cada entrada sólo existe uno y sólo un estado al que el autómata puede hacer la transición a partir de su estado actual.

Un autómata finito determinístico AFD es un modelo matemático que consiste de cinco componentes  $A = (S, \Sigma, \delta, s_0, F)$

Donde A es el nombre del AFD

- 1.- Un conjunto finito de estados  $S$ .
- 2.- Un conjunto de símbolos de entrada  $\Sigma$ .
- 3.- Una función de transición  $\delta$  que corresponde pares estado-símbolo a conjuntos de estados.
- 4.- Un estado  $s_0$  que denota como el estado inicial.
- 5.- Un conjunto de estados  $F$  que denotan los estados de aceptación o finales.

Para llegar de la expresión regular al autómata determinístico se emplea el algoritmo de construcción de Thompson que permite hacer la transformación y preparar la construcción del analizador de léxico.

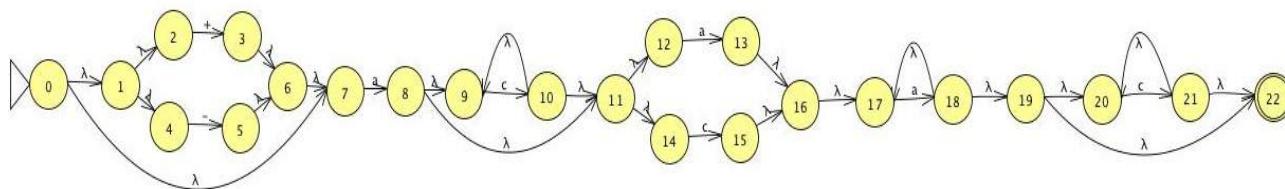
### 3.3 Conversión de una expresión regular a un AFD

Para convertir una expresión regular a un autómata finito deterministas AFD se puede realizar mediante la aplicación de dos algoritmos donde el primer algoritmo es que a partir de una expresión regular se construya un autómata finito no determinístico AFN, este algoritmo es llamado Construcción de Thompson y expone que para cada expresión regular primitiva como lo es la cerradura de Kleene, la cerradura positiva, cero o más apariciones, disyunción y concatenación de forme un AFN siguiendo un modelo o plantilla para cada expresión regular. El segundo algoritmo es el de Construcción de Subconjuntos y expresa que a partir del AFN resultante en el algoritmo de Construcción de Thompson, se debe realizar la creación de subconjuntos para crear la tabla de transiciones TranD que es el AFD resultante.

**Ejemplo.** Con la Expresión Regular  $(+|-)^2 ac^* (a|c) a^+ c^2$

- Utilice la Construcción de Thompson para desarrollar el AFN
- Utilice la Construcción de Subconjuntos para desarrollar el AFD

De acuerdo a la Construcción de Thompson el Autómata Finito No determinístico (AFN) queda



El alfabeto lo integran los símbolos  $+ - a c$  por lo tanto  $\Sigma = \{+, -, a, c\}$

Para de desarrollar el algoritmo de la Construcción de Subconjuntos, primero se aplica la función de la cerradura vacía  $\_cerradura$  del estado inicial que es el estado cero y al conjunto de estados resultante se establece como conjunto A, así en este ejemplo queda:

$$\varepsilon\_cerradura(0) = \{0, 1, 2, 4, 7\} = A$$

Posteriormente se crea la tabla de transición Dtran considerando la entrada que es el AFN, el alfabeto y el primer conjunto A. Esto es, que con el conjunto A y para cada símbolo del alfabeto se aplica la función mover(A,+) para el caso del símbolo + y a los estados resultantes se aplica la función  $\varepsilon\_cerradura$ , en otras palabras la aplicación de ambas funciones queda  $\varepsilon\_cerradura(mover(A,+))$ .

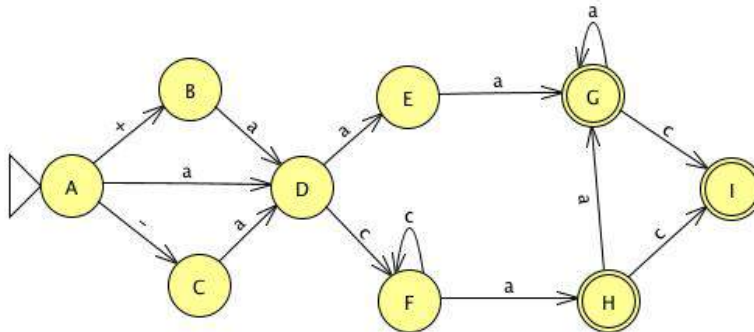
$$\begin{aligned} Dtran[A,+] &= \varepsilon\_cerradura(mover(A,+)) = \varepsilon\_cerradura(3) = \{3, 6, 7\} = B \\ Dtran[A,-] &= \varepsilon\_cerradura(mover(A,-)) = \varepsilon\_cerradura(5) = \{5, 6, 7\} = C \\ Dtran[A,a] &= \varepsilon\_cerradura(mover(A,a)) = \varepsilon\_cerradura(8) = \{8, 9, 11, 12, 14\} = D \\ Dtran[A,c] &= \varepsilon\_cerradura(mover(A,c)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[B,+] &= \varepsilon\_cerradura(mover(B,+)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[B,-] &= \varepsilon\_cerradura(mover(B,-)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[B,a] &= \varepsilon\_cerradura(mover(B,a)) = \varepsilon\_cerradura(8) = D \\ Dtran[B,c] &= \varepsilon\_cerradura(mover(B,c)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[C,+] &= \varepsilon\_cerradura(mover(C,+)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[C,-] &= \varepsilon\_cerradura(mover(C,-)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[C,a] &= \varepsilon\_cerradura(mover(C,a)) = \varepsilon\_cerradura(8) = D \\ Dtran[C,c] &= \varepsilon\_cerradura(mover(C,c)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[D,+] &= \varepsilon\_cerradura(mover(D,+)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[D,-] &= \varepsilon\_cerradura(mover(D,-)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[D,a] &= \varepsilon\_cerradura(mover(D,a)) = \varepsilon\_cerradura(13) = \{13, 16, 17\} = E \\ Dtran[D,c] &= \varepsilon\_cerradura(mover(D,c)) = \varepsilon\_cerradura(10, 15) = \{9, 10, 11, 12, 14, 15, 16, 17\} = F \\ Dtran[E,+] &= \varepsilon\_cerradura(mover(E,+)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[E,-] &= \varepsilon\_cerradura(mover(E,-)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[E,a] &= \varepsilon\_cerradura(mover(E,a)) = \varepsilon\_cerradura(18) = \{17, 18, 19, 20, 22\} = G \\ Dtran[E,c] &= \varepsilon\_cerradura(mover(E,c)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[F,+] &= \varepsilon\_cerradura(mover(F,+)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[F,-] &= \varepsilon\_cerradura(mover(F,-)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[F,a] &= \varepsilon\_cerradura(mover(F,a)) = \varepsilon\_cerradura(13, 18) = \{13, 16, 17, 18, 19, 20, 22\} = H \\ Dtran[F,c] &= \varepsilon\_cerradura(mover(F,c)) = \varepsilon\_cerradura(\emptyset) = (10, 15) = F \\ Dtran[G,+] &= \varepsilon\_cerradura(mover(G,+)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[G,-] &= \varepsilon\_cerradura(mover(G,-)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[G,a] &= \varepsilon\_cerradura(mover(G,a)) = \varepsilon\_cerradura(18) = G \\ Dtran[G,c] &= \varepsilon\_cerradura(mover(G,c)) = \varepsilon\_cerradura(21) = (10, 15) = I \\ Dtran[H,+] &= \varepsilon\_cerradura(mover(H,+)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[H,-] &= \varepsilon\_cerradura(mover(H,-)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[H,a] &= \varepsilon\_cerradura(mover(H,a)) = \varepsilon\_cerradura(18) = G \\ Dtran[H,c] &= \varepsilon\_cerradura(mover(H,c)) = \varepsilon\_cerradura(21) = (10, 15) = I \\ Dtran[I,+] &= \varepsilon\_cerradura(mover(I,+)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[I,-] &= \varepsilon\_cerradura(mover(I,-)) = \varepsilon\_cerradura(\emptyset) = \emptyset \\ Dtran[I,a] &= \varepsilon\_cerradura(mover(I,a)) = \varepsilon\_cerradura(\emptyset) = \emptyset \end{aligned}$$

$$Dtran[I,c] = \varepsilon\_cerradura(mover(I,c)) = \varepsilon\_cerradura(\emptyset) = \emptyset$$

La tabla de transición Dtran es la siguiente

Dtran	+	-	a	c
A	B	C	D	—
B	—	—	D	—
C	—	—	D	—
D	—	—	E	F
E	—	—	<b>G</b>	—
F	—	—	<b>H</b>	F
<b>G</b>	—	—	<b>G</b>	<b>I</b>
<b>H</b>	—	—	<b>G</b>	<b>I</b>
<b>I</b>	—	—	—	—

De acuerdo a la tabla de transición Dtran el Autómata Finito Determinístico (AFD) de la expresión regular  $(+|-)^? ac^* (a|c) a^+ c^?$ , el diagrama de transición del AFD se presenta en la siguiente imagen.



### EJERCICIOS.

- Una cadena de números enteros puede iniciar con el símbolo + o el – o ninguno de ellos y luego continua una subcadena formada por uno o más dígitos y se puede representar mediante la expresión regular  $(+|-)^? d^+$  donde el símbolo  $d$  representa los dígitos del 0 al 9.
  - Contruya el AFN de la expresión regular mediante la Construcción de Thompson.
  - Contruya el AFD partiendo del AFN realizado en el inciso anterior para lo cual deberá crear la tabla de transiciones Dtran y el diagrama de transición del AFD.
- Considerando que un identificador es una cadena de símbolos que inicia con un letra y seguido de cero o más símbolos como letras, dígitos o guión bajo ( \_ ), la

expresión regular de un identificador queda  $l(l|d|_ )^*$  donde el símbolo  $l$  representa las letras de  $a-z$  o  $A-Z$  y el símbolo  $d$  representa los dígitos del 0 al 9.

- a) Contruya el AFN de la expresión regular mediante la Construcción de Thompson.
  - b) Contruya el AFD partiendo del AFN realizado en el inciso anterior para lo cual deberá crear la tabla de transiciones Dtran y el diagrama de transición del AFD.
3. La expresión regular de números con punto decimal con signo es la siguiente  $(+|-)^?[(d^*.d^+) | (d^+.d^*)]$  donde el símbolo  $d$  representa los dígitos del 0 al 9.
- a) Contruya el AFN de la expresión regular mediante la Construcción de Thompson.
  - b) Contruya el AFD partiendo del AFN realizado en el inciso anterior para lo cual deberá crear la tabla de transiciones Dtran y el diagrama de transición del AFD.

