

# TALFi 2.0



Rocío Barrigüete

Mario Huete

Luis San Juan

Director: Alberto De La Encina

*Facultad de Informática.*

*Universidad Complutense de Madrid.*

*Curso 2009-2010*

*Junio 2010*



# Índice general

<b>1. Resumen</b>	<b>5</b>
1.1. TALFi (Versión en castellano) . . . . .	5
1.2. TALFi (English version) . . . . .	5
<b>2. Objetivos</b>	<b>7</b>
<b>3. Información</b>	<b>9</b>
3.1. Antecedentes . . . . .	9
3.2. Problemas encontrados y resueltos en TALFi 1.0 . . . . .	10
3.2.1. Minimización . . . . .	10
3.2.2. Palabras reconocidas por un autómata de pila . . . . .	14
3.3. Introducción a la ampliación de la aplicación . . . . .	16
3.4. Conceptos teóricos de la ampliación . . . . .	17
3.4.1. Lenguaje Independiente de Contexto . . . . .	17
3.4.2. Autómata de Pila . . . . .	17
3.4.3. Gramática Libre de Contexto . . . . .	19
3.4.4. Máquina de Turing . . . . .	22
3.5. Lenguaje generado por un AP . . . . .	28
3.5.1. Algoritmo de paso de AP $\rightarrow$ Gramática . . . . .	28
3.5.2. Algoritmo de simplificación Gramática tipo 2 a Gramática de Greibach . . . . .	31
3.5.3. Algoritmo de simplificación de gramática de tipo 2 a gramática de Chomsky . . . . .	37
3.5.4. Generación aleatoria de palabras . . . . .	41
3.6. Máquinas de Turing . . . . .	42
3.6.1. Nociones preliminares . . . . .	42
3.6.2. Algoritmo de reconocimiento . . . . .	43
3.6.3. Ampliaciones y problemas . . . . .	44
3.7. Ejercicios . . . . .	45

3.7.1. Cambios en la interfaz gráfica e implementación interna para ejercicios de Autómatas de Pila y Máquinas de Turing . . . . .	45
<b>4. Anexos</b>	<b>47</b>
4.1. $\text{\LaTeX}$ . . . . .	47
4.1.1. Introducción a $\text{\LaTeX}$ . . . . .	47
4.1.2. Descripción: . . . . .	48
4.1.3. Uso . . . . .	49
4.1.4. Uso de $\text{\LaTeX}$ en la herramienta TALFi . . . . .	50
4.1.5. LatexCodeConverter . . . . .	51
4.1.6. TraductorHTML . . . . .	56
4.1.7. Entornos $\text{\LaTeX}$ utilizados . . . . .	58
4.2. Ampliación del Manual de Usuario . . . . .	62
4.3. Cambios en la implementación . . . . .	63
4.4. Cambios en la interfaz gráfica . . . . .	66

# Capítulo 1

## Resumen

### 1.1. TALFi (Versión en castellano)

TALFi nació en el curso académico 2008-2009 como una aplicación de apoyo y consulta sobre teoría de autómatas y lenguajes formales. Su primera versión contenía funcionalidad acerca de autómatas finitos (deterministas, no deterministas y con transiciones vacías), y expresiones regulares, así como su equivalencia, simplificación y demás aspectos relacionados. Para TALFi 2.0 se han creado muchas más características adicionales, así como la mejora de ciertos aspectos de la primera versión, que no resultaban del todo intuitivos para el usuario y claros en la comprensión. Con esta nueva ampliación, TALFi se convierte en una herramienta de gran expresividad dentro del entorno de los autómatas y la generación de lenguajes, a la altura de otras tecnologías ya conocidas como JFLAP, etc.

### 1.2. TALFi (English version)

TALFi was created in the academic course of 2008-2009 as a student support program for automaton theories and formal languages. Its first version contained finited-state automaton (deterministic, non-deterministic and generalized nondeterministic finite automaton) and regular expressions, so as their equivalence, simplified and more related aspects. For TALFi 2.0 has been created a lot more of additional options, as the improvement of some parts of the first version, which was not completely intuitive for the user and clear in its comprehension. With this new ampliation, TALFi turns into a great tool into the automaton world and the language generations, in the same leves as JFLAP, etc.



# Capítulo 2

## Objetivos

En esta segunda versión de TALFi, como se ha mencionado antes, se ha buscado la ampliación de la herramienta inicial añadiéndole funcionalidad extra para el tratamiento de gramáticas independientes del contexto (GIC), autómatas de pila (AP) y máquinas de Turing (MT), parte muy importante de la teoría de autómatas y lenguajes formales, así como ciertos algoritmos necesarios para su simplificación, corrección y tratamiento. Además TALFi incorpora la posibilidad de obtener ciertos aspectos (visualización de minimizaciones, autómatas, ...) en formatos web como html o en formatos de texto como el más usado a nivel científico,  $\text{\LaTeX}$ .

También se nos pidió que mejorásemos la minimización de autómatas finitos que nuestros compañeros habían implementado en TALFi 1.0, describiendo de forma más detallada las tablas que se generan en este algoritmo para que quedasen más claros los pasos que se iban siguiendo.





# Capítulo 3

## Información

### 3.1. Antecedentes

Existen muchas y diversas aplicaciones que operan con autómatas de manera similar a como lo hace TALFi, pero la mayoría de estas están enfocadas a un tipo de problema en concreto. Algunas tratan autómatas finitos, otras se centran en operaciones de traducción de lenguajes, algunas incorporan simulación de máquinas de Turing, pero ninguna cuenta con la expresividad y riqueza de TALFi 2.0. Para más detalle de estas aplicaciones conviene visitar los siguientes enlaces:

<http://www.versiontracker.com/dyn/moreinfo/win/35508>  
<http://www.cs.duke.edu/csed/jflap/>  
<http://www.ucse.edu.ar/fma/sepa/>  
<http://www.brics.dk/automaton/>

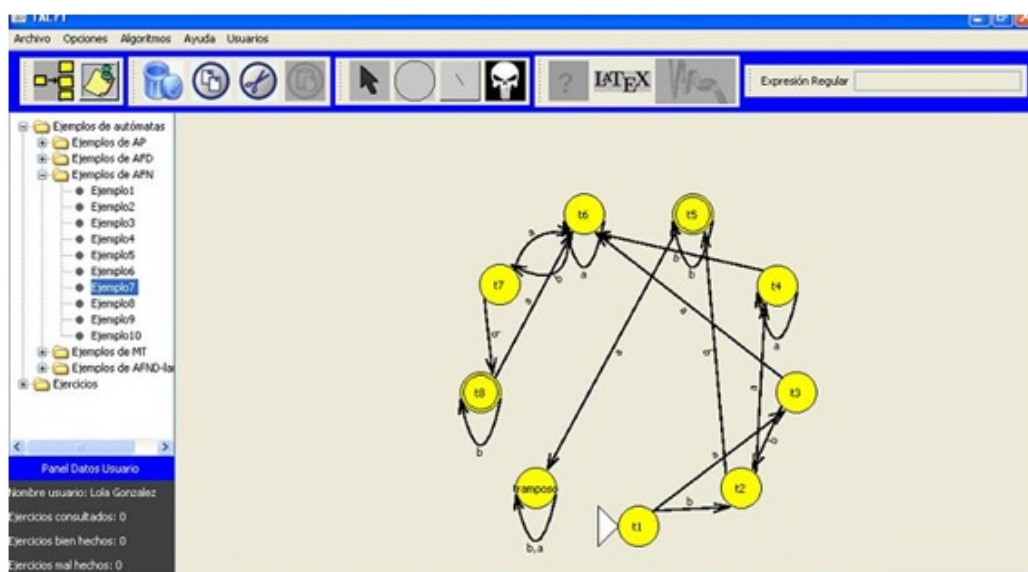
Todas estas versiones pueden ser descargadas gratuitamente. Obviamente, como antecedente más directo contamos con la primera versión de TALFi. A diferencia de los programas anteriormente mencionados, TALFi dispone de la posibilidad de ver paso a paso como se simplifica una gramática, saber si una palabra puede ser reconocida por un autómata de pila, generación de palabras que son reconocidas por un lenguaje y multitud de posibilidades que TALFi contempla a diferencia de sus antecedentes.

## 3.2. Problemas encontrados y resueltos en TAL-Fi 1.0

En un primer momento, tuvimos que dedicar mucho tiempo a entender el funcionamiento de la aplicación, y por tanto el código creado por nuestros compañeros el año pasado. A continuación describimos en que partes encontramos dificultades.

### 3.2.1. Minimización

Mientras que nos dedicábamos a ir un paso más allá con la minimización, descubrimos varios fallos en este algoritmo, teniéndolo que rehacer en parte, para poder cuadrarlo con los cambios que se nos habían pedido. Finalmente, conseguimos llegar a los objetivos en un primer momento propuestos. Véase un ejemplo de la ejecución actual de la minimización de dos autómatas finitos sobre uno de los ejemplos que el año pasado se usaba para mostrar el funcionamiento de este algoritmo: Aquí tenemos el autómata sobre el que vamos a aplicar la minimización, concretamente el ejemplo 7 de los autómatas finitos no deterministas:



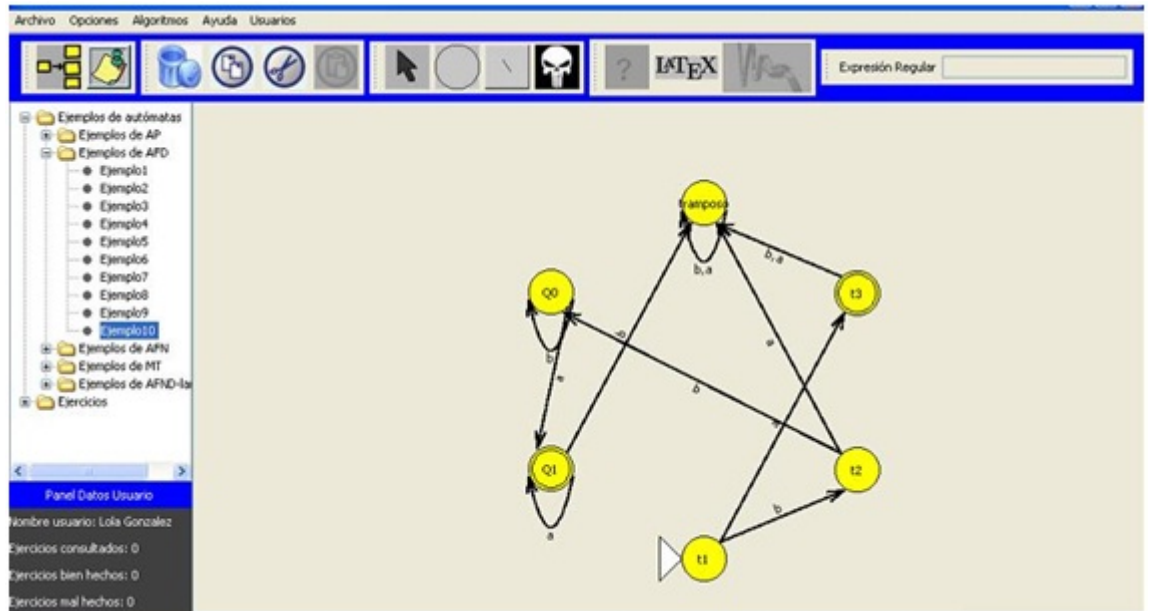
Y la tabla que obtenemos en el HTML generado cuando queremos ver todos los pasos que hemos seguido en la minimización, es la siguiente:

t2	2 t2-t5							
t3	5 t3-t6	2 t5-t2						
t4	3 t3-t4	2 t5-t6	2 t2-t6					
t5	1 Final-NoFinal	1 Final-NoFinal	1 Final-NoFinal	1 Final-NoFinal				
t6	5 t3-t6	2 t5-t7	4 t2-t7	3 t4-t6	1 Final-NoFinal			
t7	2 t2-t8	4 t4-t6	2 t2-t8	2 t6-t8	1 Final-NoFinal	2 t7-t8		
t8	1 Final-NoFinal	1 Final-NoFinal	1 Final-NoFinal	1 Final-NoFinal	4 tramp-t6	1 Final-NoFinal	1 Final-NoFinal	
tramposo	3 t3-tramp	2 t5-tramp	2 t2-tramp	4 t4-tramp	1 Final-NoFinal	3 t6-tramp	2 t8-tramp	1 Final-NoFinal
	t1	t2	t3	t4	t5	t6	t7	t8

En cada celda podemos ver:

1. Si el estado de la columna es final y el de la fila no final, o viceversa, los dos estados no podrán colapsarse, y lo indicamos con Final-No final.
2. A partir de las casillas marcadas con estados que nunca podrán colapsarse, procedemos a comprobar que ocurre con el resto de casillas. El número que aparece nos indica en que "vuelta" hemos descubierto que los estados correspondientes a la fila y a la columna no se pueden colapsar. Y para terminar la explicación de por qué se marca esa casilla, debajo o al lado indicamos que par de estados son los causantes.

