

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*



Se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores abajo firmantes, tanto la propia memoria, como el código, la documentación y / o el prototipo desarrollado.

Fdo.:

Luis San Juan Germán:

A handwritten signature in black ink. It includes the name "Luis" on the left, "San Juan" in the middle, and "Germán" on the right. There is also a stylized surname "Fernández" written above the main name.

Mario Jose Huete Jiménez:

Rocío Barrigüete Ibáñez:

Índice general

1. Resumen	5
1.1. TALFi (Versión en castellano)	5
1.2. TALFi (English version)	5
2. Objetivos	7
3. Información	9
3.1. Antecedentes	9
3.2. Problemas encontrados y resueltos en TALFi	10
3.2.1. Minimización	10
3.2.2. Intento de reconocimiento de palabras simulando la ejecución de un autómata de pila	13
3.3. Introducción a la ampliación de la aplicación	16
3.4. Conceptos teóricos de la ampliación	18
3.4.1. Autómata de pila	18
3.4.2. Gramática independiente de contexto	22
3.4.3. Lenguaje independiente de contexto	25
3.4.4. Máquina de Turing	26
3.4.5. Algoritmo CYK	31
3.5. Lenguaje generado por un autómata de pila	33
3.5.1. Pasos de simplificación antes y después de realizar trans- formaciones FNC o FNG	33
3.5.2. Algoritmo de paso de AP → gramática	35
3.5.3. Algoritmo de simplificación gramática independiente de contexto a FNG	37

3.5.4. Algoritmo de simplificación gramática independiente de contexto a FNC	41
3.5.5. Generación aleatoria de palabras	45
3.5.6. Palabras reconocidas por un autómata de pila	46
3.6. Máquinas de Turing	46
3.6.1. Diferencias entre máquinas de Turing con y sin estados finales	46
3.6.2. Algoritmo de reconocimiento	47
3.6.3. Ampliaciones y problemas	48
3.7. Cambios destacables en TALFi 2.0	48
3.7.1. Interfaz gráfica	48
3.7.2. Ejercicios	
	53
4. Entorno L^AT_EX	57
4.1. L ^A T _E X	57
4.1.1. Introducción a L ^A T _E X	57
4.1.2. Descripción:	57
4.2. Uso	58
4.2.1. Uso de L ^A T _E Xen la herramienta TALFi	
	60
4.3. LatexCodeConverter	
	60
4.4. TraductorHTML	66
4.5. Entornos L ^A T _E X utilizados	67
5. Anexos	69
5.1. Ampliación del Manual de Usuario	69
5.2. Cambios en la implementación	70
5.3. Estructura XML de los ejercicios de autómatas de pila	72
5.4. Estructura XML de los ejercicios de máquinas de Turing	76
6. Palabras clave	81

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. Como nueva funcionalidad podemos encontrar el tratamiento de autómatas de pila, máquinas de 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 united-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 levels 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 de 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:

- La inclusión de autómatas de pila y máquinas de Turing requirió nuevas clases para implementar su comportamiento, así como nuevos elementos gráficos para su tratamiento por parte del usuario (botones para introducir la cinta de una máquina de Turing, nuevos ejemplos de las nuevas representaciones . . .).
- Posteriormente fue necesario diferenciar en cierta manera un autómata de pila de un autómata de pila determinista. Aunque TALFi no ofrece la posibilidad al usuario de decidir si el autómata de pila que introduce es o no determinista, internamente TALFi si distingue este caso, ya que es una característica importante de estos autómatas dado que por naturaleza son no deterministas.
- De forma secundaria y sin formar parte de los objetivos, tuvieron que incluirse gramáticas independientes de contexto, en sus formas normales de Chomsky y Greibach. Esto era necesario porque, parte muy importante de TALFi (el algoritmo de reconocimiento de una palabra por un autómata de pila) requería poder pasar de gramática a autómata de pila y viceversa.
- Como parte importante de los objetivos, podemos mencionar las mejoras y arreglos realizados en TALFi 1.0:

- Minimización: la minimización en TALFi 1.0 tuvo que ser corregida, sólo se podía aplicar a aquellos autómatas que fueran finitos y deterministas. Gracias a TALFi 2.0 la minimización es posible además para autómatas finitos no deterministas y autómatas finitos no deterministas con transiciones vacías, gracias al previo paso de transformación de estos dos últimos tipos de autómatas en un autómata finito determinista. Para cerrar de manera final la minimización se imprime en un archivo HTML de forma detallada las tablas que se generan en este algoritmo para que quedasen más claros los pasos que se van realizando.^{3.2.1}
- TALFi 1.0 diferenciaba entre ejercicios y ejemplos, pero era posible ver la solución de un ejercicio propuesto por el profesor, si lo abríamos como ejemplo.^{3.7.2} Este comportamiento no nos pareció muy adecuado debido al carácter docente y de aprendizaje de TALFi. Dicho comportamiento fue subsanado, además de incluir mayor número de ejemplos y ejercicios.
- Inclusión de un modo L^AT_EX 4.1 con el que podemos obtener en dicho formato, tanto los autómatas que se creen en TALFi, como los ejemplos, así como la simplificación de las gramáticas anteriormente mencionadas.
- Como posible mejora futura, proponemos incluir TALFi 2.0 como una aplicación web dentro de un entorno como, por ejemplo, el Campus Virtual UCM. Evidentemente la herramienta seguiría teniendo las mismas aplicaciones, pero la identificación de usuario ya no sería necesaria dentro de la aplicación: estaría ligada al tipo de login que se realizase en la web. No sería la única adaptación necesaria, pues el envío de ejercicios propuestos también requiere hacer uso de los medios de comunicación con tutor del Campus Virtual UCM, para poder hacer llegar al administrador de la aplicación la solución realizada por el alumno y terminar la corrección. En TALFi 1.0 ya se diferenció los tipos de usuarios que tendría la aplicación, simulando el comportamiento y garantizando su eficiencia.

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>
Crea y simula autómatas finitos y máquinas de Turing, permite guardar la información en archivo, y abrir varios de estos archivos a la vez.
- <http://www.cs.duke.edu/csed/jflap/>
Quizá el programa más conocido, puesto que se usa en la asignatura de TALF. Permite crear autómatas finitos, expresiones regulares, algoritmos de conversión entre autómatas finitos y a expresión regular, autómatas de pila, gramáticas independientes de contexto y simulación de máquinas de Turing.
- <http://www.ucse.edu.ar/fma/sepa/>
Es español. Permite diseñar, desarrollar y evaluar un conjunto de herramientas de software que ayude al personal docente en la enseñanza de las teorías y tecnologías involucradas en los procesos de diseño y construcción de traductores de lenguajes formales.

- <http://www.brics.dk/automaton/>

Trata autómatas finitos, expresiones regulares, y operaciones de cierre sobre lenguajes regulares.

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 antecesores.

Además, resaltar que los programas que trabajan con teoría de autómatas y lenguajes formales están más enfocados al estudiante que a los profesores que imparten esta enseñanza, y ninguno de los mencionados está concebido para ser utilizado como una herramienta de comunicación y evaluación. Es el rasgo principal de TALFi, que diferencia dos perfiles de usuario, y más que un apoyo en el estudio se enfoca a un posible uso real al impartir una asignatura. Y por último lo que es único en talfi es la salida que genera en HTML de los algoritmos aplicados, y por supuesto no tienen la opción de generar código en L^AT_EX.

3.2. Problemas encontrados y resueltos en TALFi

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 qué partes encontramos dificultades.

3.2.1. Minimización

Mientras 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.

En primer lugar, introducimos un estado trampa llamado “tramposo”, para aclarar el estado a dónde llegan las transiciones que no están definidas ni tienen una arista equivalente en autómatas finitos no deterministas y autómatas finitos con transiciones vacías.

En segundo lugar, ampliamos la información que aparece en las casillas de la tabla que se construye para comprobar que estados son colapsables en

el algoritmo de minimización. Anteriormente, dentro de las celdas podíamos encontrar:

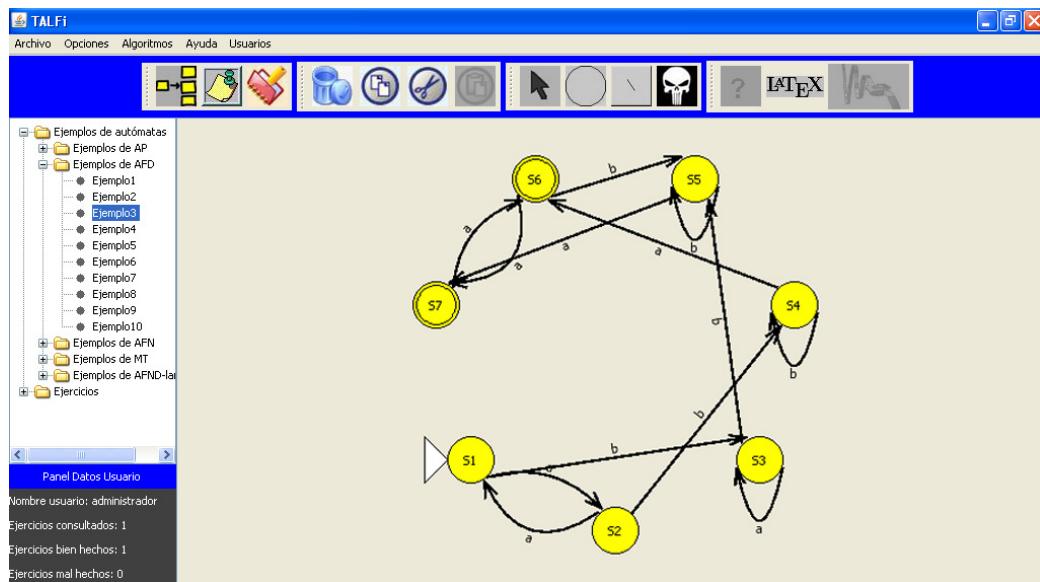
- Una “X” si los estados no se colapsaban.
- Un “+” si los estados son colapsables.

Ahora, ampliando la información de la siguiente manera:

- Si los estados no son colapsables, damos los motivos de por qué: uno puede pertenecer a los estados finales y el otro no, o si es causado por otros dos estados que se sabe que no son colapsables y tienen también su casilla marcada. Además, como la tabla la revisamos hasta que no hayamos marcado ninguna nueva casilla, incluimos en que número de vuelta hemos realizado el cambio en la celda.
- Si los estados son colapsables dejamos la casilla en blanco.

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		
trámposo	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	

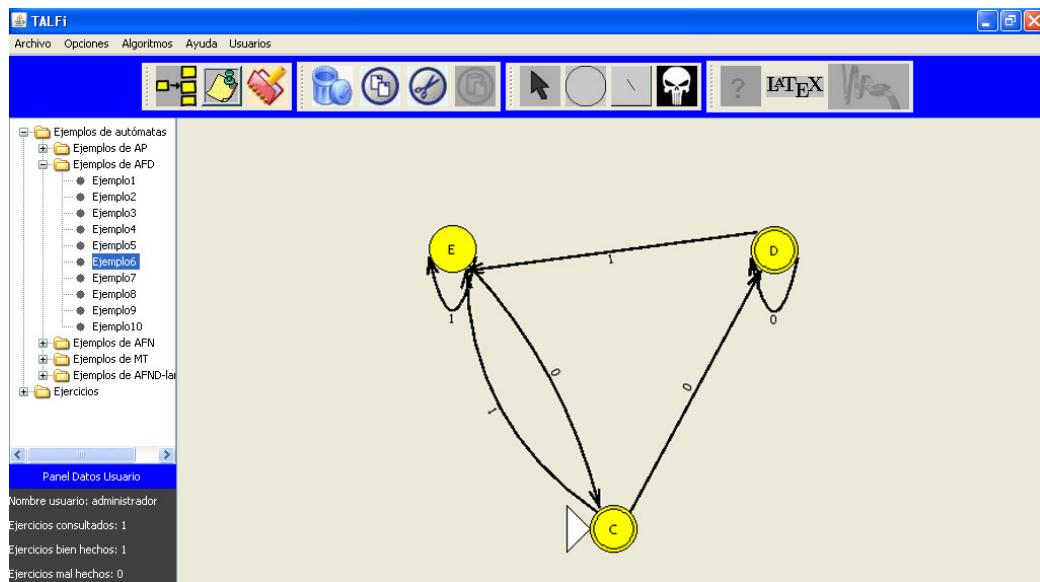
En cada celda podemos ver:

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. Para terminar la explicación de por qué se marca esa casilla, debajo o al lado indicamos que par de estados son los causantes.