#### 4. Gramáticas

Una gramática en la teoría de la computación describe en forma natural la estructura jerárquica de la mayoría de las instrucciones de un lenguaje de programación y se define como  $G = \langle T, N, S, P \rangle$  donde

$T$  es un conjunto de símbolos Terminales que normalmente se les llama tokens y representan los símbolos elementales del lenguaje definido por la gramática.

$N$  es un conjunto de símbolos No terminales a los que a veces se les llama variables sintácticas.

$P$  es un conjunto de producciones, en donde cada producción consiste en un no terminal llamado encabezado o lado izquierdo de la producción, una flecha y una secuencia de terminales y no terminales llamada cuerpo o lado derecho de la producción. Ejemplo  $A \rightarrow aBbc$  es una producción y el lado izquierdo es un no terminal, la flecha se lee “produce” y del lado derecho de la producción es una secuencia de terminales y/o no terminales, en este caso los símbolos  $a$ ,  $b$  y  $c$  son terminales (por minúsculas) y  $B$  es un no terminal. En este caso se puede entender que  $A$  se transforma en  $aBbc$ .

$S$  es el símbolo inicial

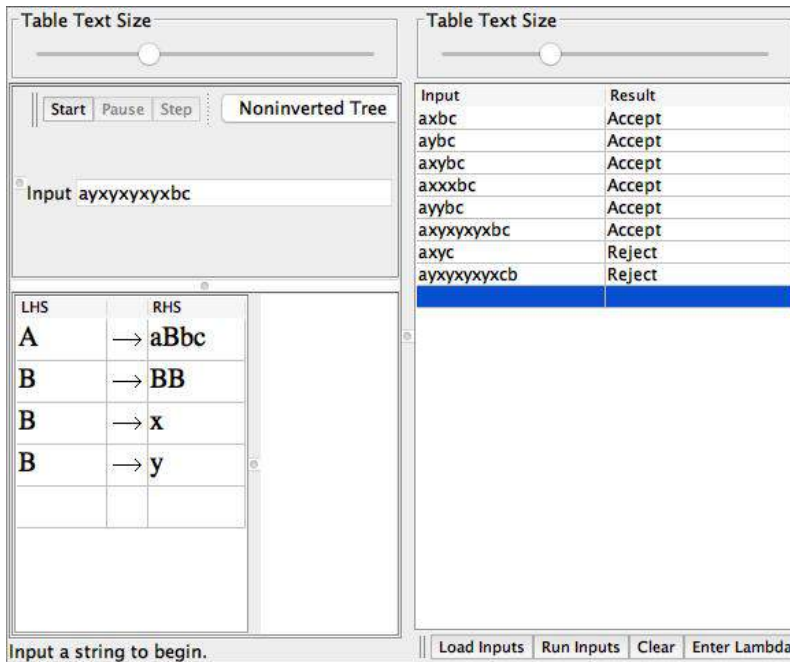
Para especificar una gramática se utiliza una notación de uso popular llamada Notación de Backus-Naur (BNF), ejemplo:

$$A \rightarrow aBbc$$

$$B \rightarrow BB \mid x \mid y$$

Los símbolos terminales  $T=\{a, b, c, x, y\}$ , los símbolos no terminales  $N=\{A, B\}$ , las producciones  $P=\{A \rightarrow aBbc, B \rightarrow BB, B \rightarrow x, B \rightarrow y\}$  y el símbolo inicial es  $S=A$ .

Es una gramática que representa todas las cadenas del alfabeto  $\Sigma = \{a, b, c, x, y\}$ , que inician con el símbolo  $a$  luego una o más combinaciones de los símbolos  $x$  e  $y$ , después terminan con  $bc$ . Ejemplo de cadenas válidas son  $axbc$ ,  $aybc$ ,  $axybc$ ,  $axxxbc$ ,  $ayybc$ ,  $axyxyxyxyc$  y de cadenas inválidas  $axyc$ ,  $axyxyxyxycb$ . Como se puede apreciar en la siguiente imagen de JPLAP



## 4.1 Derivaciones y Árbol de Análisis Sintáctico

En una gramática, las **derivaciones** son sustituciones que empiezan con el símbolo inicial  $S$  y se van sustituyendo en forma repetida un no terminal mediante el cuerpo de una producción para ese lenguaje. La gramática del ejemplo anterior es:

$$\begin{aligned} A &\rightarrow aBbc \\ B &\rightarrow BB \mid x \mid y \end{aligned}$$

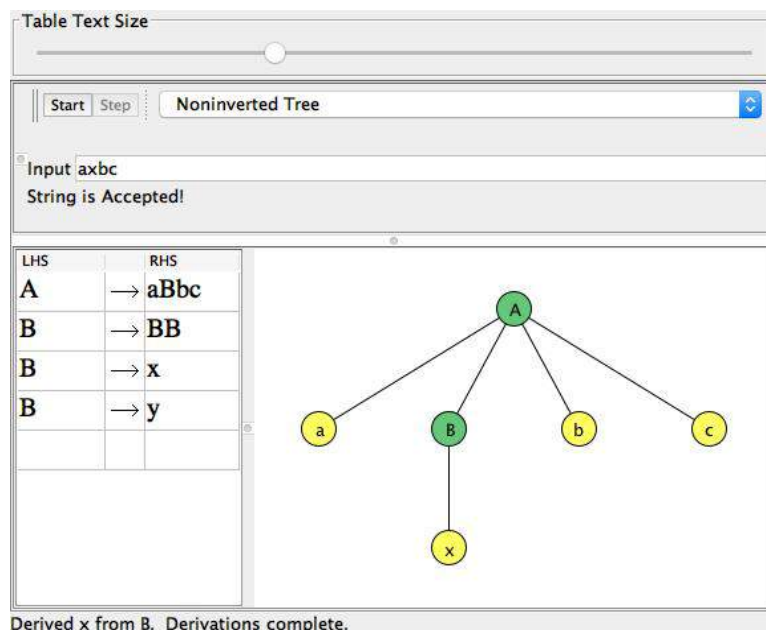
Para demostrar que las cadenas  $axbc$ ,  $aybc$  son váidas o pertenecen al lenguaje se realizan las siguientes derivaciones:

$$A \rightarrow a\underline{B}bc \rightarrow axbc$$

$$A \rightarrow a\underline{B}bc \rightarrow aybc$$

El símbolo  $\rightarrow$  se lee “deriva” por lo que  $A \rightarrow a\underline{B}bc$  dice  $A$  deriva  $a\underline{B}bc$  luego el subrayado en  $\underline{B}$  indica que el no terminal  $B$  se sustituye por el cuerpo  $axbc$  y termina el proceso de derivación. De igual manera para la cadena  $aybc$ .

Un **árbol de análisis sintáctico** muestra en forma gráfica, la manera en que el símbolo inicial de una gramática deriva a una cadena en el lenguaje. El árbol de análisis sintáctico para la cadena  $axbc$  se muestra en la figura, los círculos de color verde son los símbolos no terminales y los círculos de color amarillo los símbolos terminales.



Para la cadena  $axyxyc$  las derivaciones se pueden realizar por la izquierda lo que significa que en cada derivación se elige al símbolo no terminal más a la izquierda para realizar la sustitución o también se pueden realizar derivaciones por la derecha lo que significa que en cada derivación se elige al símbolo no terminal más a la derecha para realizar la sustitución.

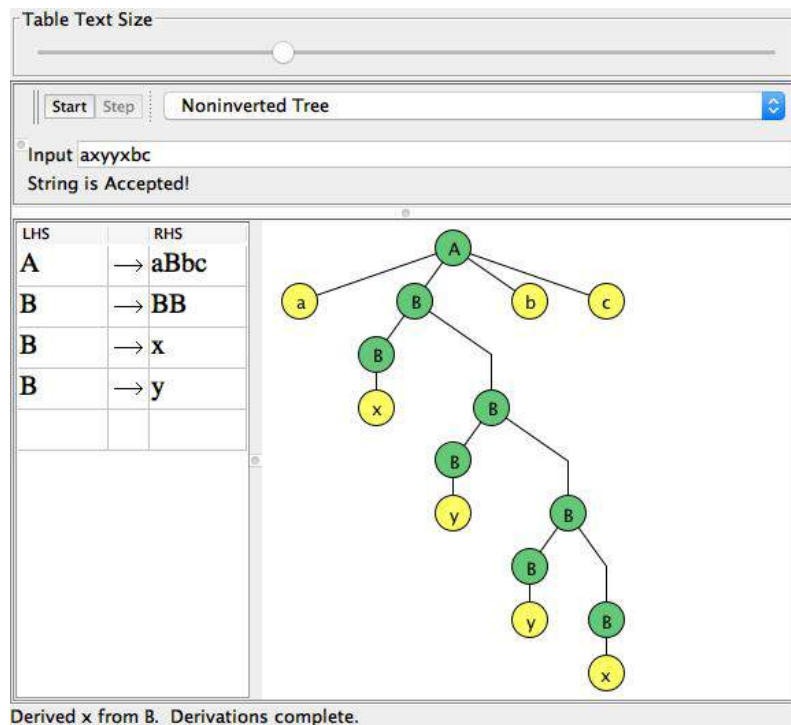
Derivaciones por la izquierda para validar la cadena  $axyxyc$ .

$A \Rightarrow aBbc \Rightarrow aBBbc \Rightarrow axBbc \Rightarrow axBBbc \Rightarrow axyBbc \Rightarrow axyBBbc \Rightarrow axyyBbc \Rightarrow axyyxyc$

Derivaciones por la derecha para validar la cadena  $axyxyc$ .

$A \Rightarrow aBbc \Rightarrow aBBbc \Rightarrow aBxyc \Rightarrow aBBxyc \Rightarrow aByxyc \Rightarrow aBByxyc \Rightarrow aByyxyc \Rightarrow axyyxyc$

El árbol de análisis sintáctico es el mismo en ambas derivaciones como se muestra en la siguiente imagen.





## 4.2 Generadores de Analizadores Sintácticos (Parser)

Generadores de analizadores sintácticos (parsers), que producen de manera automática analizadores sintácticos a partir de una descripción gramatical de un lenguaje de programación (Aho, 2008).

Ejemplo de herramientas generadoras de analizadores sintáctico son:

**Yacc.** Tipo de analizador: Ascendente, LALR(1). Código generado: C, C++.

Características adicionales:

- Se puede integrar con Lex dejando a éste el análisis léxico.
- La precedencia se puede definir al margen de la gramática, manteniendo ésta más simple.
- Conjuntamente con Memphis se puede construir un árbol sintáctico como salida del analizador.

**Bison.** Tipo de analizador: Ascendente, LALR(1). Código generado: C, parece que hay una versión para Eiffel que no he encontrado.

Características adicionales:

- Compatible con Yacc: una gramática de entrada para Yacc puede utilizarse en Bison sin ninguna modificación.