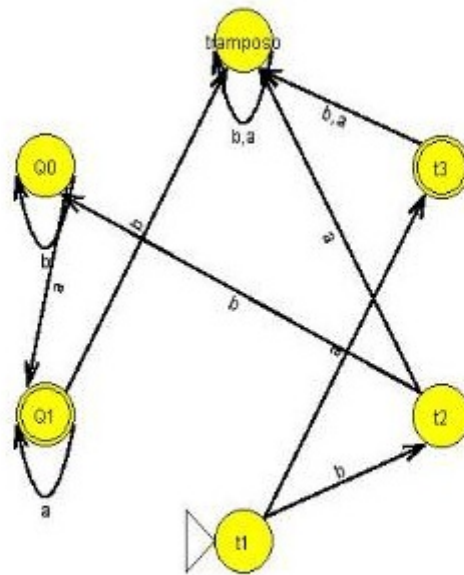
Si los estados pudieran colapsarse, la casilla no tendrá nada escrito dentro de ella, como ocurre en el ejemplo 10 de autómatas finitos deterministas:



Cuya tabla de equivalencia es:

t2	2 t3-tramp							
t3	1 Final- NoFinal	1 Final- NoFinal						
t4	3 t3-t5	2 tramp-t5	1 Final- NoFinal					
t5	1 Final- NoFinal	1 Final- NoFinal	2 tramp-t7	1 Final- NoFinal				
t6	3 t3-t5	2 tramp-t5	1 Final- NoFinal		1 Final- NoFinal			
t7	1 Final- NoFinal	1 Final- NoFinal	2 tramp-t7	1 Final- NoFinal		1 Final- NoFinal		
tramoso	2 t3-tramp	3 t4-tramp	1 Final- NoFinal	2 t5-tramp	1 Final- NoFinal	2 t5-tramp	1 Final- NoFinal	
	t1	t2	t3	t4	t5	t6	t7	

Y por último, en el HTML dibujamos el autómata resultado de la minimización, que será el mismo que se visualice en la aplicación.



### 3.2.2. Palabras reconocidas por un autómata de pila

Al tener la aplicación un enfoque didáctico, la primera idea que nos vino a la cabeza para saber si una palabra es reconocida por un autómata de pila, fue seguir la misma metodología que vimos en clase de TALF. No traducíamos el autómata de pila en una gramática, sino que íbamos simulando el comportamiento que tenía el autómata según iba leyendo los símbolos de la cadena. Conseguimos que este algoritmo funcionase correctamente, siempre que se cumpliera una condición: que el autómata no tuviera ciclos donde solo se lea la cadena vacía sin modificar la pila o añadiendo más símbolos a la misma.

Básicamente la idea de funcionamiento de este algoritmo es muy sencilla: Es un algoritmo recursivo, que posee backtracking, pues en muchos casos tenemos más de un camino posible a seguir al procesar una cadena. Si en algún momento, al terminar de consumir los símbolos de la cadena, llegaba a un estado de aceptación, o se vaciaba la pila, se devolvía un booleano que indicaba que la palabra era aceptada y la se terminaba la ejecución del algoritmo. En caso contrario, la palabra no se aceptaba y se devolvía falso. El procedimiento que contenía este algoritmo recibe como parámetros:

1. Lista de estados posibles a los que podemos ir. Tiene prioridad si existe una transición que no consume símbolos. La idea es la siguiente: cuando vamos a procesar el primer símbolo de la cadena, con transiciones vacías es trivial, porque no hemos consumido aún ningún símbolo, pero si no las hubiera, recopilaríamos todos los posibles estados destino, y asumiríamos que ya hemos consumido un símbolo de la cadena a reconocer.
2. Pila actual: El estado de la pila hasta el momento. En cada estado al que nos movemos va cambiando, así que solo la modificamos si volvemos a llamar al procedimiento. Si en algún momento se vacía, y si estamos comprobando si la palabra es reconocida por estado, abortamos el procesamiento.
3. Número de elementos que tiene la pila: Incluido por si en algún momento sirve para detectar los ciclos de los que hemos hablado anteriormente.
4. Cima de la pila en el momento actual: Por comodidad, y por si se quiere comprobar el caso extremo de que si se ha desapilado el símbolo de fondo de pila, y solo queda un símbolo en la pila, pero no coinciden, se pare la simulación por no poder averiguar, si se ha terminado de vaciar la pila o no. Aquí sería fácil averiguarlo, pero en la teoría de autómatas se restringe de esta manera.

5. Palabra a procesar: Se podría haber pasado sólo como parámetro el símbolo actual que se procesa, pero para posibles explicaciones se incluyó la palabra completa.
6. Índice del símbolo procesado: Se incluye para poder ubicar por cuál símbolo de la cadena va la ejecución.
7. Estado: Estado actual en el que estamos, que nos sirve para poder calcular los posibles movimientos que podemos hacer.

Este procedimiento se apoya en otros dos, que se intuyen según se ha explicado cómo funciona: el primero, que calcula como quedará la pila para la siguiente llamada al procedimiento, y el segundo, que devuelve la lista de posibles estados a los que podemos seguir procesando con un símbolo o no, si es que la transición es vacía, el estado actual, y la cima de pila actual.

Quizá en un futuro sean pautas que alguien pueda seguir para explicar el funcionamiento de los autómatas de pila. Nosotros íbamos a utilizarlo para probar que una palabra pertenecía al lenguaje de un autómata de pila, pero el coste se disparaba si lo comparamos con el algoritmo llamado CYK, cuyo coste es cúbico sobre la longitud de la cadena.

### 3.3. Introducción a la ampliación de la aplicación

Repaso a la funcionalidad de TALFi 1.0:

TALFi 1.0 es capaz de crear autómatas finitos(deterministas, no deterministas y con transiciones vacías)y expresiones regulares. Como algoritmos importantes cuenta con la transformación de un autómata con transiciones vacías a uno no determinista, la transformación de un autómata no determinista a uno determinista, la transformación de autómata finito determinista a expresión regular y la minimización de autómatas finitos no deterministas. La interfaz de usuario de TALFi 1.0 cuenta con un árbol desplegable donde podemos encontrar ejemplos y ejercicios que la primera versión de TALFi es capaz de llevar a cabo. También dispone de los usuales controles para modificar, copiar, pegar, borrar cualquier aspecto de la figura que creemos en la zona central de la interfaz.



## 3.4. Conceptos teóricos de la ampliación

### 3.4.1. Lenguaje Independiente de Contexto

Un lenguaje independiente de contexto es aquel generado por una gramática independiente de contexto. Estos conceptos pertenecen a un área de la Ciencia de la Computación llamada Computación Teórica. No hay algoritmo que nos diga el lenguaje de la gramática, por eso tenemos que ir viendo los símbolos y cadenas que produce.

### 3.4.2. Autómata de Pila

Un autómata con pila o autómata de pila o autómata a pila o autómata apilador es un modelo matemático de un sistema que recibe una cadena constituida por símbolos de un alfabeto y determina si esa cadena pertenece al lenguaje que el autómata reconoce. El lenguaje que reconoce un autómata a pila pertenece al grupo de los lenguajes de contexto libre en la clasificación de la Jerarquía de Chomsky.

- Definición formal:

Formalmente, un autómata con pila puede ser descrito como una séptupla

$$M = (S, \Sigma, \Gamma, \delta, s, Z, F)$$

- $\Sigma$  y  $\Gamma$  son alfabetos de entrada, de la cadena y de la pila respectivamente;
- $S$  un conjunto de estados;
- $\delta: S \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \rho(S \times \Gamma^*)$
- $s \in S$  es el estado inicial;
- $Z \in \Gamma$  es el símbolo inicial de la pila;
- $F \subseteq S$  es un conjunto de estados de aceptación o finales.

La interpretación de  $\delta(s,a,Z) = \{(s_1,\gamma_1), (s_2,\gamma_2), \dots, (s_n,\gamma_n)\}$ , con  $s, p_i \in Q$ ,  $a \in (\Sigma \cup \{\epsilon\})$ ,  $\gamma_i \in \Gamma$  es la siguiente:

Cuando el estado del autómata es  $s$ , el símbolo que la cabeza lectora está inspeccionando en ese momento es  $a$ , y en la cima de la pila nos encontramos el símbolo  $Z$ , se realizan las siguientes acciones:

- Si  $a \in \Sigma$ , es decir no es la palabra vacía, se avanza una posición la cabeza lectora para inspeccionar el siguiente símbolo.
  - Se elimina el símbolo  $Z$  de la pila del autómata.
  - Se selecciona un par  $(p_i, \gamma_i)$  de entre los existentes en la definición de  $\delta(s,A,Z)$ , la función de transición del autómata.
  - Se apila la cadena  $\gamma_i = A_1 A_2 \dots A_k$  en la pila del autómata, quedando el símbolo  $A_1$  en la cima de la pila.
  - Se cambia el control del autómata al estado  $p_i$ .
- Ejemplo Sea el siguiente lenguaje libre del contexto  $L = \{a^k b^k \mid k \geq 0\}$ ; formado por las cadenas  $L = \{\epsilon, ab, aabb, aaabbb, aaaabbbb, \dots\}$

Dicho lenguaje puede ser reconocido por el siguiente autómata con pila:

$M = (\{q_0, q_1, q_2, q_3\}, \{a,b\}, \{A, \underline{A}\}, \delta, q_0, \{q_0, q_3\})$ , donde las transiciones son:

$$\begin{aligned} \delta(q_0, a, \epsilon) &= \{(q_1, \underline{A})\} \\ \delta(q_1, a, \epsilon) &= \{(q_1, \underline{A})\} \\ \delta(q_1, b, \underline{A}) &= \{(q_2, \epsilon)\} \\ \delta(q_1, b, \underline{A}) &= \{(q_3, \epsilon)\} \\ \delta(q_2, b, \underline{A}) &= \{(q_2, \epsilon)\} \\ \delta(q_2, b, \underline{A}) &= \{(q_3, \epsilon)\} \\ \delta(q, \theta, \rho) &= \text{para cualquier } (q, \theta, \rho) \end{aligned}$$

El significado de las transiciones puede ser explicado analizando la primera transición:

$$\delta(q_0, a, \epsilon) = \{(q_1, \underline{A})\}$$

donde  $q_0$  es el estado actual,  $a$  es el símbolo en la entrada y  $\epsilon$  se extrae de la cima de la pila. Entonces, el estado del autómata cambia a  $q_1$  y el símbolo  $\underline{A}$  se coloca en la cima de la pila.

La idea del funcionamiento del autómata es que al ir leyendo los diferentes símbolos  $a$ , estos pasan a la pila en forma de símbolos  $A$ . Al aparecer el primer símbolo  $b$  en la entrada, se comienza el proceso de desapilado, de forma que coincida el número de símbolos  $b$  leídos con el número de símbolos  $A$  que aparecen en la pila.

■ Autómata de Pila Determinista:

Nótese que, a diferencia de un autómata finito o una máquina de Turing, la definición básica de un autómata con pila es de naturaleza no determinista, pues la clase de los autómatas con pila deterministas, a diferencia de lo que ocurría con aquellos modelos, tiene una potencia descriptiva estrictamente menor. Para calificar a un autómata con pila como determinista deben darse dos circunstancias; en primer lugar, por supuesto, que en la definición de cada componente de la función de transición existan un único elemento lo que da la naturaleza determinista. Pero eso no es suficiente, pues además puede darse la circunstancia de que el autómata esté en el estado  $s$  y en la pila aparezca el símbolo  $sZ$ , entonces, si existe una definición de transición posible para algún símbolo cualquiera  $a$  del alfabeto de entrada, pero, además existe otra alternativa para la palabra vacía  $\epsilon$ , también esto es una forma de no determinismo, pues podemos optar entre leer un símbolo o no hacerlo. Por eso, en autómata determinista no debe existir transición posible con lectura de símbolo si puede hacerse sin ella, ni al contrario.

Para cada  $q \in Q$ ,  $Z \in \Gamma$ , se dé que:  $|\delta(q, \epsilon, Z)| + |\delta(q, a, Z)| \leq 1$ , para cada  $a \in \Sigma$ .

### 3.4.3. Gramática Libre de Contexto

En lingüística e informática, una gramática libre de contexto (o de contexto libre) es una gramática formal en la que cada regla de producción es de la forma:

$$V \rightarrow w$$

Donde  $V$  es un símbolo no terminal y  $w$  es una cadena de terminales y/o no terminales. El término libre de contexto se refiere al hecho de que el no terminal  $V$  puede siempre ser sustituido por  $w$  sin tener en cuenta el contexto en el que ocurra. Un lenguaje formal es libre de contexto si hay una gramática libre de contexto que lo genera.

Las gramáticas libres de contexto permiten describir la mayoría de los lenguajes de programación, de hecho, la sintaxis de la mayoría de lenguajes de programación está definida mediante gramáticas libres de contexto. Por otro lado, estas gramáticas son suficientemente simples como para permitir el diseño de eficientes algoritmos de análisis sintáctico que, para una cadena de caracteres dada determinen como puede ser generada desde la gramática. Los analizadores LL y LR tratan restringidos subconjuntos de gramáticas libres de contexto.

La notación más frecuentemente utilizada para expresar gramáticas libres de contexto es la forma Backus-Naur.

■ Definición Formal:

Así como cualquier gramática formal, una gramática libre de contexto puede ser definida mediante la 4-tupla:

$G = (V_t, V_n, P, S)$  donde

- $V_t$  es un conjunto finito de terminales
- $V_n$  es un conjunto finito de no terminales
- $P$  es un conjunto finito de producciones
- $S \in V_n$  el denominado Símbolo Inicial
- los elementos de  $P$  son de la forma:  

$$V_n \rightarrow (V_t \cup V_n)^*$$

■ Ejemplos:

1. Una simple gramática libre de contexto es

$$S \rightarrow aSb \mid \epsilon$$

donde  $\mid$  es un o lógico y es usado para separar múltiples opciones para el mismo no terminal,  $\epsilon$  indica una cadena vacía. Esta gramática genera el lenguaje no regular  $\{a^n b^n : n \geq 0\}$ .

2. Aquí hay una gramática libre de contexto para expresiones enteras algebraicas sintácticamente correctas sobre las variables  $x$ ,  $y$  y  $z$ :

$$S \rightarrow x \mid y \mid z \mid S + S \mid S - S \mid S * S \mid S / S \mid (S)$$

Generaría, por ejemplo, la cadena  $(x + y) * x - z * y / (x + x)$

3. Una gramática libre de contexto para un lenguaje consistente en todas las cadenas que se pueden formar con las letras a y b, habiendo un número diferente de una que de otra, sería:

$$\begin{aligned} S &\rightarrow U \mid V \\ U &\rightarrow TaU \mid TaT \\ V &\rightarrow TbV \mid TbT \\ T &\rightarrow aTbT \mid bTaT \mid \epsilon \end{aligned}$$

T genera todas las cadenas con la misma cantidad de letras a que b, U genera todas las cadenas con más letras a, y V todas las cadenas con más letras b.

4. Otro ejemplo para un lenguaje es  $\{a^n b^m c^{m+n} : n \geq 0, m \geq 0\}$ . No es un lenguaje regular, pero puede ser generado por la siguiente gramática libre de contexto.

$$S \rightarrow aSc \mid B \quad B \rightarrow bBc \mid E$$

■ Formas normales:

- Forma Normal de Greibach:

Una gramática independiente del contexto (GIC) está en Forma normal de Greibach (FNG) si todas y cada una de sus reglas de producción tienen un consecuente que empieza por un carácter del alfabeto, también llamado símbolo terminal. Formalmente, cualquiera de las reglas tendrá la estructura:

$$A \rightarrow aw$$

Donde “A” es el antecedente de la regla, que en el caso de las GIC debe ser necesariamente un solo símbolo auxiliar. Por su parte, “a” es el mencionado comienzo del consecuente y, por tanto, un símbolo terminal. Finalmente, “w” representa una concatenación genérica de elementos gramaticales, esto es, una sucesión exclusivamente de auxiliares, inclusive, pudiera ser la palabra vacía; en este caso particular, se tendría una regla llamada “terminal”:

$$A \rightarrow a$$

Existe un teorema que prueba que cualquier GIC, cuyo lenguaje no contiene a la palabra vacía, si no lo está ya, se puede transformar en otra equivalente que sí esté en FNG. Para su demostración, normalmente, se procede por construcción, es decir, se plantea directamente un algoritmo capaz de obtener la FNG a partir de una GIC dada.

- Forma Normal de Chomsky:

Una gramática formal está en Forma normal de Chomsky si todas sus reglas de producción son de alguna de las siguientes formas:

$$A \rightarrow BC \text{ ó } A \rightarrow a$$

donde A, B y C son símbolos no terminales (o variables) y a es un símbolo terminal. Todo lenguaje independiente del contexto que no posee a la cadena vacía, es expresable por medio de una gramática en forma normal de Chomsky (GFNCH) y recíprocamente. Además, dada una gramática independiente del contexto, es posible algorítmicamente producir una GFNCH equivalente, es decir, que genera el mismo lenguaje.

#### 3.4.4. Máquina de Turing

- Descripción:

La máquina de Turing es un modelo computacional introducido por Alan Turing en el trabajo “On computable numbers, with an application to the Entscheidungsproblem”, publicado por la Sociedad Matemática de Londres en 1936, en el cual se estudiaba la cuestión planteada por David Hilbert sobre si las matemáticas son decidibles, es decir, si hay un método definido que pueda aplicarse a cualquier sentencia matemática y que nos diga si esa sentencia es cierta o no. Turing ideó un modelo formal de computador, la máquina de Turing, y demostró que existían problemas que una máquina no podía resolver. La máquina de Turing es un modelo matemático abstracto que formaliza el concepto de algoritmo.

La máquina de Turing consta de un cabezal lector/escritor y una cinta infinita en la que el cabezal lee el contenido, borra el contenido anterior y escribe un nuevo valor. Las operaciones que se pueden realizar en esta máquina se limitan a:

1. avanzar el cabezal lector/escritor hacia la derecha.
2. avanzar el cabezal lector/escritor hacia la izquierda.

El cómputo es determinado a partir de una tabla de estados de la forma:

$$(\text{estado}, \text{valor}) \rightarrow (\text{nuevo estado}, \text{nuevo valor}, \text{dirección})$$

Esta tabla toma como parámetros el estado actual de la máquina y el carácter leído de la cinta, dando la dirección para mover el cabezal, el nuevo estado de la máquina y el valor a ser escrito en la cinta.

Con este aparato extremadamente sencillo es posible realizar cualquier cómputo que un computador digital sea capaz de realizar.

Mediante este modelo teórico y el análisis de complejidad de algoritmos, fue posible la categorización de problemas computacionales de acuerdo a su comportamiento, apareciendo así, el conjunto de problemas denominados P y NP, cuyas soluciones en tiempo polinómico son encontradas según el determinismo y no determinismo respectivamente de la máquina de Turing.

De hecho, se puede probar matemáticamente que para cualquier programa de computadora es posible crear una máquina de Turing equivalente. Esta prueba resulta de la Tesis de Church-Turing, formulada por Alan Turing y Alonzo Church, de forma independiente a mediados del siglo XX.

La idea subyacente es el concepto de que una máquina de Turing es una persona ejecutando un procedimiento efectivo definido formalmente, donde el espacio de memoria de trabajo es ilimitado, pero en un momento determinado sólo una parte finita es accesible. La memoria se divide en espacios de trabajo denominados celdas, donde se pueden escribir y leer símbolos. Inicialmente todas las celdas contienen un símbolo especial denominado “blanco”. Las instrucciones que determinan el funcionamiento de la máquina tienen la forma, “si estamos en el estado  $x$  leyendo la posición  $y$ , donde hay escrito el símbolo  $z$ , entonces este símbolo debe ser reemplazado por este otro símbolo, y pasar a leer la celda siguiente, bien a la izquierda o bien a la derecha”. La máquina

de Turing puede considerarse como un autómata capaz de reconocer lenguajes formales. En ese sentido es capaz de reconocer los lenguajes recursivamente enumerables, de acuerdo a la jerarquía de Chomsky. Su potencia es, por tanto, superior a otros tipos de autómatas, como el autómata finito, o el autómata con pila, o igual a otros modelos con la misma potencia computacional.

■ Definición Formal:

Una máquina de Turing con una sola cinta puede ser definida como una 7-tupla

$$M=(Q, \Sigma, \Gamma, s, b, F, \delta), \text{ donde}$$

- $Q$  , es un conjunto finito de estados.
- $\Sigma$ , es un conjunto finito de símbolos distinto del espacio en blanco, denominado alfabeto de máquina.
- $\Gamma$ , es un conjunto finito de símbolos de cinta, denominado alfabeto de cinta.
- $s \in Q$  es el estado inicial.
- $b \in \Gamma$  es un símbolo denominado blanco, y es el único símbolo que se puede repetir un número infinito de veces.
- $F \subseteq Q$  es el conjunto de estados finales de aceptación.
- $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{I,D,N\}$  es una función parcial denominada función de transición, donde I es un movimiento a la izquierda, D es el movimiento a la derecha y N no realiza movimiento alguno.

Existen en la literatura un abundante número de definiciones alternativas, pero todas ellas tienen el mismo poder computacional, por ejemplo se puede añadir el símbolo S como símbolo de “no movimiento” en un paso de cómputo o el símbolo  $\Sigma$  para indicar el alfabeto de entrada.



■ Ejemplo:

Definimos una máquina de Turing sobre el alfabeto  $\{0,1\}$ , donde 0 representa el símbolo blanco. La máquina comenzará su proceso situada sobre un símbolo “1” de una serie. La máquina de Turing copiará el número de símbolos “1” que encuentre hasta el primer blanco detrás de dicho símbolo blanco. Es decir, situada sobre el 1 situado en el extremo izquierdo, doblará el número de símbolos 1, con un 0 en medio. Así, si tenemos la entrada “111” devolverá “1110111”, con “1111” devolverá “111101111”, y sucesivamente.

El conjunto de estados es  $\{s_1, s_2, s_3, s_4, s_5\}$  y el estado inicial es  $s_1$ .

La tabla que describe la función de transición es la siguiente:

Estado	Símbolo leído	Símbolo escrito	Mov.	Estado sig.
$s_1$	1	0	D	$s_2$
$s_1$	1	0	D	$s_2$
$s_2$	1	1	D	$s_2$
$s_2$	0	0	D	$s_3$
$s_3$	0	1	I	$s_4$
$s_3$	1	1	D	$s_3$
$s_4$	1	1	I	$s_4$
$s_4$	0	0	I	$s_5$
$s_5$	1	1	I	$s_5$
$s_5$	0	1	D	$s_1$

El funcionamiento de una computación de esta máquina se puede mostrar con el siguiente ejemplo (en negrita se resalta la posición de la cabeza lectora/escritora):

Paso	Estado	Cinta
1	$s_1$	<b>11</b>
2	$s_2$	0 <b>1</b>
3	$s_2$	01 <b>0</b>
4	$s_3$	010 <b>0</b>
5	$s_4$	010 <b>1</b>
6	$s_5$	0 <b>1</b> 01
7	$s_5$	<b>0</b> 101
8	$s_1$	1 <b>1</b> 01
9	$s_2$	100 <b>1</b>
10	$s_3$	100 <b>1</b>
11	$s_3$	1001 <b>0</b>
12	$s_4$	1001 <b>1</b>
13	$s_4$	100 <b>1</b> 1
14	$s_5$	1001 <b>1</b>
15	$s_1$	11 <b>0</b> 11
Parada		

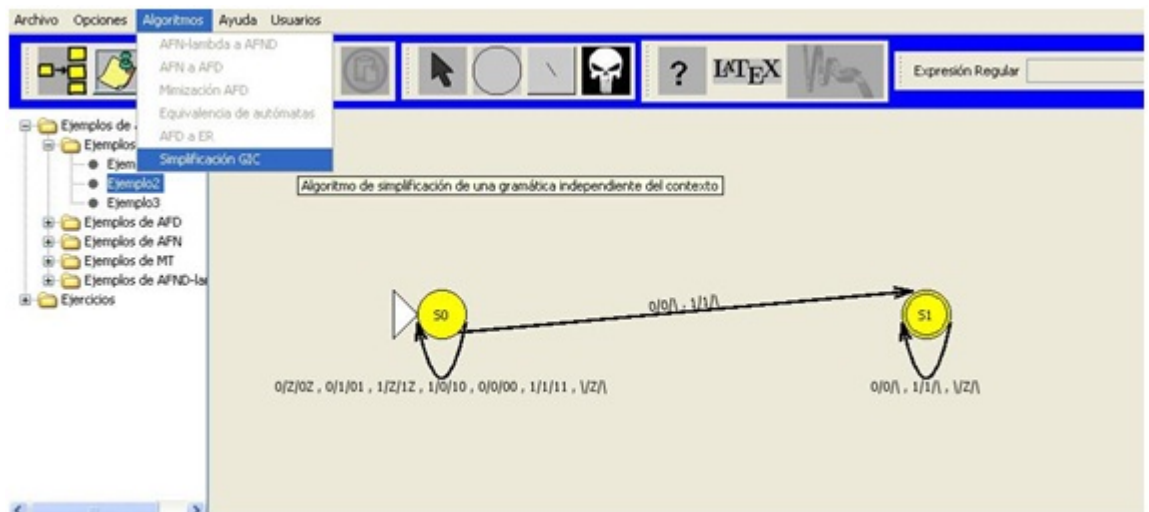
La máquina realiza su proceso por medio de un bucle, en el estado inicial  $s_1$ , reemplaza el primer 1 con un 0, y pasa al estado  $s_2$ , con el que avanza hacia la derecha, saltando los símbolos 1 hasta un 0 (que debe existir), cuando lo encuentra pasa al estado  $s_3$ , con este estado avanza saltando los 1 hasta encontrar otro 0 (la primera vez no habría ningún 1). Una vez en el extremo derecho, añade un 1. Después comienza el proceso de retorno; con  $s_4$  vuelve a la izquierda saltando los 1, cuando encuentra un 0 (en el medio de la secuencia), pasa a  $s_5$  que continúa a la izquierda saltando los 1 hasta el 0 que se escribió al principio. Se reemplaza de nuevo este 0 por 1, y pasa al símbolo siguiente, si es un 1, se pasa a otra iteración del bucle, pasando al estado  $s_1$  de nuevo. Si es un símbolo 0, será el símbolo central, con lo que la máquina se detiene al haber finalizado su cómputo.

### 3.5. Lenguaje generado por un AP

Esta parte fue la que nos llevo más tiempo, pues como hemos explicado antes, seguimos una línea de trabajo que desafortunadamente nos llevo a un callejón sin salida. Nos centramos en palabras que pertenecían al lenguaje, punto importante pero no el vital para el que sirven los autómatas: definir un lenguaje.