**Grammatica.** Tipo de analizador: Descendente, LL(k). Código generado: C#, Java.

Características adicionales:

- Soporte para depurar las gramáticas sin necesidad de generar el analizador.
- Genera código legible y comentado.
- Mensajes de error detallados durante el análisis.

**Sid.** Tipo de analizador: Descendente, LL(1). Código generado: C.

Características adicionales:

- Transforma la gramática a una gramática LL(1) si puede, eliminando recursividad por la izquierda y factorizando.

**YaYacc.** Tipo de analizador: Ascendente, LALR(1). Código generado: C++.

Características adicionales:

- El generador sólo corre sobre FreeBSD, pero el código generado no está ligado a ninguna plataforma concreta.

**Gold Parser Builder.** Tipo de analizador: Ascendente (LALR(1)). Código generado: Multilenguaje (Java, C#, ANSI C, Delphi, Python, VB, VB .NET, VC++, wxWidgets, todos los lenguajes .NET, todos los lenguajes ActiveX).

Características adicionales:

- Incluye análisis léxico.
- El código fuente está disponible también en numerosos lenguajes.

**Byacc/Java.** Tipo de analizador: Ascendente, LALR(1). Código generado: Java.

**COCO/R.** Tipo de analizador: Descendente, LL(k). Código generado: C#, Java, Oberon, Pascal, Modula-2, C, C++, Delphi, Unicon.

Características adicionales:

- Incluye el analizador léxico.

### **TP Lex/Yacc**

Tipo de analizador: Ascendente, LALR(1).

Código generado: Turbo y Borland Pascal, Delphi.

**Aflex/Ayacc.** Tipo de analizador: Ascendente, LALR(1). Código generado: Ada.

Características adicionales:

- Basados en flex (léxico) y yacc (sintáctico).

**AntLR.** Tipo de analizador: Descendente recursivo, LL(k). Código generado: Java, C++, C#.

Características adicionales:

- Construcción de ASTs.
- Hay abundante documentación en la página, incluyendo una guía en castellano.

**JavaCC.** Tipo de analizador: Descendente recursivo, LL(k). Código generado: Java.

Características adicionales:

- Dispone de JJTree para construir el árbol.
- JJDoc toma una gramática y genera documentación HTML al estilo javaDoc.

- La especificación léxica puede hacerse en la misma gramática.

**SableCC.** Tipo de analizador: LALR(1). Código generado: Java.

Características adicionales:

- Es un framework para generar tanto compiladores como intérpretes.
- Genera además del parser, tree walkers y ASTs (tipados).
- Analizador léxico basado en DFAs, que soporta unicode.
- La gramática se especifica en EBNF.
- En la página hay abundante documentación y ejemplo de proyectos donde se usa SableCC.

**AnaGram.** Tipo de analizador: LALR(1). Código generado: C, C++.

**LISA.** Tipo de analizador: LL(k). Código generado: C++.

Características adicionales:

- Incluye analizador léxico.

**CUP.** Tipo de analizador: Ascendente, LALR(1). Código generado: Java.

Características adicionales:

- La precedencia puede especificarse al margen de la gramática.

**Spirit.** Tipo de analizador: Descendente, LL(k). Código generado: C++.

Características adicionales:

- EBNF.

**ParseView.** Tipo de analizador: Descendente, LL(k). Código generado: Java.

Características adicionales:

- Es un visualizador basado en antlr.

**Oops.** Tipo de analizador: Descendente, LL(1).

Código generado: Java.

Características adicionales:

- Se puede integrar con constructores de ASTs.
- Requiere un analizador léxico.
- La gramática se especifica en notación EBNF.

**Beaver.** Tipo de analizador: LALR(1).

Código generado: Java.

Características adicionales:

- EBNF.
- Necesita un analizador léxico.

### 4.3 Gold Parser Builder.

GOLD Parser Builder es una herramienta desarrollada por Devin Cook, que implementa una gramática mediante el análisis sintáctico ascendentes que utiliza gramáticas LALR (Look Ahead Left to Right- Rightmost derivation); se pueden construir proyectos con base de analizadores, o construir versiones propias de un lenguaje de programación.

Gold Parser Builder está disponible en <http://goldparser.org> con la versión 5.2

Tabulador {HT} Conocido por Tab o la tecla



Salto de Línea {LF} Despliega una nueva línea

Tabulación Vertical {VT} Salta a la siguiente línea marcada

Limpiar pantalla {FF} Limpia toda la pantalla

Retorno de carro {CR} Retorna el apuntador hacia el inicio de la línea actual

Espacio {Space} Indica un espacio

Espacio sin Romper {NBSP} Indica un espacio donde el salto de línea no es permitido

Símbolo del Euro {Euro Sign} Representa el símbolo del euro

Conjunto de caracteres comunes

Dígito {Number} {Digit}

Dígitos del 0 al 9

Letra {Letter} Letra del alfabeto en minúscula o mayúscula excepto ñ o Ñ  
 Alfanumérico {AlphaNumeric} Letras y dígitos  
 Cualquier carácter {Printable} #32 - #127 y #160  
 Letra extendida {Letter Extended} Todas las letras incluyendo ñ o Ñ  
 Cualquier carácter extendido {Printable Extended} Cualquier carácter después de #127  
 Espacio en blanco {Whitespace} caracteres como espacio en blanco, salto de líneas, tabulado.

**Ejemplo.** Para realizar un software que calcule las operaciones aritméticas de sumar, restar, multiplicar, dividir y cambie las prioridades utilizando los paréntesis izquierdo y derecho se define la siguiente gramática:

$E \rightarrow E + T \mid E - T \mid T$   
 $T \rightarrow T * F \mid T / F \mid F$   
 $F \rightarrow ( E ) \mid \text{num}$

Una gramática está compuesta por cuatro componentes  $G = \langle N, T, P, S \rangle$

Los símbolos No terminales son  $N = \{E, T, F\}$

Los símbolos Terminales son  $T = \{+, -, *, /, (, ), \text{num}\}$

Las producciones son

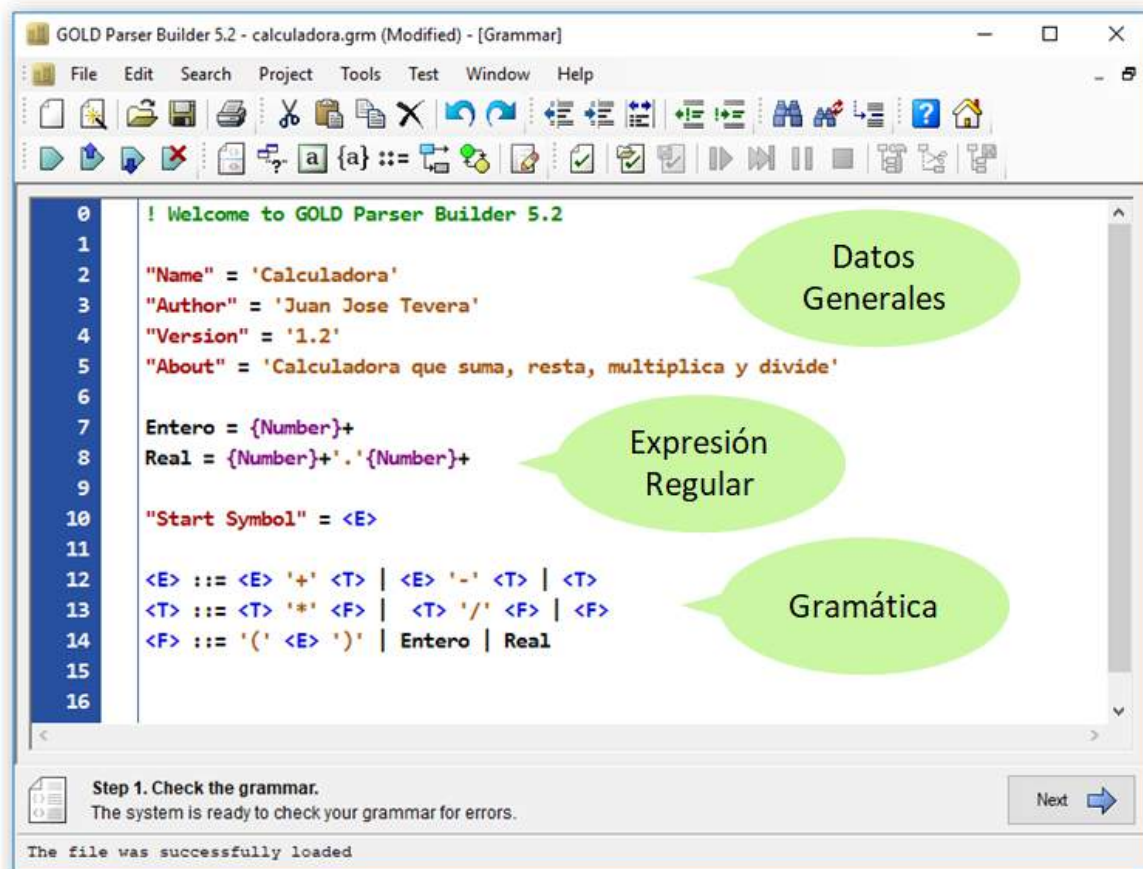
$P = \{E \rightarrow E+T, E \rightarrow E-T, E \rightarrow T, T \rightarrow T * F, T \rightarrow T / F, T \rightarrow F, F \rightarrow (E), F \rightarrow \text{num}\}$

El símbolo inicial  $S = E$

Para desarrollar el software en C# utilizando Gold Parser Builder y Visual Studio se realizar en los siguientes pasos:

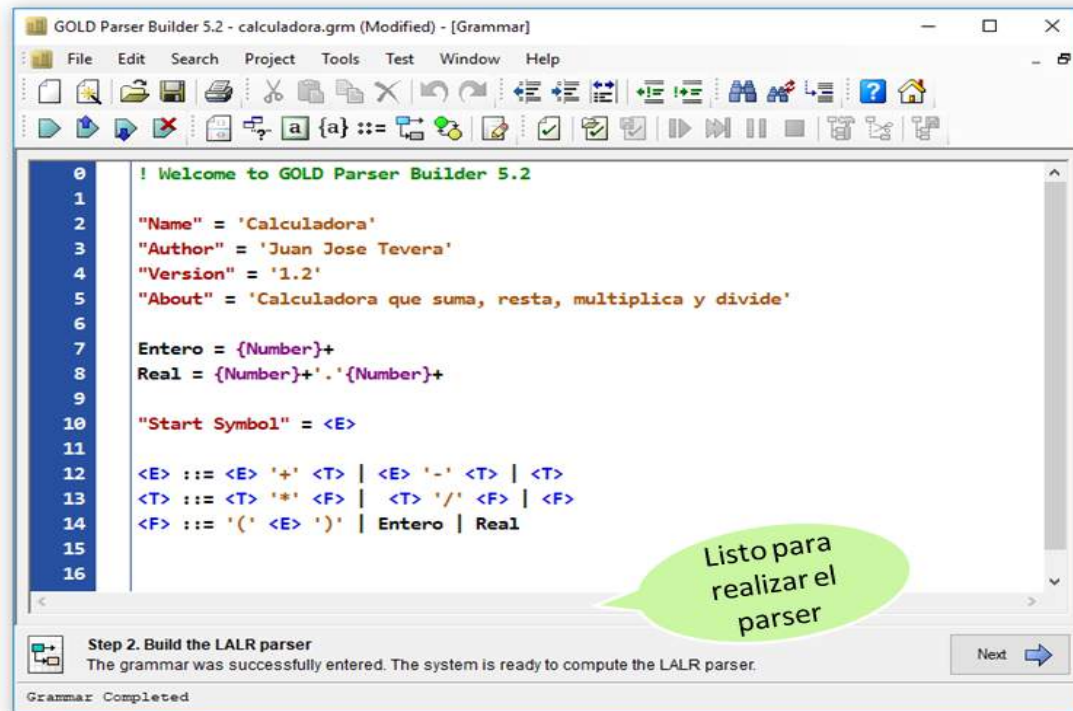
1. Definir la gramática en Gold Parser Builder y generar los archivos con extensión grm, cgt y cs.
2. Crear una aplicación en visual studio utilizando el lenguaje C#