#### 3.5.1. Algoritmo de paso de AP $\rightarrow$ Gramática

Encontramos muchas dificultades, pues este algoritmo viene detallado muy formalmente y los ejemplos que pudimos encontrar no eran demasiado aclaratorios. Para realizar esta nueva función, hemos incluido en el menú de algoritmos el botón “Simplificación GIC”:

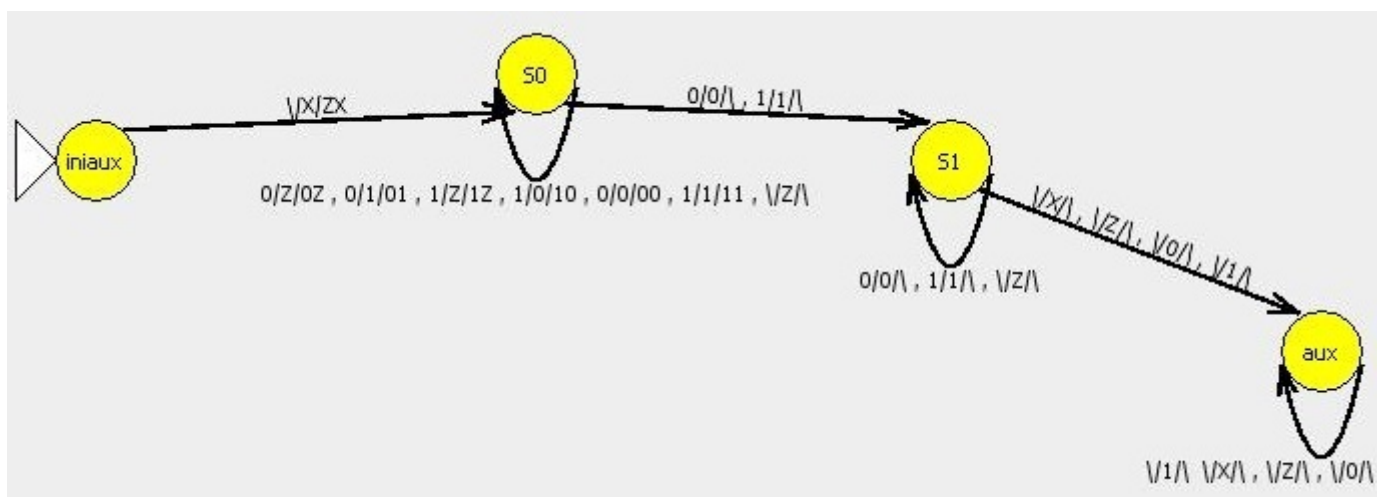


En primer lugar, tenemos que distinguir si el autómata de pila tiene estados finales o no. Si se da el primer caso, tenemos que convertir el autómata dado en otro equivalente que reconozca el mismo lenguaje por pila vacía que por estado final. Para ello, seguimos el algoritmo que nos dice cómo hacerlo:

- Se debe incluir un nuevo estado inicial y un nuevo fondo de pila. Se añade una transición vacía que vaya al estado inicial antiguo, que apile el antiguo fondo de pila sobre el nuevo.

- Se incluye un nuevo estado, que será donde se proceda a desapilar todos los símbolos que pudiera haber apilados en el momento de aceptación de la cadena cuando era aceptada por estado final. Se añade una transición vacía desde cada estado final que desapile para cada símbolo del alfabeto de pila y que vaya al nuevo estado. Por último, en este nuevo estado existirán aristas con las mismas características que las anteriores. Así, nos aseguramos que, sea posible la transición o no, se desapilaría siempre la pila.

Podemos verlo más claro en la imagen:



En ningún caso modificamos el autómata que dibuje el usuario, haremos una copia de él. Creemos que no es necesario porque es una conversión necesaria para ejecutar el algoritmo, y es bastante intuitivo el cambio, pero si en un futuro se desea se podría sustituir el original por el nuevo calculado a partir de él.

Una vez que tenemos el autómata libre de estados finales, aplicamos el algoritmo que nos convierte un autómata de pila en una gramática independiente del contexto. Consiste en lo siguiente:

- El símbolo inicial de la gramática será  $S$ . Para cada estado  $q$  del autómata, siendo  $q_0$  el estado inicial tendremos la producción  $S \rightarrow [q_0 Z q]$
- Si existe una transición tal que desapile la cima de la pila,  $\delta(p, Z, a) = (q, \epsilon)$ , añadimos la producción  $[p Z q] \rightarrow a$ . Esta  $a$  puede ser también  $\epsilon$ .
- Por último, para transiciones  $\delta(p, Z, a) = (q, A_1 A_2 \dots A_n)$ , debemos construir todas las listas de estados de longitud  $n$ , y crear para cada lista la producción:  $[p Z r_k] = a[q A_1 r_1][r_2 A_2 r_3] \dots [r_{k-1} A_n r_k]$ . Aquí  $a$  también puede ser  $\epsilon$ .

Una vez que tenemos la gramática, la simplificamos, pues en muchos casos ayuda a reducir los símbolos de variable. Sustituimos todas las variables con producciones unitarias excepto con la variable inicial, y eliminamos las producciones que sean vacías siempre y cuando no aparezcan en  $S$ . Para hacer más fácil al usuario el entender la gramática, cambiamos los símbolos de producción, que hasta el momento eran del tipo  $[q Z p]$ , por letras mayúsculas del abecedario. Creemos que es bastante grande y que no es probable que ninguna gramática rebase ese número de variables.

### 3.5.2. Algoritmo de simplificación Gramática tipo 2 a Gramática de Greibach

Cuando ya tenemos la gramática simplificada podemos dedicarnos a convertirla en una gramática de Greibach. Se caracteriza porque las producciones empiezan por un símbolo terminal, como ya sabemos.

Para poder localizar que variables tienen producciones que comiencen por otra variable, construimos una tabla cuyas filas y columnas son las variables de la gramática. Si alguna producción de las variables de las filas comienza con alguna de las variables de las columnas, marcamos la casilla correspondiente. Veámoslo paso por paso, para el ejemplo de la última imagen:

La gramática que nos genera este autómata, es la siguiente:

Gramatica	
<b>Variables:</b> [S, A, B, C, D, E, F, G, H, I, J, K, L, M]	
<b>Variable Inicial:</b> S	
<b>Simbolos Terminales:</b> [0, 1]	
<b>Producciones:</b> {D=[0,J,D, 1,F,D], E=[0,J,E, 0,K, 0,M, 1,F,E, 1,G, 1,I], F=[0,J,F, 1,F,F], G=[0,J,G, 0,K,1, 1,F,G, 1,G,1, 1], A=[C, D,A, E], B=[0,J,B, 1,F,B, \], C=[0,J,C, 0,K, 1,F,C, 1,G], L=[1,F,L, 0,J,L], M=[1,F,M, 1,G, 1,I, 0,J,M, 0,K, 0,M], H=[0,J,H, 1,F,H], I=[0,J,I, 0,K, 0,M, 1,F,I, 1,G, 1,I], J=[1,F,J, 0,J,J], K=[1,F,K, 1,G,0, 0,J,K, 0,K,0, 0], S=[A]}	

Y la tabla que se genera tal y como hemos dicho antes es la siguiente:

[illegible]



Y mientras que no marquemos una casilla diagonal, es sencillo. Seleccionamos las casillas de la columna más a la izquierda, y sustituimos creando producciones nuevas de la variable marcada en la columna, incluyéndolas en la variable de la fila. Este proceso es automático y no se muestra al usuario por pasos, se le vuelve a mostrar la gramática resultante de las sustituciones:

Gramatica	
<b>Variables:</b> [S, A, B, C, D, E, F, G, H, I, J, K, L, M]	
<b>Variable Inicial:</b> S	
<b>Simbolos Terminales:</b> [0, 1]	
<b>Producciones:</b> {D=[0,J,D, 1,F,D], E=[0,J,E, 0,K, 0,M, 1,F,E, 1,G, 1,I], F=[0,J,F, 1,F,F], G=[0,J,G, 0,K,1, 1,F,G, 1,G,1, 1], A=[C, D,A, E], B=[0,J,B, 1,F,B, \], C=[0,J,C, 0,K, 1,F,C, 1,G], L=[1,F,L, 0,J,L], M=[1,F,M, 1,G, 1,I, 0,J,M, 0,K, 0,M], H=[0,J,H, 1,F,H], I=[0,J,I, 0,K, 0,M, 1,F,I, 1,G, 1,I], J=[1,F,J, 0,J,J], K=[1,F,K, 1,G,0, 0,J,K, 0,K,0, 0], S=[C, D,A, E]}	

Y volvemos a generar la tabla, hasta que llegamos a un punto que la tabla no está marcada, y eso quiere decir que ya está en forma normal de Greibach.

[illegible]

Momento en el que ya hemos terminado y devolvemos la gramática final:

<b>Gramatica final simplificada</b>	
<b>Variables:</b> [S, A, B, C, D, E, F, G, H, I, J, K, L, M]	
<b>Variable Inicial:</b> S	
<b>Simbolos Terminales:</b> [0, 1]	
<b>Producciones:</b> {D=[0,J,D, 1,F,D], E=[0,J,E, 0,K, 0,M, 1,F,E, 1,G, 1,I], F=[0,J,F, 1,F,F], G=[0,J,G, 0,K,1, 1,F,G, 1,G,1, 1], A=[0,J,C, 0,K, 1,F,C, 1,G, 0,J,D,A, 1,F,D,A, 0,J,E, 0,M, 1,F,E, 1,I], B=[0,J,B, 1,F,B, 0,J, 1,F], C=[0,J,C, 0,K, 1,F,C, 1,G], L=[1,F,L, 0,J,L], M=[1,F,M, 1,G, 1,I, 0,J,M, 0,K, 0,M], H=[0,J,H, 1,F,H], I=[0,J,I, 0,K, 0,M, 1,F,I, 1,G, 1,I], J=[1,F,J, 0,J,J], K=[1,F,K, 1,G,0, 0,J,K, 0,K,0, 0], S=[0,J,C, 0,K, 1,F,C, 1,G, 0,J,D,A, 1,F,D,A, 0,J,E, 0,M, 1,F,E, 1,I]}	

A esta gramática que devolvemos como final, le aplicamos otro algoritmo de simplificación, el de eliminar las producciones vacías siempre que no sean producciones de la variable inicial. Este algoritmo consiste en lo siguiente:

1. Localizamos que variables, siempre que no sea la inicial, tiene una producción vacía. Eliminamos la producción vacía de sus producciones para cada una de las variables que cumplan esta característica.
2. Buscamos en el resto de las producciones de las variables si existe alguna que contenga la variable con la producción vacía. La sustituimos, y la añadimos a la lista de producciones.

Consideramos que cuando la tabla no contiene marcas es el momento ideal de proceder a eliminarlas, porque si lo hacemos antes de que esté en Forma Normal de Greibach, no podemos asegurar que entremos en un bucle infinito y no podamos salir de él. Por ejemplo:

Tenemos dos producciones del tipo:

$$\begin{aligned} A &\rightarrow \dots | E | \epsilon | \dots \\ E &\rightarrow \dots | A | \dots \end{aligned}$$

Siendo A, E variables de la gramática. Si eliminamos las producciones vacías, ocurriría lo siguiente:

1. Sustituimos las apariciones de A por  $\epsilon$ . Obtendríamos lo siguiente:

$$\begin{aligned} A &\rightarrow \dots | E | \dots \\ E &\rightarrow \dots | \epsilon | \dots \end{aligned}$$

2. Volveríamos a estar en la misma situación, pues seguimos teniendo una variable con una producción vacía.

Este problema no lo tenemos si la gramática tiene la Forma Normal de Greibach, pues nos aseguramos que el primer símbolo de la producción es un terminal, y nunca ocurrirá el caso anterior.

Por otro lado, puede ocurrir que se marque una casilla que esté en la diagonal de la tabla. En ese caso el proceso es algo más delicado, pues hemos de incluir una nueva variable, ya que existe recursividad y por mucho que se sustituya una y otra vez no podríamos eliminarla.

### 3.5.3. Algoritmo de simplificación de gramática de tipo 2 a gramática de Chomsky

Ya que hemos incluido simplificaciones de gramática, no podemos no incluir otras de las simplificaciones, quizá más importante incluso que la Forma Normal de Greibach: La Forma Normal de Chomsky.

Convertir una gramática en Forma Normal de Chomsky, consiste en dos pasos muy sencillos. En primer lugar, una gramática está en forma normal de Chomsky si las producciones son del tipo:

$$\begin{aligned} A &\rightarrow BC \text{ Siendo } A, B, C \in V \\ A &\rightarrow a \text{ Siendo } a \in T \end{aligned}$$

Y sólo se puede incluir la producción  $S \rightarrow \epsilon$  si  $S$  no aparece en ninguna de las producciones del resto de variables y, por supuesto, aparece en los casos que únicamente  $\epsilon$  es reconocida por la gramática.

Con esta idea en la cabeza, es fácilmente entendible el algoritmo de transformación, que consta de dos fases:

- Fase 1: Para cada símbolo terminal que pertenece a  $T$ , creamos producciones del tipo  $[Ca] \rightarrow a$  para todo  $a \in T$ .
  - Fase 1.1: Reemplazamos en todas las producciones las apariciones de símbolos terminales  $a$  por sus “variables terminales”  $[Ca]$  asociados que acabamos de crear.
- Fase 2: Ahora tenemos que conseguir que las producciones estén formadas solamente por dos variables. Para ello procedemos de la siguiente manera:
  - Fase 2.2: Recorremos todas las producciones. Si la longitud es mayor o igual a 3, esto es lo que hacemos:  
Imaginemos que tenemos la siguiente producción:  $A \rightarrow A_1 A_2 A_3 \dots A_n$ . Creamos una nueva variable,  $[D0]$ , que contendrá la producción  $A_2 A_3 \dots A_n$ , y modificamos la producción que acabamos de encontrar suplantándola por  $A \rightarrow A_1 [D0]$ .

Esta fase 2 se repite hasta que todas las producciones tengan longitud menor que 3, o lo que es lo mismo, que ninguna cumpla la condición para ser modificada.

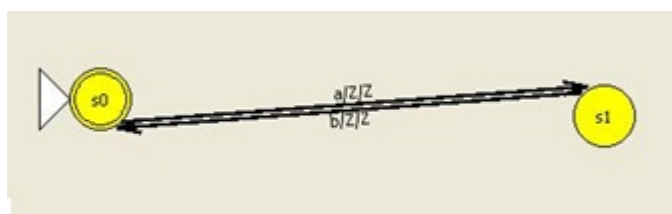
En teoría habríamos acabado aquí, pero puede ocurrir, que la gramática original, tuviera producciones de sólo una variable, y entonces no estaría la forma normal de Chomsky bien construida.

Este detalle en el algoritmo no se tiene en cuenta, pero es lo que nos ha ocurrido a lo largo de todo este año trabajando en el proyecto:

las diferencias entre las aplicaciones teóricas, y lo que en la práctica supone estudiar todos los casos, no únicamente el general, y buscar soluciones eficientes para devolver el resultado correcto. Así que, informalmente, añadiríamos un tercer paso:

- Fase 3: Si en algún momento encontramos en las producciones alguna que consista en una variable, sustituimos esa variable por todas sus producciones, pues en principio tendrán la forma adecuada que requiere la Forma Normal de Chomsky. Y finalmente diremos que la gramática está simplificada en el momento que hayamos revisado todas las producciones y no hayamos hecho ninguna modificación.

A continuación, vamos a mostrar cuál es el resultado de obtener la Forma Normal de Chomsky en un autómata sencillo:



Y la salida HTML mostrada con su Forma Normal de Chomsky, que tiene un formato idéntico a la salida de la Forma Normal de Greibach excepto en los pasos que sigue para llegar al resultado final, es la siguiente:

1. Lo primero, saber cuál es la gramática primigenia:

**Gramatica**

<b>Variables:</b> [S, A]
<b>Variable Inicial:</b> S
<b>Simbolos Terminales:</b> [a, b]
<b>Producciones:</b> {A=[a,b,A, \], S=[A]}

2. La salida obtenida de aplicar la primera fase, cuyo resultado es claramente intuitivo comparándolo con la gramática original:

\*\*\*\*\*FASE 1\*\*\*\*\*

**Gramatica**

<b>Variables:</b> [S, A, [Ca], [Cb]]
<b>Variable Inicial:</b> S
<b>Simbolos Terminales:</b> [a, b]
<b>Producciones:</b> {[Ca]=[a], A=[[Ca],[Cb],A, \], S=[A], [Cb]=[b]}



3. La salida obtenida de terminar con la segunda fase:

\*\*\*\*\*FASE 2\*\*\*\*\*

## Gramatica

<b>Variables:</b> [S, A, [Ca], [Cb], [D0]]
<b>Variable Inicial:</b> S
<b>Simbolos Terminales:</b> [a, b]
<b>Producciones:</b> {[Ca]=[a], A=[[Ca],[D0], \], S=[A], [D0]=[[Cb],A], [Cb]=[b]}

4. Observemos la producción  $S \rightarrow A$  y la producción de [D0] cómo cambia al terminar con la simplificación y conseguir una gramática en Forma Normal de Chomsky:

## Gramatica final simplificada

<b>Variables:</b> [S, A, [Ca], [Cb], [D0]]
<b>Variable Inicial:</b> S
<b>Simbolos Terminales:</b> [a, b]
<b>Producciones:</b> {[Ca]=[a], A=[[Ca],[D0]], S=[[Ca],[D0], \], [D0]=[[Cb],A, b], [Cb]=[b]}



### 3.5.4. Generación aleatoria de palabras

Como ya sabemos, los autómatas de pila definen lenguajes. El uso de la pila dificulta en algunos casos saber cómo son las cadenas reconocidas, por tanto implementamos una manera de conseguir una lista que mostrase algunas de las palabras reconocidas. Esta lista contiene como máximo cinco palabras. En un primer momento pensamos en que tuviera diez, pero para gramáticas con muchas variables, es muy probable que tengan muchas producciones para cada una de ellas, aumentando considerablemente el coste y el uso de memoria. Después de tener esta información, cogemos las producciones de la variable inicial, y creamos una lista donde iremos guardando las derivaciones por la izquierda que podríamos construir con la gramática. Y a partir de aquí, el algoritmo consiste en lo siguiente:

Mientras que la lista de palabras construidas no llegue a cinco, haremos lo siguiente:

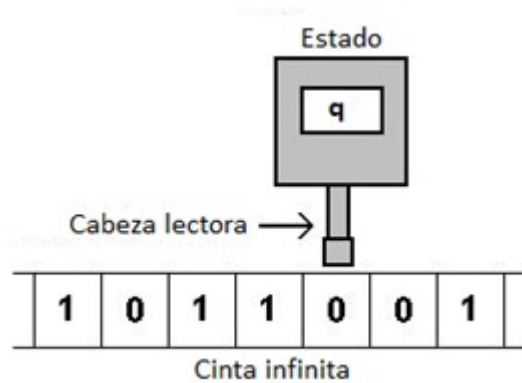
- Cogemos la primera derivación de la producción y la eliminamos de la lista.
- Si está formada solamente por símbolos terminales, la eliminamos de la lista y la metemos en la lista de palabras reconocidas por el autómata.
- Si no, buscamos el primer símbolo de variable que aparezca por la izquierda. En caso de que esa variable pertenezca a la lista que calculamos antes de variables con producciones que son un terminal, la sustituimos por todas las producciones que tenga de ese tipo, y se añaden las diferentes derivaciones resultantes para cada sustitución.

La longitud de la lista es un parámetro fácilmente modificable para futuras mejoras de la aplicación, pues está definida como una constante. Consideramos no dejarla a la elección del usuario, porque la idea de mostrar esta lista es que el usuario tenga una ligera intuición del lenguaje generado por el autómata, animando a que sea él mismo el que profundice en las cadenas que pertenecen al lenguaje y también en las que no.

## 3.6. Máquinas de Turing

### 3.6.1. Nociones preliminares

El modelo de máquina de Turing es equivalente en cuanto a potencia de cómputo a un ordenador. Las máquinas de Turing han acabado siendo una definición formal aceptada de algoritmo. Lo que no pueda computar ninguna máquina de Turing no es computable. Por tanto, la potencia de estos autómatas no es superable.



Alain Turing se planteó formalizar el comportamiento humano (como computadora) al usar lápiz y papel para resolver un problema mediante un algoritmo.

Para ello descompuso todo en procesos muy simples:

- Un número finito de estados mentales ( $Q$ ).
- Papel cuadriculado, cinta potencialmente infinita dividida en celdas.
- En un mismo paso de cómputo podemos:
  1. Cambiar de estado (mental).
  2. Alterar el contenido de la casilla de trabajo (borrar o escribir).
  3. Concentrarnos en otra casilla (la misma, derecha o izquierda).

Con esta definición y sabiendo que un lenguaje es reconocido por una máquina de Turing cuando:

$$L(M) = \{x \in A^* \mid M \text{ acepta a } x\}$$

Se ha creado un algoritmo para TALFi, el cual, dada una palabra y un autómata que representa a una máquina de Turing, nos dice si dicha palabra es reconocida por la máquina.

Para ello, se coloca la palabra en un archivo de texto (para poder trabajar de la manera más parecida posible a una cinta con celdas infinitas), y se le pasa al algoritmo como argumento para que compruebe la palabra en cuestión.

### 3.6.2. Algoritmo de reconocimiento

Una vez comprendida la formalización de Turing, la hemos usado de una manera muy intuitiva, para comprobar si la palabra que introducimos puede ser reconocida por el lenguaje que denota la máquina de Turing diseñada.

Una vez hemos diseñado una máquina de Turing en el área específica para ello en TALFi, disponemos de un botón con el cual abrir el archivo de texto que contiene la palabra a tratar.



Una vez pulsado el botón, se abrirá el usual diálogo de apertura de archivos. Seleccionado el archivo, el algoritmo pasa a realizar los siguientes pasos:

1. Busca el primer estado para iniciar el procesamiento de la cinta (archivo de texto).
2. Coloca la cabeza lectora al principio de la cinta (primer símbolo que no es un blanco).
3. Analiza que transición entre las existentes en la máquina, contienen el estado inicial y el carácter actual de la cabeza lectora.
4. Si existe la transición, actualiza el valor de la celda de la cinta y se desplaza a la celda que indique la propia transición (N = no se desplaza, I = izquierda, D = derecha. Si no existe, la palabra no será reconocida, mostrando en el archivo de texto dicha situación).
5. Vuelve al paso 3 para ver si la nueva transición existe (pero ahora con el estado y el símbolo de cinta actualizados) y en caso afirmativo, tratarla de la misma manera, modificando la cinta.

6. Si al final de todo el cómputo, la palabra resulta reconocida, la salida que se ofrecerá en la cinta (archivo de texto) de la máquina de Turing (por definición) será la celda que se encuentra inmediatamente a la derecha de la última posición de la cinta de entrada que no es un blanco de cinta.

### 3.6.3. Ampliaciones y problemas

No hemos mencionado en la anterior descripción del algoritmo, el caso en el que una máquina de Turing, no acabe nunca su cómputo, es decir, cicle o no sea terminante. Este problema no es decidible, ni siquiera existe una máquina de Turing para esto. Por ello recurrimos a una idea que si bien no será capaz de solucionar todos los casos, si un alto porcentaje de ellos. Supongamos una máquina de Turing y una palabra  $x$  dada, dicha máquina puede:

- Aceptar a  $x$ , si existe una configuración de parada y aceptadora.
- Rechazar a  $x$ , si existe una configuración de parada y no aceptadora.
- Ciclar, si no existe una configuración de parada.

Dado que en un principio este problema no fue contemplado, era necesario añadir algún tipo complemento para que se detectase el caso de ciclo en el cómputo de una máquina de Turing. Dicho complemento, es una cota que utilizan varios algoritmos conocidos. Dicha cota está relacionada con la longitud de la cinta y el número de transiciones de la máquina de Turing y su tratamiento consiste en consultar el número de transiciones visitadas hasta el momento y compararlas con dicha cota.

## 3.7. Ejercicios

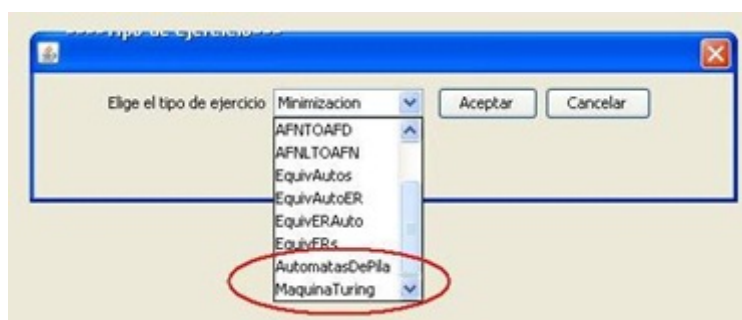
Ahora también es posible crear ejercicios de autómatas de pila y máquinas de Turing. Hemos incluido una pequeña colección, pero el administrador puede ampliarla creando sus propios ejercicios y añadiéndolos a la base de datos de los mismos. A continuación detallaremos como crearlos, que información interna generan, y la forma de corregirlos que hemos implementado.

### 3.7.1. Cambios en la interfaz gráfica e implementación interna para ejercicios de Autómatas de Pila y Máquinas de Turing

Hasta el momento, el administrador solamente tenía que escribir el autómata o la expresión regular que fuera solución, y el enunciado del ejercicio.

Para los nuevos ejercicios necesitamos más información para poder asegurar que la solución que proporcione el usuario es la suministrada por el administrador de la aplicación. Por este motivo, ahora se pueden suministrar dos listas de palabras, en el caso de autómatas de pila, y una en el caso de Turing.

Se siguen creando de la misma manera que los anteriores ejercicios, seleccionándolos de la lista de todos los posibles:



Una contendrá las palabras que han de ser reconocidas o procesadas, y otra que contendrá palabras que deben ser rechazadas. Ninguna de las dos tiene límite. Al autómata que pinte el usuario se le aplicará el algoritmo de CYK en el caso de los autómatas de pila, y si coinciden todos los resultados, se considerará el ejercicio como apto, en otro caso no se dará como válido.