#### 4.4 Ejemplo en Gold Parser Builder.



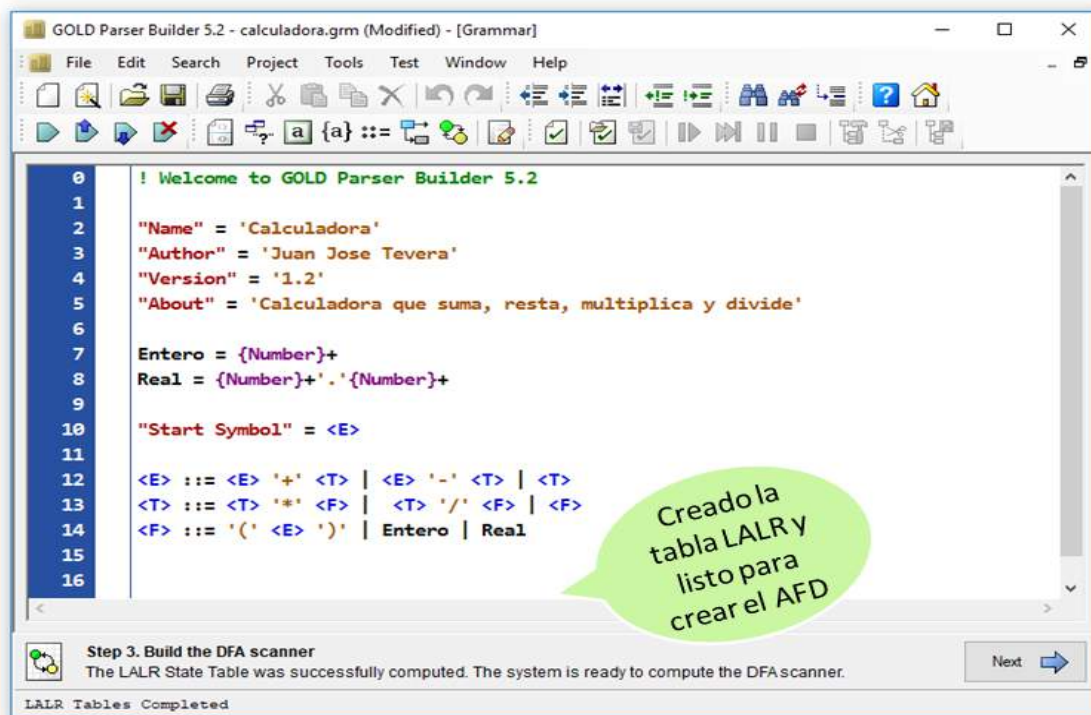
Fuente: Imagen del autor.

Si en los datos generales, las expresiones regulares y la gramática no contiene errores oprimir Next para avanzar al paso 2 sino habrá que corregir los datos.



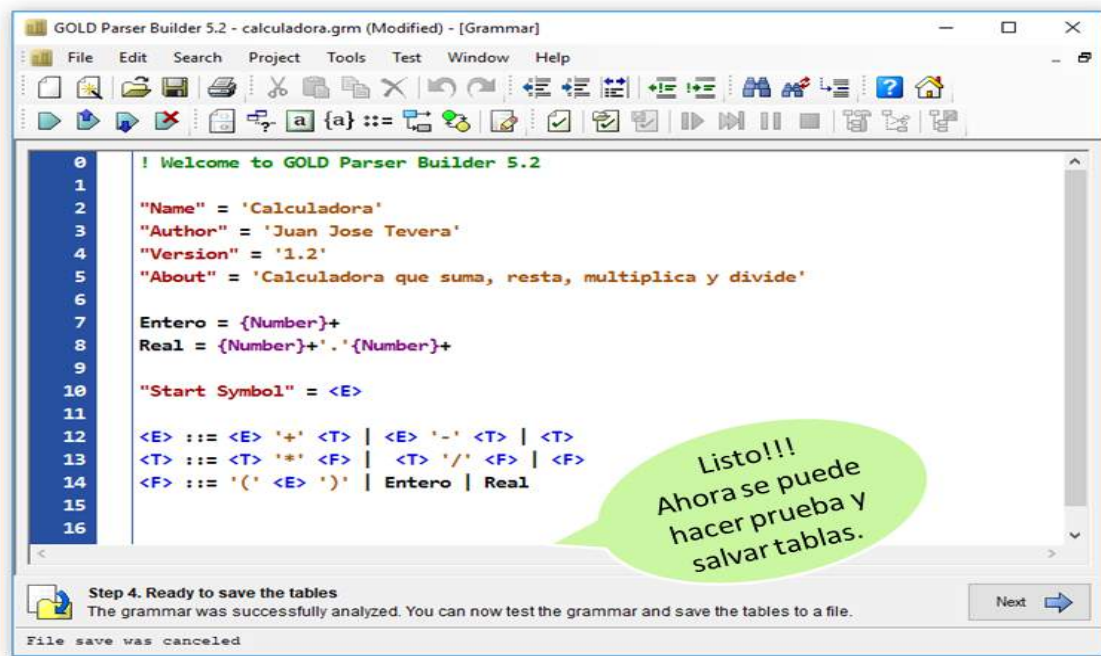
Fuente: Imagen del autor.

En el paso 2 Gold Parser Builder ha identificado correcta la gramática y puede crear el parser, nuevamente oprimir Next para avanzar al paso 3.



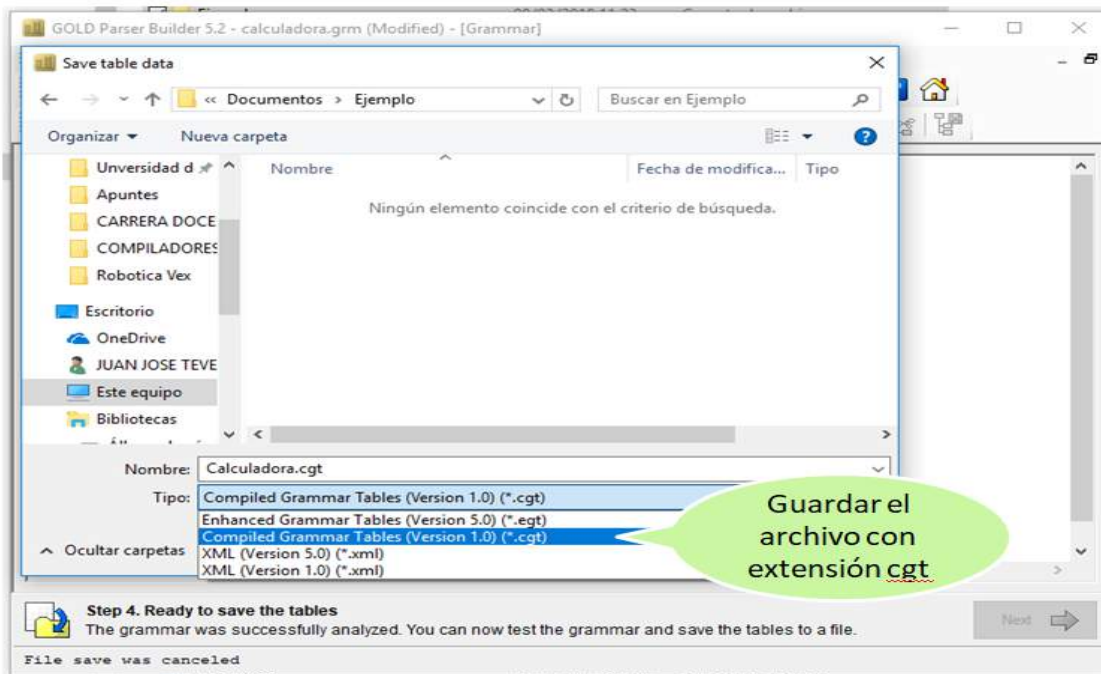
Fuente: Imagen del autor.

El paso 3 ha creado la tabla de análisis sintáctico ascendente LALR y en el paso 4 se creará la tabla del AFD, nuevamente se prime Next para avanzar al paso 4.



Fuente: Imagen del autor.

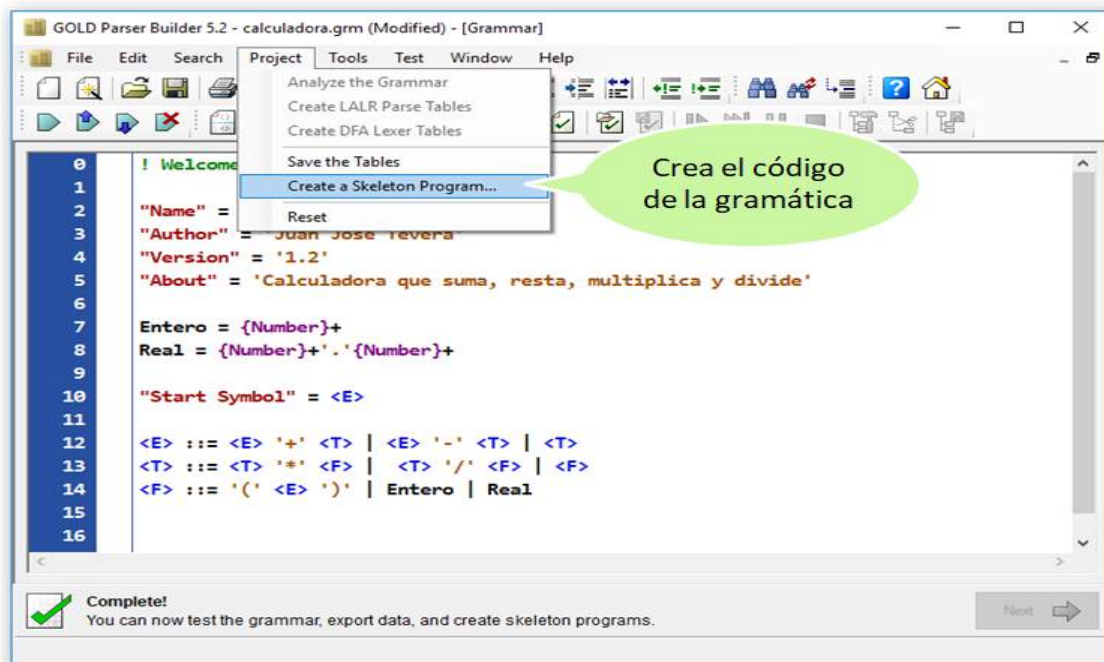
Al oprimir Next se despliega un cuadro de dialogo para guardar las tablas LALR y AFD, anotar el nombre del archivo con extensión **cgt**.