The screenshot shows a web interface with a large white rectangular area at the top for drawing. Below this area, there is a text prompt: "Escribe el enunciado arriba y pinta el autómata resultante debajo". To the right of the prompt is a button labeled "Guardar ejercicio". Further right, there are two input fields: "Palabras reconocidas" and "Palabras no reconocidas".

En Turing sólo consideramos una lista, porque si las cadenas no son aceptadas no podemos asegurar que la máquina pare. Internamente se simulará la salida de la cinta cuando la máquina se detenga, y se comparará con la salida de la cinta de la máquina que proporcione el usuario. En caso de que coincidan, ocurre como en el otro tipo de ejemplos, se consideran aptos, y no aptos en otro caso.

# Capítulo 4

## Anexos

### 4.1. $\text{\LaTeX}$

La opción de que TALFi imprimiese cualquier autómeta que se dibujase en  $\text{\LaTeX}$  o mostrase los pasos de una simplificación de gramáticas en dicho formato, no fue uno de los requisitos de la especificación en un principio. Pero la opción concedida gracias a una beca del PIE (Programa de Iniciación a la Empresa), nos dio la oportunidad de añadir esta funcionalidad a la aplicación y poder trabajar con este formato de texto tan usado entre profesionales.

#### 4.1.1. Introducción a $\text{\LaTeX}$

$\text{\LaTeX}$  es un sistema de composición de textos, orientado especialmente a la creación de libros, documentos científicos y técnicos que contengan fórmulas matemáticas.

$\text{\LaTeX}$  está formado por un gran conjunto de macros de  $\text{\TeX}$ , escrito por Leslie Lamport en 1984, con la intención de facilitar el uso del lenguaje de composición tipográfica  $\text{\TeX}$ , creado por Donald Knuth. Es muy utilizado para la composición de artículos académicos, tesis y libros técnicos, dado que la calidad tipográfica de los documentos realizados con  $\text{\LaTeX}$  es comparable a la de una editorial científica de primera línea.

$\text{\LaTeX}$  es software libre bajo licencia LPPL.

### 4.1.2. Descripción:

$\text{\LaTeX}$  es un sistema de composición de textos que está formado mayoritariamente por órdenes (macros) construidas a partir de comandos de  $\text{\TeX}$ —un lenguaje “de bajo nivel”, en el sentido de que sus acciones últimas son muy elementales— pero con la ventaja añadida, en palabras de Lamport, de “poder aumentar las capacidades de  $\text{\LaTeX}$  utilizando comandos propios del  $\text{\TeX}$  descritos en *The TeXbook*”. Esto es lo que convierte a  $\text{\LaTeX}$  en una herramienta práctica y útil pues, a su facilidad de uso, se une toda la potencia de  $\text{\TeX}$ . Estas características hicieron que  $\text{\LaTeX}$  se extendiese rápidamente entre un amplio sector científico y técnico, hasta el punto de convertirse en uso obligado en comunicaciones y congresos, y requerido por determinadas revistas a la hora de entregar artículos académicos.

Su código abierto permitió que muchos usuarios realizasen nuevas utilidades que extendiesen sus capacidades con objetivos muy variados, a veces ajenos a la intención con la que fue creado: aparecieron diferentes dialectos de  $\text{\LaTeX}$  que, a veces, eran incompatibles entre sí. Para atajar este problema, en 1989 Lamport y otros desarrolladores iniciaron el llamado “Proyecto LaTeX3”. En otoño de 1993 se anunció una reestandarización completa de  $\text{\LaTeX}$ , mediante una nueva versión que incluía la mayor parte de estas extensiones adicionales (como la opción para escribir transparencias o la simbología de la American Mathematical Society) con el objetivo de dar uniformidad al conjunto y evitar la fragmentación entre versiones incompatibles de  $\text{\LaTeX}$  2.09. Esta tarea la realizaron Frank Mittlebach, Johannes Braams, Chris Rowley y Sebastian Rahtz junto al propio Leslie Lamport. Hasta alcanzar el objetivo final del “Proyecto 3”, a las distintas versiones se las viene denominando  $\text{\LaTeX}2_{\epsilon}$  (o sea, “versión 2 y un poco más...”). Actualmente cada año se ofrece una nueva versión, aunque las diferencias entre una y otra suelen ser muy pequeñas y siempre bien documentadas.

Con todo, además de todas las nuevas extensiones, la característica más relevante de este esfuerzo de reestandarización fue la arquitectura modular: se estableció un núcleo central (el compilador) que mantiene las funcionalidades de la versión anterior pero permite incrementar su potencia y versatilidad por medio de diferentes paquetes que solo se cargan si son necesarios. De ese modo,  $\text{\LaTeX}$  dispone ahora de innumerables paquetes para todo tipo de objetivos, muchos dentro de la distribución oficial, y otros realizados por terceros, en algunos casos para usos especializados.



### 4.1.3. Uso

$\text{\LaTeX}$  presupone una filosofía de trabajo diferente a la de los procesadores de texto habituales (conocidos como WYSIWYG, es decir, “lo que ves es lo que obtienes”) y se basa en comandos. Tradicionalmente, este aspecto se ha considerado una desventaja (probablemente la única). Sin embargo,  $\text{\LaTeX}$ , a diferencia de los procesadores de texto de tipo WYSIWYG, permite a quien escribe un documento centrarse exclusivamente en el contenido, sin tener que preocuparse de los detalles del formato. Además de sus capacidades gráficas para representar ecuaciones, fórmulas complicadas, notación científica e incluso musical, permite estructurar fácilmente el documento (con capítulos, secciones, notas, bibliografía, índices analíticos, etc.), lo cual brinda comodidad y lo hace útil para artículos académicos y libros técnicos.

Con  $\text{\LaTeX}$ , la elaboración del documento requiere normalmente de dos etapas: en la primera hay que crear mediante cualquier editor de texto llano un fichero fuente que, con las órdenes y comandos adecuados, contenga el texto que queramos imprimir. La segunda consiste en procesar este fichero; el procesador de textos interpreta las órdenes escritas en él y compila el documento, dejándolo preparado para que pueda ser enviado a la salida correspondiente, ya sea la pantalla o la impresora. Ahora bien, si se quiere añadir o cambiar algo en el documento, se deberá hacer los cambios en el fichero fuente y procesarlo de nuevo. Esta idea, que puede parecer poco práctica a priori, es conocida a los que están familiarizados con el proceso de compilación que se realiza con los lenguajes de programación de alto nivel (C, C++, etc.), ya que es completamente análogo.

El modo en que  $\text{\LaTeX}$  interpreta la “forma” que debe tener el documento es mediante etiquetas. Por ejemplo, “`documentclass{article}`” le dice a  $\text{\LaTeX}$  que el documento que va a procesar es un artículo. Puede resultar extraño que hoy en día se siga usando algo que no es WYSIWYG, pero las características de  $\text{\LaTeX}$  siguen siendo muchas y muy variadas. También hay varias herramientas (aplicaciones) que ayudan a una persona a escribir estos documentos de una manera más visual (LyX, TeXmacs y otros). A estas herramientas se les llama WYSIWYM (“lo que ves es lo que quieres decir”).

Una de las ventajas de  $\text{\LaTeX}$  es que la salida que ofrece es siempre la misma, con independencia del dispositivo (impresora, pantalla, etc.) o el sistema operativo (MS Windows, MacOS, Unix, GNU/Linux, etc.) y puede ser exportado a partir de una misma fuente a numerosos formatos tales como Postscript, PDF, SGML, HTML, RTF, etc. Existen distribuciones e IDEs de LaTeX para todos los sistemas operativos más extendidos, que incluyen todo lo necesario para trabajar. Hay, por ejemplo, programas para Windows como TeXnicCenter, MikTeX o WinEdt, para Linux como Kile, o para MacOS como TeXShop, todos liberados bajo la Licencia GPL. Existe además un editor multiplataforma (para MacOS, Windows y Unix) llamado Texmaker, que también tiene licencia GPL.

#### 4.1.4. Uso de $\text{\LaTeX}$ en la herramienta TALFi

Aunque  $\text{\LaTeX}$  es primordialmente un sistema de composición de textos para la creación de documentos que contengan fórmulas matemáticas, en TALFi lo usaremos para poder colocar en nuestros textos autómatas o simplificaciones paso a paso de gramáticas.

Esto es posible debido a una gran cantidad y diversidad de bibliotecas/paquetes que podemos añadir a nuestro editor de  $\text{\LaTeX}$ . Gracias a esta amplitud de posibilidades para crear gráficos complejos en  $\text{\LaTeX}$ , el trabajo inicial de poder imprimir los autómatas o las simplificaciones acabó dando como resultado 3 posibles representaciones en  $\text{\LaTeX}$ . De esta manera y debido a las diferencias entre las representaciones, el usuario podrá elegir entre un diseño sobrio y con formas claras, un diseño colorido o un diseño en forma matricial para poder modificarlo de manera más intuitiva.

Sin embargo para la representación de los pasos en la simplificación de gramáticas, sólo disponemos de un diseño, puesto que la representación se hace mediante tablas y texto plano, siendo innecesaria la inclusión de colores, formas, etc.

### 4.1.5. `LatexCodeConverter`

Esta clase, es la encargada de traducir cualquier autómata de TALFi en un archivo  $\text{\TeX}$  totalmente listo para ser compilado en cualquier entorno para  $\text{\LaTeX}$ .

- Algoritmo conversor:

La implementación del algoritmo consiste básicamente en traducir los distintos campos de los que consta un autómata en comandos  $\text{\LaTeX}$ :

1. Para cada estado del autómata se crea un círculo en el que dentro se incluye su etiqueta. Se detectan el estado inicial y los de aceptación, para los que se usan otra representación, además de su círculo.
2. Se toma cada coordenada de cada nodo del autómata TALFi para colocarlo de la misma manera en el archivo que generemos con  $\text{\LaTeX}$ .
3. Generamos las aristas viendo desde que estado parten y a cual se dirigen.
4. Cerramos el archivo.

- Representaciones:

1. Mediante biblioteca xy:

Esta es la representación más fiable y visualmente más clara, puesto que siempre que se pueda dibujar un autómata en TALFi, su representación en  $\text{\LaTeX}$  será prácticamente idéntica.

```

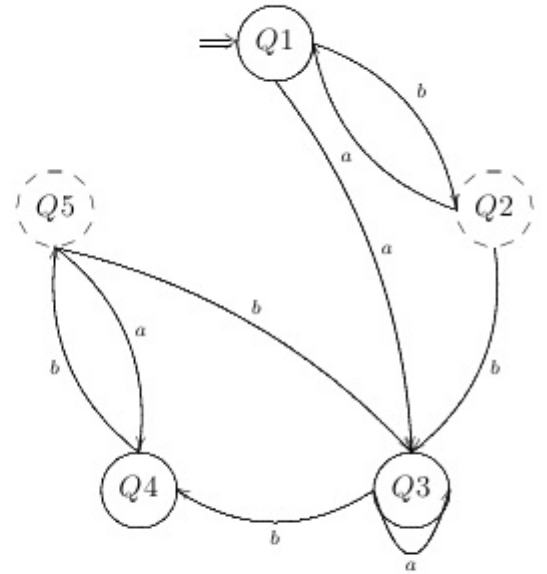
\documentclass[12pt]{article}
\input xy
\xyoption{all}
\usepackage[all, knot]{xy}
\xyoption{arc}
\begin{document}

\l[
\xy
(87,83)*{Q1};
(116,61)*{Q2};
(105,24)*{Q3};
(69,24)*{Q4};
(58,61)*{Q5};
(87,83)*\xycircle(5.00,5.00){}= "Q1";
(116,61)*\xycircle(5.00,5.00){--}= "Q2";
(105,24)*\xycircle(5.00,5.00){}= "Q3";
(69,24)*\xycircle(5.00,5.00){}= "Q4";
(58,61)*\xycircle(5.00,5.00){--}= "Q5";
\ar@{>} (77,83)*{}; (82,83)*{}
\ar@/^1pc/^b (92,83)*{}; (111,61)*{}
\ar@/^1pc/^a (87,78)*{}; (105,29)*{}
\ar@/_1pc/_b (116,56)*{}; (105,29)*{}
\ar@/_1pc/_a (111,61)*{}; (92,83)*{}
\ar@/_1pc/_b (100,24)*{}; (74,24)*{}
\ar@/_2pc/_a (100,24)*{}; (110,24)*{}
\ar@/^1pc/^b (69,29)*{}; (58,56)*{}
\ar@/^1pc/^b (58,56)*{}; (105,29)*{}
\ar@/^1pc/^a (58,56)*{}; (69,29)*{}

\endxy
\l]

\end{document}

```



## 2. Mediante biblioteca TikZ:

Esta biblioteca es visualmente muy atractiva, pero tiene el inconveniente de que el modo en el que los estados se colocan en el lienzo, se realiza especificando donde se encuentran cada uno de ellos, en relación a un estado dado. Como vemos en el código, partimos del estado A ( $q_a$  en el dibujo), para luego indicar que el estado B se encuentra arriba y a la derecha del estado A. Lo mismo ocurre con el estado C, que se indica que está abajo y a la derecha del estado B, etc.

Aunque visualmente gane muchos enteros este tipo de representación, debido a que el algoritmo construye el dibujo de manera de automática, puede llevar a muchos problemas de superposición de estados cuando los autómatas contienen un número de estados elevado (incluso podría no poder visualizarse).