Fuente: Imagen del autor.

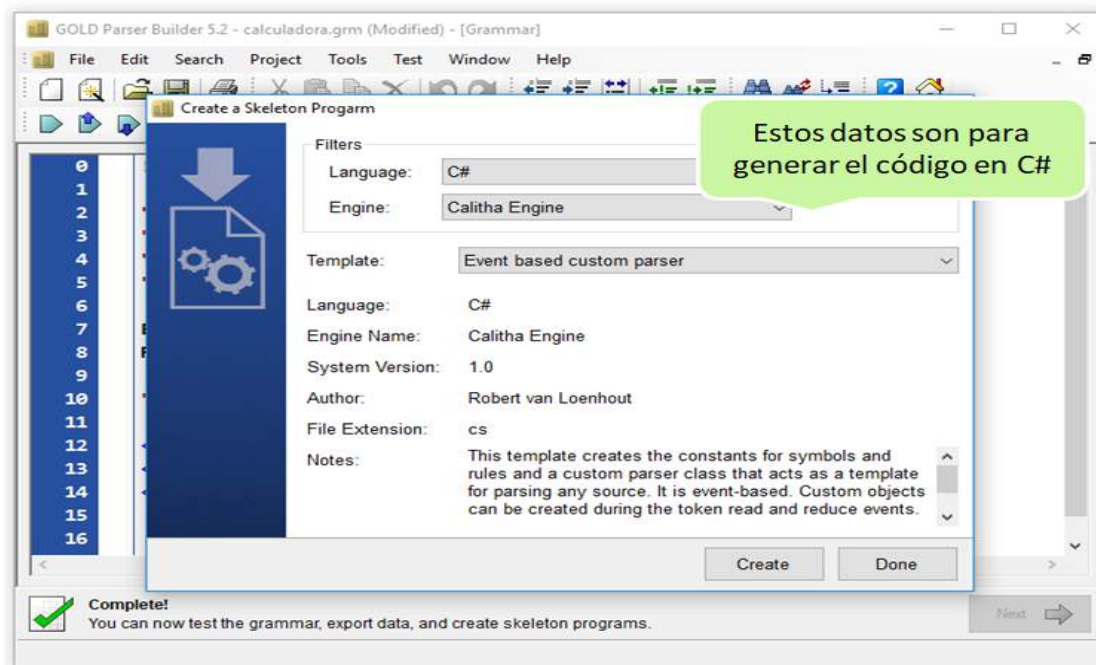


Luego, para generar el código en la opción Project del menú seleccionar la opción Create a Skeleton Program.



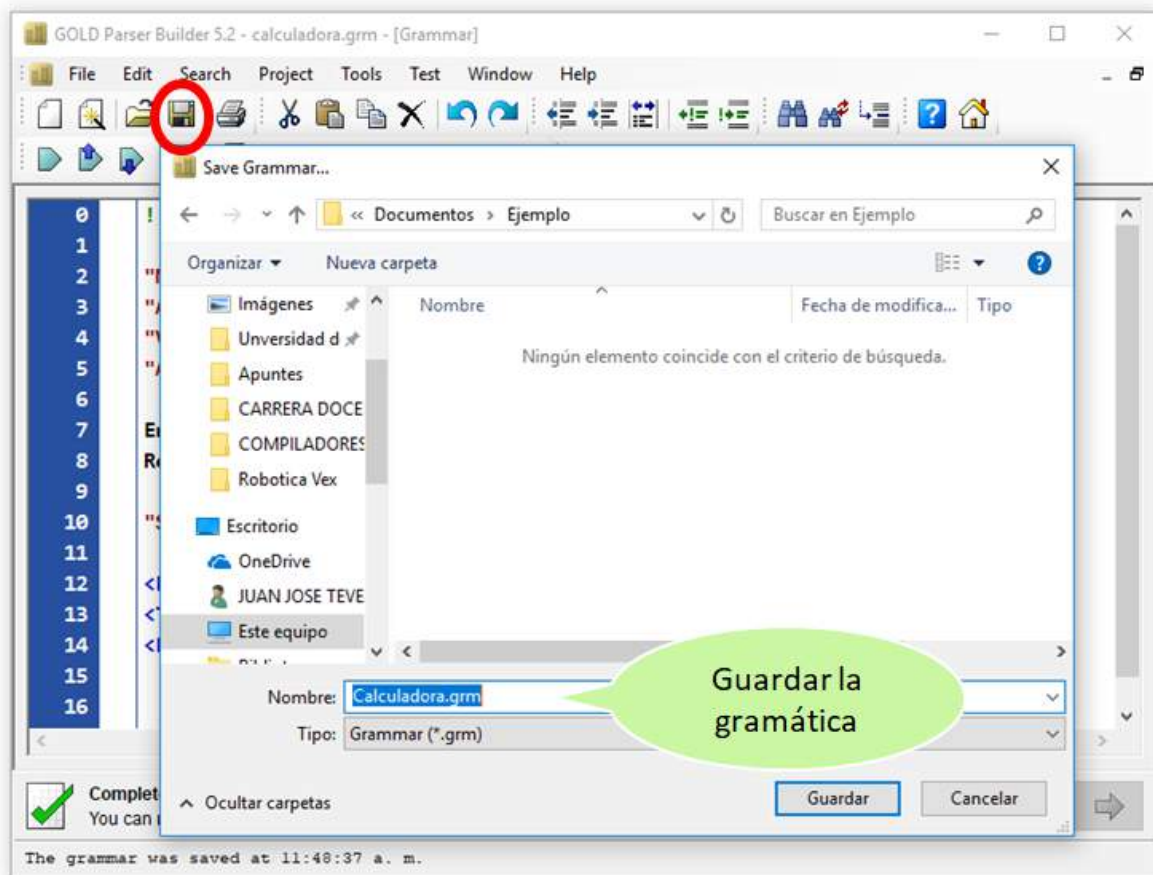
Fuente: Imagen del autor.

En este ejemplo se ha seleccionado generar código en C# con el motor Calitha Engine para aplicaciones .NET seleccionar Event based custom parser.



Fuente: Imagen del autor.

Luego, guardar la gramática con extensión **grm**.



*Fuente: Imagen del autor.*

Gold Parser Builder ha creado los tres archivos.



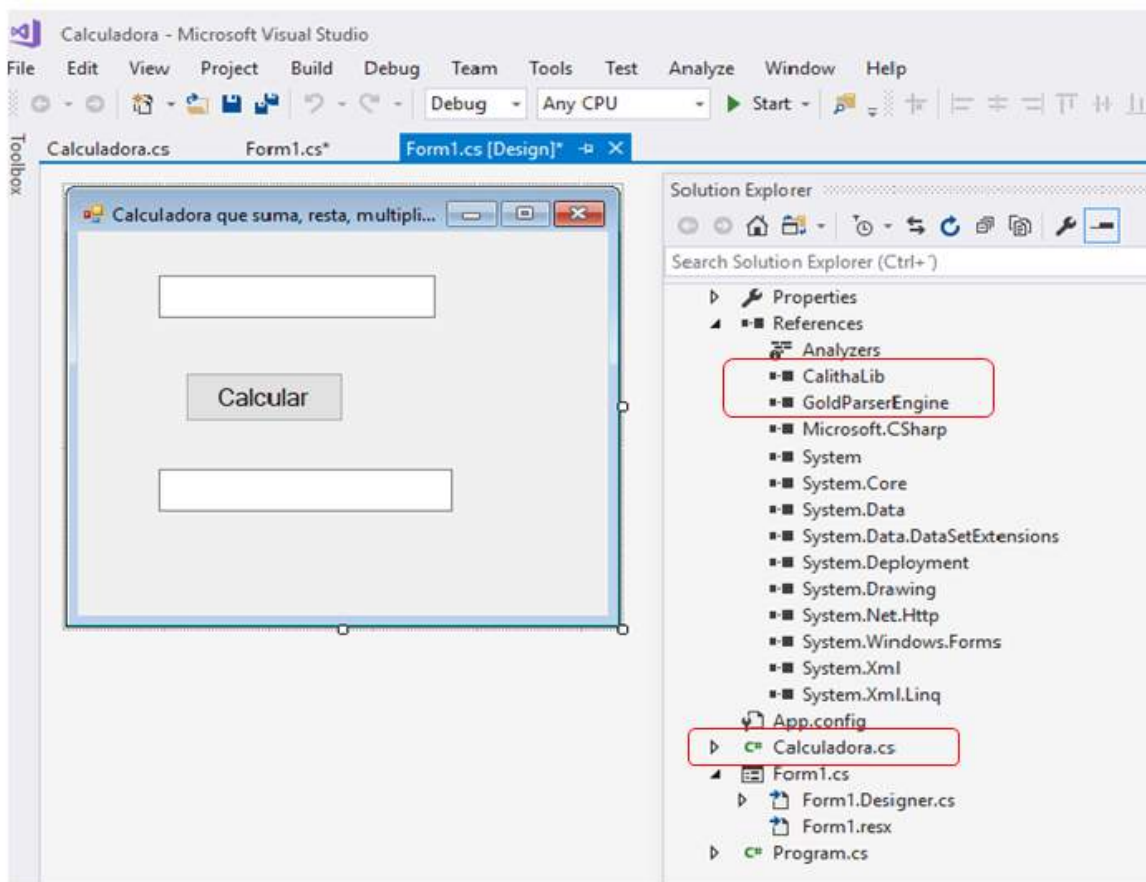
*Fuente: Imagen del autor.*

### 4.3 Construir la aplicación en Visual Studio C#.NET

En Visual Studio crear una proyecto de nombre Calculadora y en el formulario crear los siguientes controles:

- txtEntrada tipo TextBox donde se captura la expresión aritmética.
- btnCalcular tipo Button donde se realizan los cálculos.
- txtResultado tipo TextBox donde se envía el resultados de evaluar la expresión.

En el explorador de soluciones, seleccionar agregar la referencias y seleccionar los archivos dll siguientes CalithaLib y GoldParserEngine, también agregar el elemento existente archivo Calculadora.cs que generó Gold parser, como se muestra en la siguiente imagen.