Conllevaría una trabajo dificultoso de grafos para llevar a cabo una representación de autómatas totalmente libre de errores para esta representación. No obstante, si se quieren incluir autómatas en un texto  $\text{\LaTeX}$  de manera manual, este es el mejor método posible.

Lo comprobamos en un ejemplo:

```

\documentclass{article}

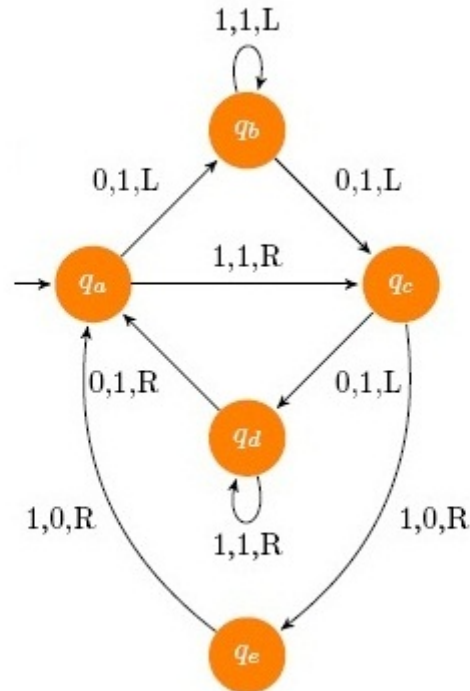
\usepackage{pgf}
\usepackage{tikz}
\usetikzlibrary{arrows,automata,positioning}
\usepackage[latin 1]{inputenc}
\begin{document}
\begin{tikzpicture}[->,>=stealth',shorten >=1pt,auto,node distance=2.5cm,semithick]
\tikzstyle{every state}=[fill=orange,draw=none,text=white]

\node[initial,state] (A) {$q_a$};
\node[state] (B) [above right of=A] {$q_b$};
\node[state] (D) [below right of=A] {$q_d$};
\node[accepting,state] (C) [below right of=B] {$q_c$};
\node[state] (E) [below of=D] {$q_e$};

\path (A) edge node {0,1,L} (B)
      (A) edge node {1,1,R} (C)
      (B) edge [loop above] node {1,1,L} (B)
      (B) edge node {0,1,L} (D)
      (C) edge node {0,1,L} (D)
      (C) edge [bend left] node {1,0,R} (E)
      (D) edge [loop below] node {1,1,R} (D)
      (D) edge node {0,1,R} (A)
      (E) edge [bend left] node {1,0,R} (A);

\end{tikzpicture}
\end{document}

```



## 3. Mediante biblioteca Psmatrix:

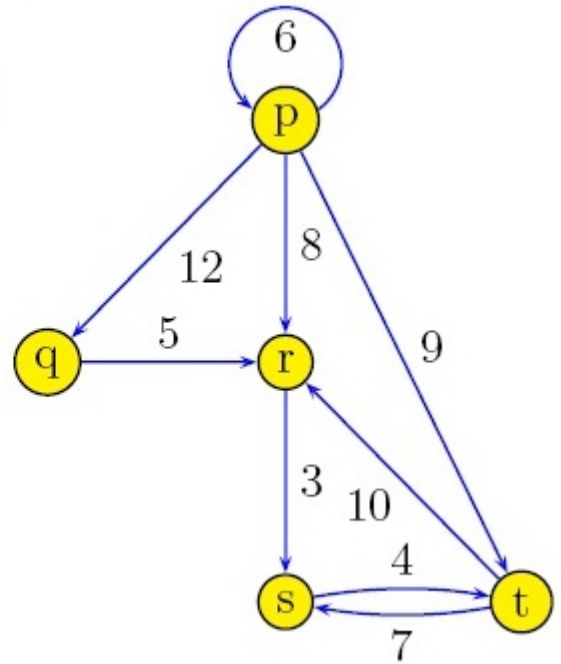
En esta implementación, no cabe posibilidad de solapamiento de estados, puesto que cada uno ocupará una “celda” de una matriz virtual que se crea en el lienzo.

Sin embargo, al igual que ocurría en el caso anterior, ante autómatas con gran cantidad de estados, nos vamos a encontrar con un problema.

En este caso el problema, será visual. Puesto que al haber tal cantidad de estados y debido a su disposición matricial, las numerosas aristas, pasarán por el centro de la matriz con toda seguridad, no permitiendo observar bien las etiquetas de las transiciones e incluso no pudiendo ver desde donde o hacia donde se dirigen.

```
\documentclass{article}
\usepackage[T1]{fontenc}
\usepackage{amsmath,amsfonts,amssymb}
\usepackage{pstricks,pstricks-add,pst-node,pst-tree}

\begin{document}
\begin{psmatrix}[fillstyle=solid,
fillcolor=yellow,mnode=circle,
colsep=1.5] & p \\ q & r & s & t
\end{psmatrix}
\psset{linecolor=blue,arrows=->,
labelsep=1mm,shortput=nab}
\ncircle{1,2}{0.5cm}^6
\ndline{1,2}{2,1}^12
\ndline{1,2}{2,2}^8
\ndline{1,2}{3,3}^9
\ndline{2,1}{2,2}^5
\ndline{2,1}{2,2}^3
\ndline{2,2}{3,2}^10
\ndline{3,3}{2,2}^4
\ncarc[arcangle=10]{3,3}{3,2}^7
\ncarc[arcangle=10]{3,2}{3,3}^4
\end{document}
```



#### 4.1.6. TraductorHTML

En TALFi 1.0 esta clase se encargaba de mostrar un archivo html en un navegador web, con los pasos de la minimización de autómatas finitos.

Ahora se ha ampliado su funcionalidad para que muestre también la simplificación de gramáticas y la anteriormente mencionada minimización, en  $\text{\LaTeX}$ .

Para ello usamos los comandos  $\text{\LaTeX}$  **tabular** y **hline** para dar un formato parecido a las tablas que se generebana en html. De esta manera podemos crear líneas verticales y horizontales para dar aspecto de tabla en  $\text{\LaTeX}$ .

El resto es introducir texto plano y los atributos de las distintas gramáticas que se van simplificando.





## Gramática

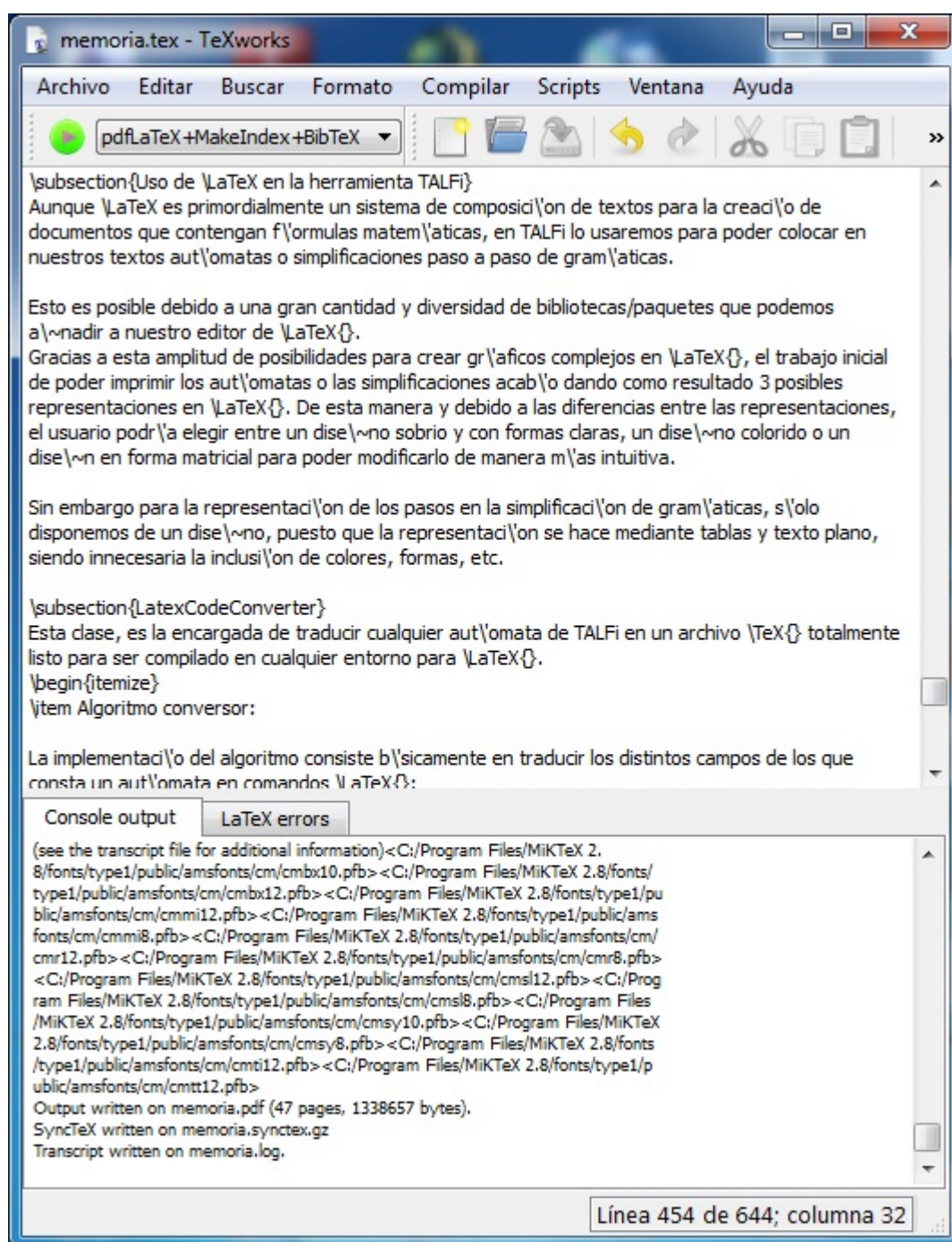
Variables: [S, A]	
Variable Inicial: S	
Símbolos Terminales: [a, b]	
Producciones: A=[a,b,A, /], S=[A]	
-	A
S	-
A	-
-	A
S	-
A	-

## Gramática final simplificada

Variables: [S, A]	
Variable Inicial: S	
Símbolos Terminales: [a, b]	
Producciones: A=[a,b,A, /], S=[a,b,A, /]	

### 4.1.7. Entornos $\text{\LaTeX}$ utilizados

- MikTeX:



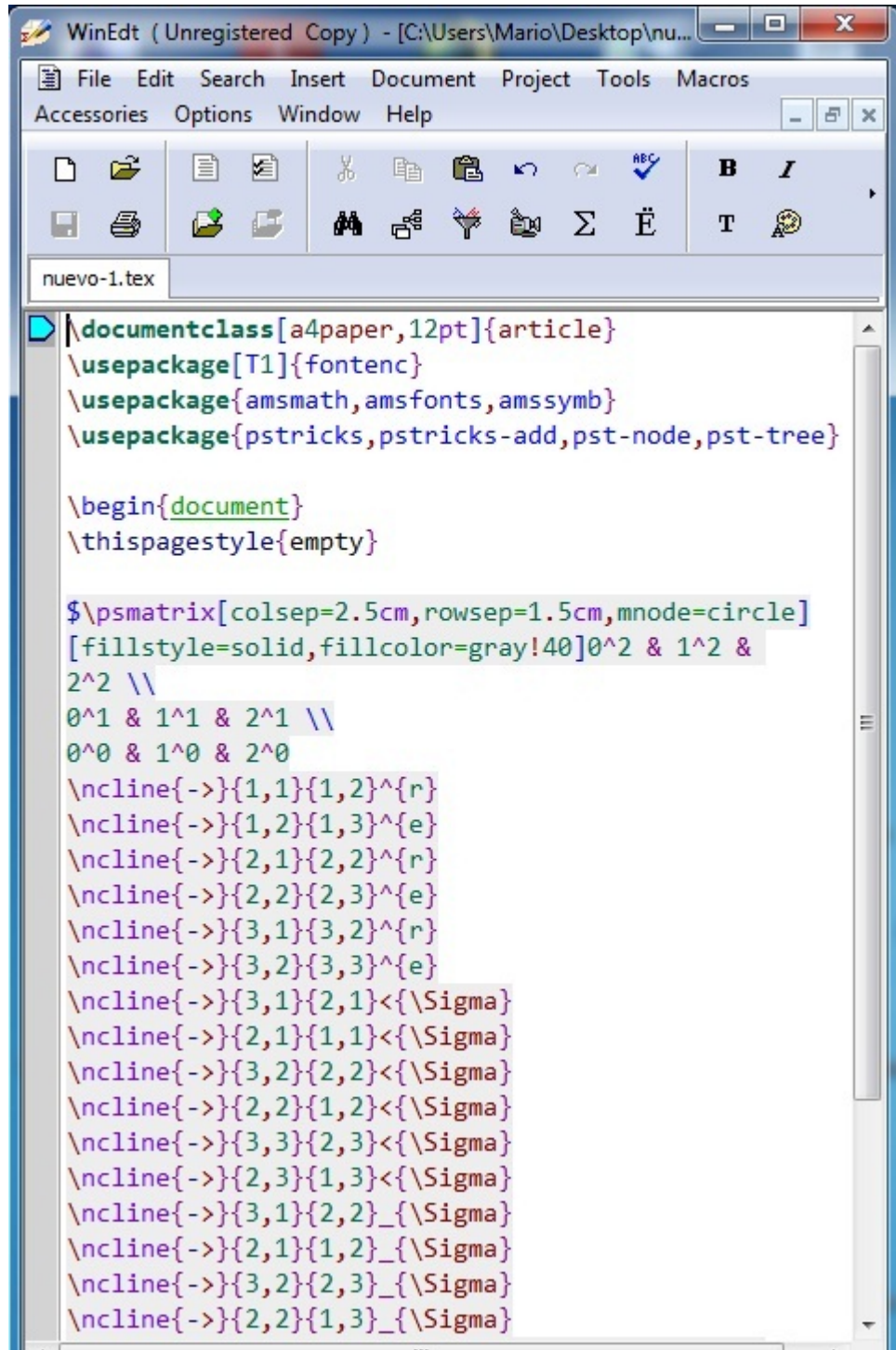
MiKTeX es una distribución  $\TeX$ / $\LaTeX$  para Microsoft Windows que fue desarrollada por Christian Schenk. Las características más apreciables de MiKTeX son su habilidad de actualizarse por sí mismo descargando nuevas versiones de componentes y paquetes instalados previamente, y su fácil proceso de instalación.

La versión actual de MiKTeX es 2.8 y está disponible en su página oficial. Además, tiene características que incluyen MetaPost y pdfTeX y compatibilidad con Windows 7. A partir de la versión 2.7 se incluyó soporte integrado para XeTeX.

Características:

- Es libre y fácil de instalar.
- Incluye más de 800 paquetes con fonts, macros, etc.
- Tiene un visor propio de archivos dvi denominado Yap.
- Su código es abierto.
- Posee compiladores  $\TeX$  y  $\LaTeX$ , convertidores para generar archivos postscripts (.ps), pdf , html , etc.; y herramientas para generar bibliografías e índices.
- Posee tres formas de instalación: pequeña, mediana y completa.

- WinEdt:



```

WinEdt (Unregistered Copy) - [C:\Users\Mario\Desktop\nu...
File Edit Search Insert Document Project Tools Macros
Accessories Options Window Help

nuevo-1.tex

\documentclass[a4paper,12pt]{article}
\usepackage[T1]{fontenc}
\usepackage{amsmath,amsfonts,amssymb}
\usepackage{pstricks,pstricks-add,pst-node,pst-tree}

\begin{document}
\thispagestyle{empty}


$$\begin{matrix}
0^2 & 1^2 & 2^2 \\
0^1 & 1^1 & 2^1 \\
0^0 & 1^0 & 2^0
\end{matrix}$$


\ncline{->}{1,1}{1,2}^{\text{r}}
\ncline{->}{1,2}{1,3}^{\text{e}}
\ncline{->}{2,1}{2,2}^{\text{r}}
\ncline{->}{2,2}{2,3}^{\text{e}}
\ncline{->}{3,1}{3,2}^{\text{r}}
\ncline{->}{3,2}{3,3}^{\text{e}}
\ncline{->}{3,1}{2,1}<\Sigma
\ncline{->}{2,1}{1,1}<\Sigma
\ncline{->}{3,2}{2,2}<\Sigma
\ncline{->}{2,2}{1,2}<\Sigma
\ncline{->}{3,3}{2,3}<\Sigma
\ncline{->}{2,3}{1,3}<\Sigma
\ncline{->}{3,1}{2,2}_\Sigma
\ncline{->}{2,1}{1,2}_\Sigma
\ncline{->}{3,2}{2,3}_\Sigma
\ncline{->}{2,2}{1,3}_\Sigma

```

WinEdt es un editor de textos de gran alcance y versatilidad para Windows, con una fuerte predisposición hacia la creación de documentos  $\text{\LaTeX}$ . La página de descargas del sitio web tiene más información sobre WinEdt,  $\text{\TeX}$ , y enlaces a otros programas necesarios para hacer operativo WinEdt en la plataforma Windows.

## 4.2. Ampliación del Manual de Usuario

Antes de que cualquier persona, administrador o no, quiera usar TALFi 2.0, debe conocer dos detalles para poder usar la aplicación correctamente:

- Si quiere incluir  $\epsilon$  en las transiciones, tendrá que hacerlo de la misma forma que en TALFi 1.0, con el carácter `/`.
- Para especificar un blanco en máquinas de Turing, tendrá que escribir `#`.
- Las direcciones posibles de movimiento en máquinas de Turing son I, D, N para español, y L,R,N para inglés. Ambas siempre en mayúsculas.
- Necesariamente el contenido de la cinta para simular una máquina de Turing debe cargarse en un archivo con extensión `“txt”`.

Al igual que en la primera versión de TALFi, cuando se pide introducir los símbolos, ya sean de cinta o de alfabeto, en ningún caso deben contener espacios y tienen que ir separados por comas.

## 4.3. Cambios en la implementación

A continuación enumeraremos brevemente las nuevas clases y los nuevos paquetes creados en este proyecto.

Ampliación de paquetes:

- Modelo.algoritmos:
  - Clases añadidas:  
AceptaTuring, AutomataP\_to\_GramaticaIC, GIC\_to\_FNG, GIC\_to\_FNChomsky
  - AceptaTuring:  
Contiene el algoritmo que simularía la ejecución de una máquina de Turing. Recibe el objeto creado para máquinas de Turing y la ruta del archivo que contiene la cinta. Aquí se abre, si ocurriese algún problema no se realiza la simulación.
  - AutomataP\_to\_GramaticaIC:  
Dado un autómata de pila aplicamos el algoritmo que nos genera su gramática asociada.
  - GIC\_to\_FNC:  
Recibe la gramática que ha generado AutomataP\_to\_GramaticaIC, y la transforma en Forma Normal de Greibach con el algoritmo que utiliza las tablas de reemplazo.
  - GIC\_to\_Chomsky:  
Al igual que GIC\_to\_FNC, genera la correspondiente Forma Normal de Chomsky a partir de la gramática obtenida en AutomataP\_to\_GramaticaIC.
- Modelo.automatas:
  - Alfabeto\_Pila: Interface con las operaciones de alfabetos de pila.
  - AlfabetoPila\_imp:  
Implementa Alfabeto\_Pila y es la clase que como su nombre indica crea y contiene información del alfabeto de pila.
  - AlfabetoCinta:  
Crea el alfabeto de cinta de una máquina de Turing.
  - AutomataPila:  
Sirve para crear objetos que identifiquen autómatas de pila.
  - MaquinaTuring:  
Sirve para crear objetos que identifiquen máquinas de Turing.

- Vista.vistaGrafica:
  - AristaGeneral:  
Clase abstracta con todos los atributos y métodos comunes a todos los tipos de aristas.
  - Arista:  
Arista utilizada para todos los autómatas finitos.
  - AristaAP:  
Arista que se utiliza en autómatas de pila.
  - AristaTuring:  
Arista que se emplea en máquinas de Turing.
- Vista.vistaGrafica.events:
- OyenteItemPopupAristaAP:  
Crea el cuadro de diálogo cuando se modifica una arista de un autómata de pila.
- OyenteItemPopupAristaTuring:  
Crea el cuadro de diálogo cuando se modifica una arista de una máquina de Turing.
- OyenteModificaAristaAPActionListener:  
Procesa los nuevos atributos que se han cambiado al modificar la arista de un autómata de pila si se pulsa aceptar con el ratón.
- OyenteModificaAristaTuringActionListener:  
Procesa los nuevos atributos que se han cambiado al modificar la arista de una máquina de Turing si se pulsa aceptar con el ratón.
- OyenteModificaAristaAPKeyAdapter:  
Procesa los nuevos atributos que se han cambiado al modificar la arista de un autómata de pila si se aceptan pulsando Intro.
- OyenteModificaAristaTuringKeyAdapter:  
Procesa los nuevos atributos que se han cambiado al modificar la arista de una máquina de Turing si se aceptan pulsando Intro.



Creación de paquetes:

- Modelo.gramatica:
  - Chomsky:  
Crea objetos que representen una gramática en Forma Normal de Chomsky.
  - Gramatica:  
Clase abstracta con los atributos y métodos comunes a todas las gramáticas que generamos en esta aplicación.
  - GramaticaIC:  
Clase que extiende a Gramatica, y cuyos objetos representan la gramática resultante antes de transformarla en ninguna Forma Normal.
  - Greibach:  
Crea objetos que representen una gramática en Forma Normal de Greibach.
  - Produccion:  
Crea los objetos que procesaremos como producciones.

## 4.4. Cambios en la interfaz gráfica

Ante todo hemos de decir que nuestro objetivo ha sido incluir las nuevas funciones de TALFi alterando lo mínimo posible el diseño que hasta el momento tenía la aplicación.

- Crear un nuevo autómatas de pila / máquina de Turing.

Para no saturar más de botones, no hemos incluido uno que fuera un acceso directo para crear un nuevo ejemplo de máquina de Turing o autómatas de pila. Si el usuario desea un nuevo ejemplo de estas características, tendrá que dirigirse y hacer click en el apartado “Archivo” del menú superior:



- Botones:

La segunda novedad son los tres botones para la generación aleatoria de palabras para autómatas de pila, cargar la cinta inicial de la máquina de Turing y la generación del código en  $\text{\LaTeX}$  del autómatas que visualizamos en el canvas.

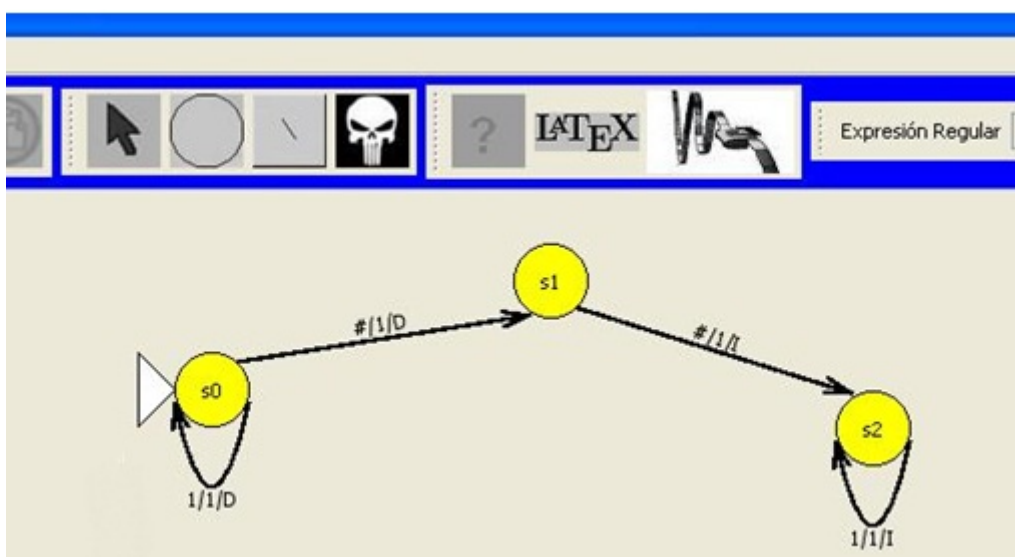
Al comienzo de la aplicación aparecen los tres desactivados, y según las áreas que trabajemos se irán activando si corresponde.

Lo mismo ocurre con la sección algoritmos, las opciones están únicamente disponibles cuando sea lógico el poder aplicarlas.

Menú al iniciar la aplicación:



Muestra de los botones activos en un ejemplo de máquina de Turing:



- Aristas:

Ahora las aristas contendrán tres campos separados por el carácter / : símbolos de entrada o símbolos de cinta, cima de pila actual o símbolo que reemplazará al que actualmente está en la cinta, y la transición de la pila o la dirección hacia la que se desplazará la cabeza lectora de la cinta de Turing, ya se trate de autómatas de pila o máquinas de Turing. En consecuencia, necesitaremos tres campos distintos a modificar cuando necesitemos modificar una arista ya dibujada en el canvas.

Cuadro de diálogo que aparece al modificar una arista de un autómata de pila:

Modificar arista

Símbolos de la arista (pueden ser varios)

Cima de pila

Símbolos de pila a apilar / desapilar

Desde: s0 Hasta: s1

Aceptar Cancelar

# Bibliografía

- [1] Hopcraft, Motwani, Ullman.  
“Introducción a la teoría de autómatas. Lenguajes y computación”  
Addison Wesley 2002.
- [2] J. G. Brookshear.  
Teoría de la Computación: Lenguajes Formales, Autómatas y Complejidad.  
Addison-Wesley Iberoamericana, 1993.
- [3] J.C. Martin.  
Introduction to Languages and the Theory of Computation  
McGraw-Hill, 1997, (Segunda edición).
- [4] D.C. Kozen  
Automata and Computability  
Springer Verlag, 1997.
- [5] H.R. Lewis, C.H. Papadimitriou  
Elements of the Theory of Computation  
Prentice Hall, 1988, (Segunda edición).
- [6] P. Isasi, P. Martínez, D. Borrajo  
Lenguajes, Gramáticas y Autómatas, un enfoque práctico.  
Addison-Wesley, 1997.
- [7] Dean Kelley.  
“Teoría de Autómatas y Lenguajes Formales”.  
Prentice Hall, 1997.
- [8] <http://es.wikipedia.org>