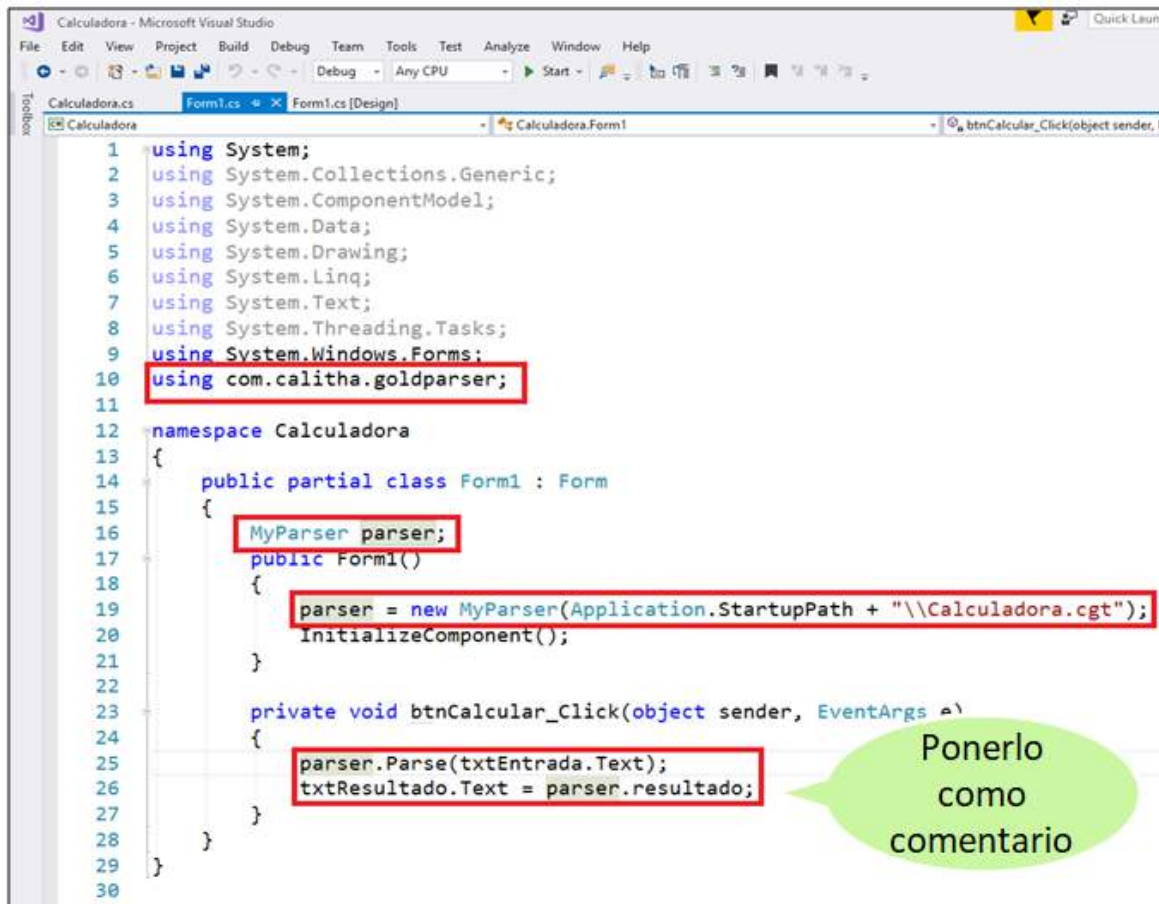


*Fuente: Imagen del autor.*

En Forms.cs escribir las líneas seleccionadas en rojo, la última línea dejarlo inicialmente en comentario para que no envíe error.

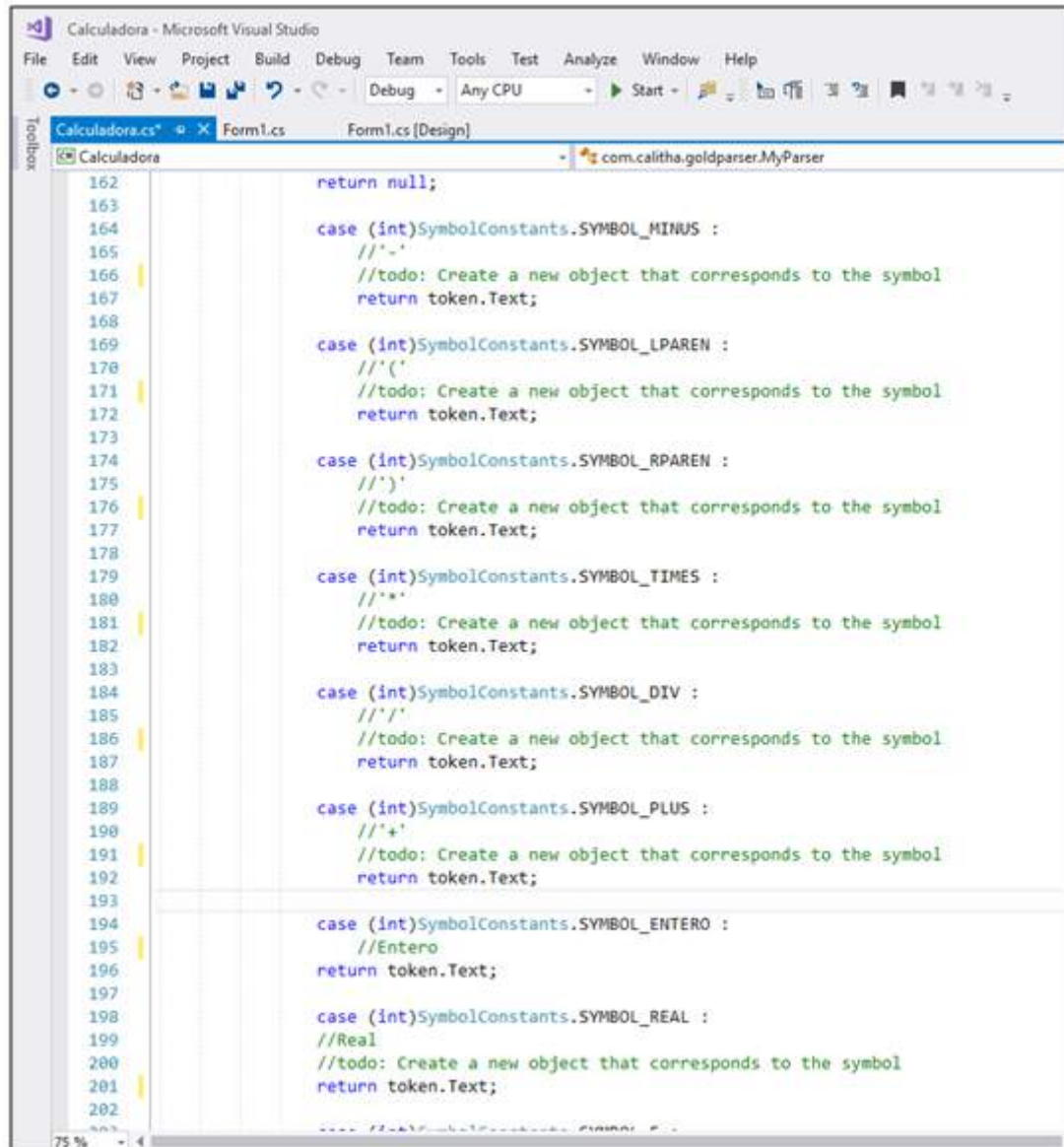
```
// txtResultado.Text = parser.resultado;
```

posteriormente cuando se termine de escribir código en Calculadora.cs se quitará el comentario de línea.



Fuente: Imagen del autor.

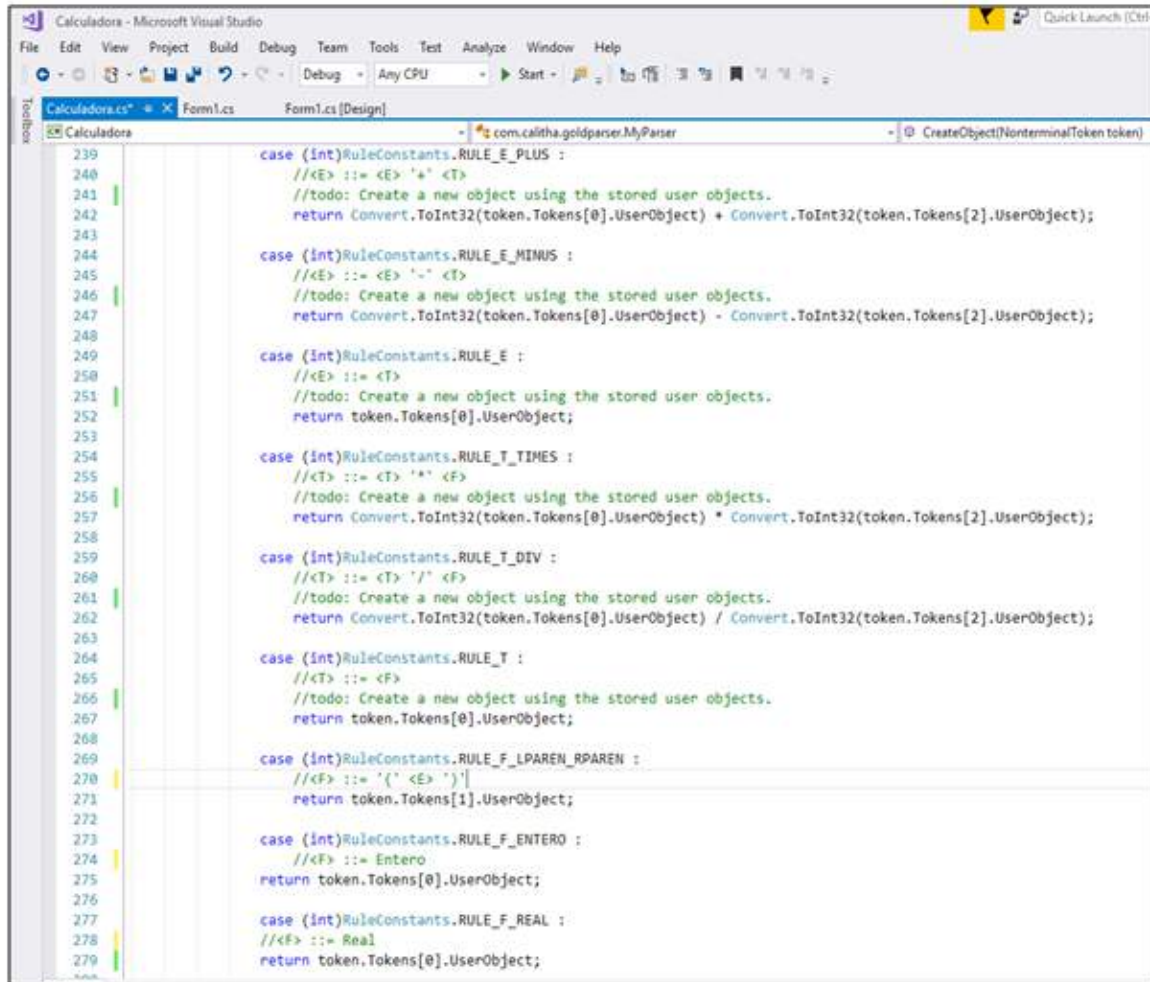
El método `private Object CreateObject(TerminalToken token)` es la interfaz con el análisis léxico o tabla AFD y se debe escribir en cada return de los símbolos terminales `+`, `-`, `*`, `/`, `(`, `)`, `Enter` y `Real` la línea `return token.Text;` como se muestra en la imagen.



*Fuente: Imagen del autor.*



El método public static Object CreateObject(NonterminalToken token) es la interfaz con el análisis sintáctico o tabla LALR y se debe escribir en cada producción cada return como aparece en la imagen siguiente.



```
239 case (int)RuleConstants.RULE_E_PLUS :  
240     //<E> ::= <E> '+' <T>  
241     //todo: Create a new object using the stored user objects.  
242     return Convert.ToInt32(token.Tokens[0].UserObject) + Convert.ToInt32(token.Tokens[2].UserObject);  
243  
244 case (int)RuleConstants.RULE_E_MINUS :  
245     //<E> ::= <E> '-' <T>  
246     //todo: Create a new object using the stored user objects.  
247     return Convert.ToInt32(token.Tokens[0].UserObject) - Convert.ToInt32(token.Tokens[2].UserObject);  
248  
249 case (int)RuleConstants.RULE_E :  
250     //<E> ::= <T>  
251     //todo: Create a new object using the stored user objects.  
252     return token.Tokens[0].UserObject;  
253  
254 case (int)RuleConstants.RULE_T_TIMES :  
255     //<T> ::= <T> '*' <F>  
256     //todo: Create a new object using the stored user objects.  
257     return Convert.ToInt32(token.Tokens[0].UserObject) * Convert.ToInt32(token.Tokens[2].UserObject);  
258  
259 case (int)RuleConstants.RULE_T_DIV :  
260     //<T> ::= <T> '/' <F>  
261     //todo: Create a new object using the stored user objects.  
262     return Convert.ToInt32(token.Tokens[0].UserObject) / Convert.ToInt32(token.Tokens[2].UserObject);  
263  
264 case (int)RuleConstants.RULE_T :  
265     //<T> ::= <F>  
266     //todo: Create a new object using the stored user objects.  
267     return token.Tokens[0].UserObject;  
268  
269 case (int)RuleConstants.RULE_F_LPAREN_RPAREN :  
270     //<F> ::= '(' <E> ')'  
271     return token.Tokens[1].UserObject;  
272  
273 case (int)RuleConstants.RULE_F_ENTERO :  
274     //<F> ::= Entero  
275     return token.Tokens[0].UserObject;  
276  
277 case (int)RuleConstants.RULE_F_REAL :  
278     //<F> ::= Real  
279     return token.Tokens[0].UserObject;
```

Fuente: Imagen del autor.

Por último, se escribe el código que se encuentra subrayado y en recuadro.

```
private void AcceptEvent(LALRParser parser, AcceptEventArgs args)
```

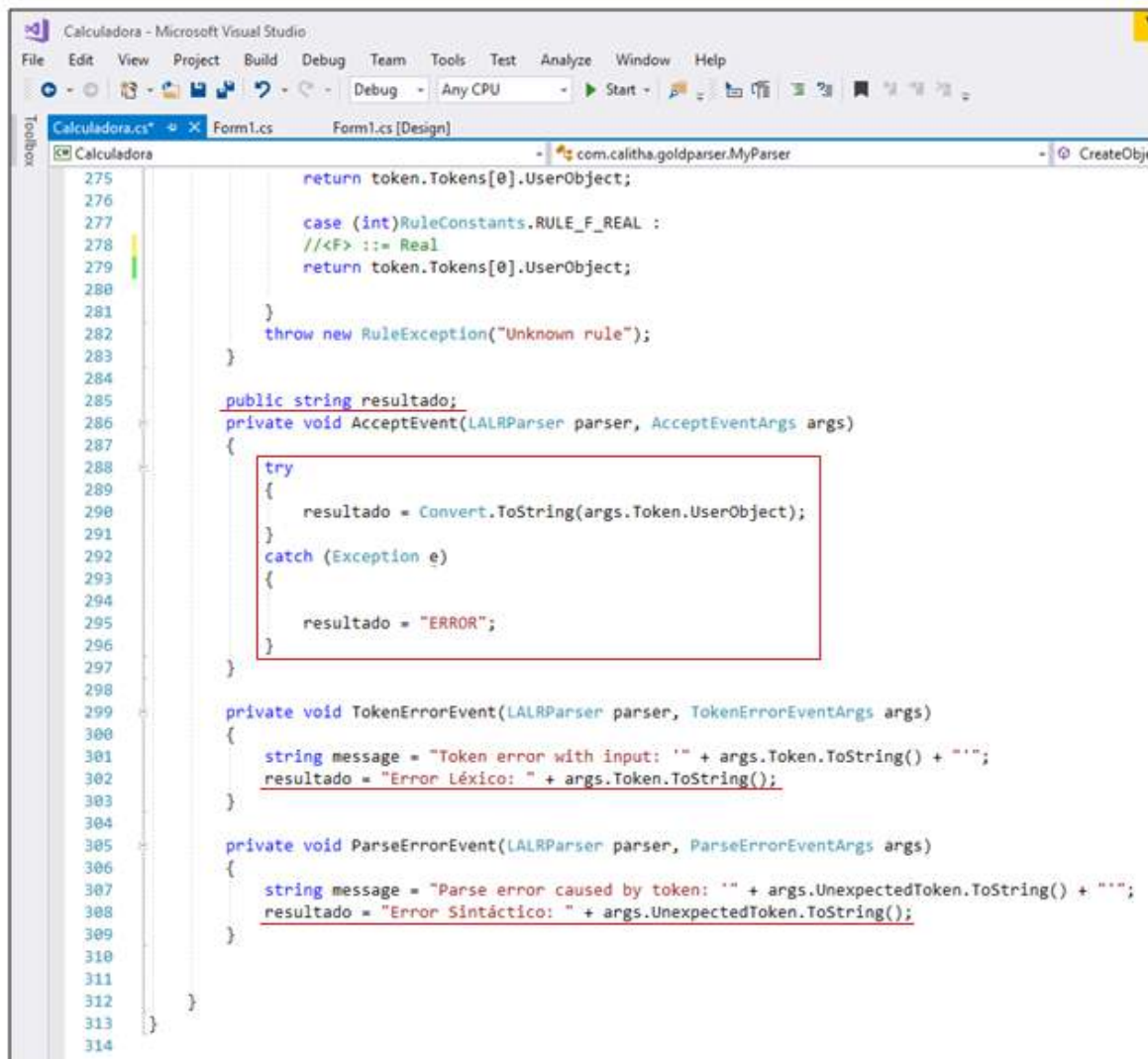
En este método se obtiene el resultado de la evaluación de la expresión.

```
private void TokenErrorEvent(LALRParser parser, TokenErrorEventArgs args)
```

Este método envía el error léxico en caso de un símbolo diferente a los terminales.

```
private void ParseErrorEvent(LALRParser parser, ParseErrorEventArgs args)
```

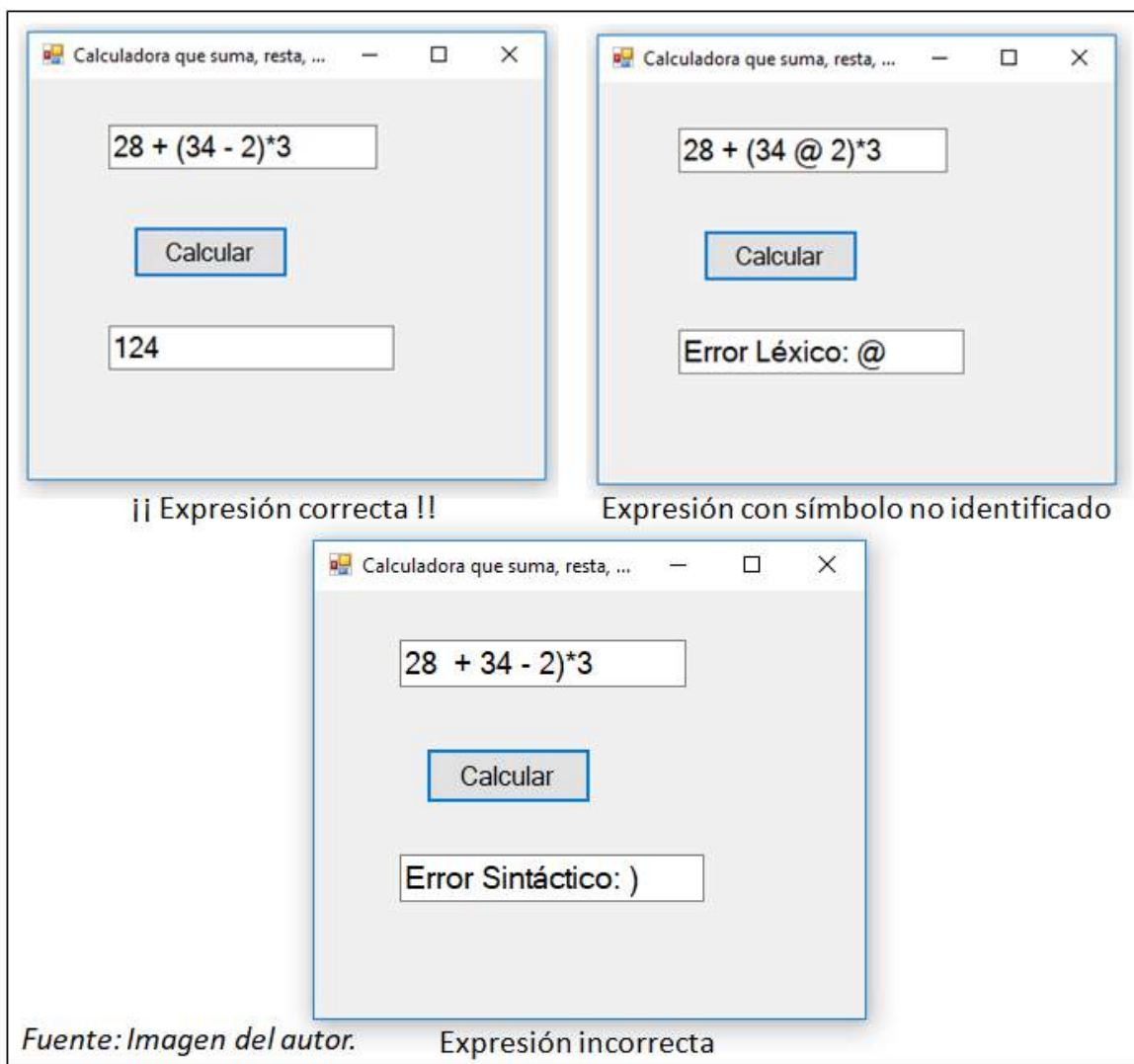
Este método envía el error sintáctico en caso de escribir una expresión errónea.



Fuente: Imagen del autor.

La siguiente imagen muestra los tres resultados que se obtiene de cada método

- `private void AcceptEvent(LALRParser parser, AcceptEventArgs args)` cuando la expresión es correcta.
- `private void TokenErrorEvent(LALRParser parser, TokenErrorEventArgs args)` cuando la expresión tiene un símbolo no identificado.
- `private void ParseErrorEvent(LALRParser parser, ParseErrorEventArgs args)` cuando la expresión está mal escrita.





## Conclusión

Con el presente apunte de la Teoría Matemática de la Computación se ofrece al estudiante los conocimientos necesarios para que aplique los modelos matemáticos de la teoría de los autómatas en el reconocimiento de cadenas y que sean las bases para el desarrollo de analizadores léxicos, así también el uso de las gramáticas y la herramienta Gold parser Builder para el desarrollo de analizadores sintácticos.

## Referencias

Aho, A. V., Lam, M. S., Sethi, R & Ullman, J. D. (2008). *Compiladores: Principios, técnicas y herramientas*. (Segunda Edi.). (A. V. Romerio Elizondo, Trad.) México: Pearson Educación.

Hopcroft J.E., Motwani R., Ullman J. D. (2008). *Introducción a la teoría de autómatas, lenguajes y computación*. (Tercera Ed.) España: Pearson Educación.

Goldparser Builder. (2012). *Gold Pasing System*: <http://goldparser.org>

Navarro, G. (2017). *Teoría de la computación (Lenguajes formales, computabilidad y complejidad)*. Apuntes. Chile:Universidad de Chile.

Pobar, J. (2008). *Cree un compilador de lenguaje para .NET Framework*. MSDN (febrero). <http://msdn.microsoft.com/es-es/magazine/cc136756.aspx>



UNIVERSIDAD AUTÓNOMA DE CHIAPAS  
Facultad de Contaduría y Administración, Campus I



**Programa descriptivo por unidad de competencia**

Programa educativo	Licenciatura en Ingeniería en Desarrollo y Tecnologías de Software	Modalidad		Presencial	
Clave	MA09	H S M		Horas	Créditos
Unidad de competencia	Teoría matemática de la computación.	Teoría	Práctica	semestrales	
Ubicación	Quinto semestre.	3	2	80	8
Prerrequisito	Estructura de datos y Matemáticas discretas.	Unidades CONAIC		58.67	
Perfil docente	Contar con un título profesional o posgrado en áreas relacionadas con informática y computación preferentemente con el grado de doctorado. Demostrar experiencia en docencia en nivel superior mínima de dos años. Dominio de la programación de propósito general y compiladores y deseable que cuente con certificación en lenguajes de programación.	H S M de cómputo		1	
Presentación	Esta unidad de competencia define modelos y máquinas que pueden ser implementadas en el desarrollo de lenguajes de programación. Las unidades de competencia como conocimiento previo son Estructura de datos por las herramientas para el procesamiento de información que proporciona y Matemáticas discretas que aporta los fundamentos matemáticos de la teoría de conjuntos.				
Propósito	Modela matemáticamente las etapas de léxico y sintaxis de un compilador.				
Competencias genéricas					
Formula propuestas y gestiona proyectos con una visión de sustentabilidad para la solución de problemas. Se mantiene actualizado en los conocimientos y habilidades de manera permanente y los utiliza en su práctica profesional y vida personal.					
Competencias disciplinares					
Aplica habilidades de abstracción y expresión matemática para la solución de problemas. Formula modelos matemáticos para la solución de problemas mediante el desarrollo de aplicaciones de software para diversos entornos. Posee conocimientos formales sobre las bases matemáticas de la computación y los aplica en la solución de problemas.					
Competencias profesionales					
Realiza el modelado de problemas con base en la computación teórica. Realiza la codificación de algoritmos en computación teórica. Selecciona técnicas o algoritmos en computación teórica para la solución de problemas.					



UNIVERSIDAD AUTÓNOMA DE CHIAPAS  
Facultad de Contaduría y Administración, Campus I



**Mapa de la unidad de competencia**

Unidad de competencia	Subcompetencia	Resultado de aprendizaje
Teoría matemática de la computación	1. Comprende las bases de la teoría de lenguaje.	1.1. Comprende conceptos sobre lenguajes formales.
	2. Utiliza el lenguaje y las expresiones regulares.	2.1. Aplica los conceptos de lenguaje regular en el desarrollo de ejercicios.
	3. Resuelve problemas de autómatas finitos.	3.1. Aplica los conceptos de autómatas finitos en el desarrollo de ejercicios.
	4. Hace uso de gramáticas.	4.1. Aplica los conceptos de gramáticas en el desarrollo de ejercicios.
	5. Comprende el funcionamiento de las Máquinas de Turing.	5.1. Conoce el concepto de la máquina de Turing.
	6. Comprende la aplicación de la Computabilidad.	6.1. Conoce los conceptos de computabilidad.



UNIVERSIDAD AUTÓNOMA DE CHIAPAS  
Facultad de Contaduría y Administración, Campus I



Cuadro descriptivo por subcompetencia

<b>Subcompetencia</b>	<b>Comprende las bases de la teoría de los lenguajes formales.</b>	<b>Número</b>	<b>1</b>
<b>Propósito de la subcompetencia</b>	Comprende los conceptos básicos relacionados con los lenguajes.	<b>Total de horas</b>	10
<b>Resultado de aprendizaje</b>	1.1. Comprende conceptos sobre lenguajes formales.	<b>Horas asignadas</b>	10
<b>Actividades de evaluación</b>	<b>Evidencias a recopilar</b>	<b>%</b>	<b>Contenido</b>
1. Resuelve un cuestionario con los temas vistos.	1. Cuestionario resuelto.	10%	1. Teoría de conjuntos orientada a lenguaje formal. 2. Conceptos de lenguaje formal. 3. Tipos de lenguaje. 4. Etapas de un compilador.



UNIVERSIDAD AUTÓNOMA DE CHIAPAS  
Facultad de Contaduría y Administración, Campus I



Cuadro descriptivo por subcompetencia

<b>Subcompetencia</b>	<b>Utiliza el lenguaje y las expresiones regulares.</b>	<b>Número</b>	<b>2</b>
<b>Propósito de la subcompetencia</b>	Aplica el concepto de lenguaje regular en el desarrollo conceptual de la etapa de léxico de un lenguaje de programación.	<b>Total de horas</b>	10
<b>Resultado de aprendizaje</b>	2.1. Aplica los conceptos de lenguaje regular en el desarrollo de ejercicios.	<b>Horas asignadas</b>	10
<b>Actividades de evaluación</b>	<b>Evidencias a recopilar</b>	<b>%</b>	<b>Contenido</b>
1. Realiza ejercicios de los conceptos vistos.	1. Reporte de ejercicios.	15%	1. Definición formal de lenguaje regular. 2. Definición de expresión regular. 3. Diseño de expresiones regulares.



UNIVERSIDAD AUTÓNOMA DE CHIAPAS  
Facultad de Contaduría y Administración, Campus I



Cuadro descriptivo por subcompetencia

<b>Subcompetencia</b>	<b>Resuelve problemas de autómatas finitos.</b>	<b>Número</b>	<b>3</b>
<b>Propósito de la subcompetencia</b>	Aplica los conceptos de autómatas finitos en el desarrollo conceptual de la etapa de léxico de un lenguaje de programación.	<b>Total de horas</b>	15
<b>Resultado de aprendizaje</b>	3.1. Aplica los conceptos de autómatas finitos en el desarrollo de ejercicios.	<b>Horas asignadas</b>	15
<b>Actividades de evaluación</b>	<b>Evidencias a recopilar</b>	<b>%</b>	<b>Contenido</b>
1. Realiza ejercicios de los conceptos vistos.	1. Reporte de ejercicios.	20%	1. Definición formal de autómata finito. 2. Tipos de autómatas finitos. 3. Conversión de autómata finito no determinista a autómata finito determinista. 4. Representación de expresión regular en autómata finito. 5. Aplicaciones de los autómatas.



UNIVERSIDAD AUTÓNOMA DE CHIAPAS  
Facultad de Contaduría y Administración, Campus I



**Cuadro descriptivo por subcompetencia**

<b>Subcompetencia</b>	<b>Hace uso de gramáticas.</b>	<b>Número</b>	<b>4</b>
<b>Propósito de la subcompetencia</b>	Aplica los conceptos de gramáticas en el desarrollo conceptual de la etapa de sintaxis de un lenguaje de programación.	<b>Total de horas</b>	<b>25</b>
<b>Resultado de aprendizaje</b>	4.1. Aplica los conceptos de gramáticas en el desarrollo de ejercicios.	<b>Horas asignadas</b>	<b>25</b>
<b>Actividades de evaluación</b>	<b>Evidencias a recopilar</b>	<b>%</b>	<b>Contenido</b>
1. Realiza ejercicios de los conceptos vistos.	1. Reporte de ejercicios.	25%	1. Definición formal de gramáticas. 2. Jerarquía de Chomsky. 3. Diseño de gramáticas. 4. Aplicaciones de las gramáticas.



UNIVERSIDAD AUTÓNOMA DE CHIAPAS  
Facultad de Contaduría y Administración, Campus I



Actitudes y valores	Responsabilidad. Honestidad. Ética.	
	Recursos, materiales y equipo didáctico	
Recursos didácticos		Equipo de apoyo didáctico
Apuntes. Diapositivas, Ejercicios. Guías de práctica.		Proyector de video.
Fuentes de información		
<b>Bibliografía básica:</b> Hopcroft, J. E. (2006). <i>Introduction to Automata Theory, Languages and Computation</i> , (3rd. Ed.). USA: Adison Wesley. Jiménez Murillo, Jose A. (2008). <i>Matemáticas para computación</i> . México: Alfaomega. Kelley, D. (1995). <i>Teoría de Autómatas y Lenguajes Formales</i> . Madrid: Prentice Hall. Guerra Crespo, H. (2005). <i>Compiladores, el comienzo...</i> México: Tecnología Didáctica.		
<b>Bibliografía complementaria:</b> Aho, Lam, Sethi, Ullman (2008). <i>Compiladores, principios, técnicas y herramientas</i> , (2a. ed.). México: Pearson Adison Wesley. Louden K.C. (2004). <i>Construcción de compiladores: principios y práctica</i> . México: Thomson.		
<b>Recursos digitales:</b> Ninguno.		