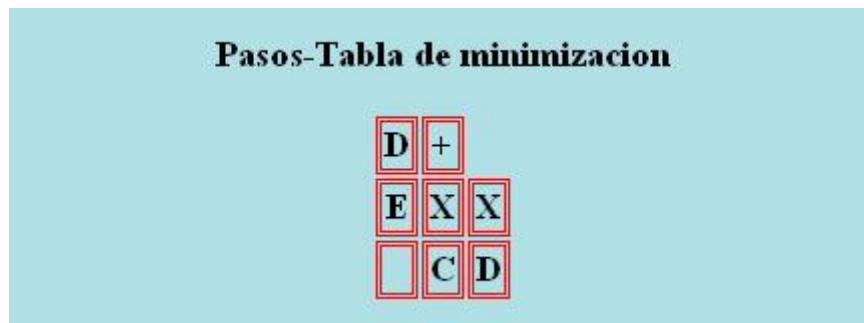
Si los estados pudieran colapsarse, la casilla no tendrá nada escrito dentro de ella.

Aquí podemos ver qué salida HTML se generaba para un ejemplo sencillo, el ejemplo 6 de autómatas finitos deterministas en TALFi 1.0.

Autómata sobre el que aplicamos el algoritmo de minimización:



Y la tabla que se generaba era la siguiente:



3.2.2. Intento de reconocimiento de palabras simulando la ejecución de 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 leía los símbolos de una cadena.

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 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: Incluído 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 qué 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.

Nos dimos cuenta que el algoritmo no terminaba si incluíamos ciclos de transiciones vacías en los que la pila no se modificase o sufriera modificaciones que no acercaban la ejecución a su final. Esto se puede arreglar de dos maneras:

- Dando prioridad a las aristas que consuman símbolos, pues si alguna sale de alguno de los estados que componen el ciclo tendremos la oportunidad de seguir el cómputo por otro camino, esperando que llegue a su fin. En algunos casos es una solución válida, pero no es suficiente pues podría no llevarnos a un estado que estuviera fuera de los estados que componen el ciclo, y seguiríamos teniendo el mismo problema.
- Fijando una cota que sólo utilizaremos en los casos en que simulando la ejecución del autómata hayamos elegido seguir el camino marcado por una arista que no consuma símbolos. Si entramos en un ciclo como los ya descritos, comenzamos a acumular las veces que ejecutamos el algoritmo. Si ejecutamos (*número de estados * número de aristas*) movimientos y no hemos avanzado en nuestro procesamiento de la cadena, asumimos que el algoritmo no terminará forzando así que finalize la ejecución.

El algoritmo se basa en dos métodos: uno que indica si se reconoce o no una cierta cadena por estado final, y otro que realiza la misma acción pero vaciando la pila. Los dividimos por un lado para mejorar la eficiencia, ya que se puede dar el caso de que el autómata de pila no tenga ningún estado final, o ninguna arista desapile el símbolo de fondo de pila, y devolver más información que únicamente concretar si la palabra es aceptada o no, dando a conocer al usuario por cuál de los dos métodos reconoce una cadena el autómata de pila.

- Método reconocedor de palabras por estado final: Se comprobará que la cadena, que puede ser o si es ϵ o una cadena formada por uno o más símbolos del alfabeto de entrada es reconocida si hemos llegado a un estado a partir del cual no podemos ir a ningún otro comprobando

la longitud de la cadena la cadena a reconocer, o en cambio si tiene transiciones posibles donde moverse pero hemos alcanzado el final de la cadena. En cualquier otro caso, se continúa con la simulación.

- Método reconocedor de palabras por pila vacía: Se comprobará si cuando llegamos a un estado donde ya no podemos movernos más la pila está vacía o no, y si hemos alcanzado la longitud de la cadena, que podrá ser también ϵ o una cadena formada por uno o más símbolos del alfabeto de entrada. De forma idéntica al método reconocedor por estado final, si no estamos en ese caso, comprobamos cómo se quedaría la pila para el siguiente paso a realizar, y si es vacía revisamos la longitud de la cadena. En caso contrario la ejecución sigue su curso.

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 a exponencial, volviéndose intratable. Optamos por el algoritmo llamado CYK 3.4.5, cuyo coste es mucho mejor, ya que es cúbico sobre la longitud de la cadena.

3.3. Introducción a la ampliación de la aplicación

TALFi 1.0 es capaz de crear autómatas finitos deterministas, no deterministas, con transiciones vacías y expresiones regulares. Como algoritmos importantes cuenta con:

- Transformación de un autómata con transiciones vacías a uno no determinista.
- Transformación de un autómata no determinista a uno determinista.
- Transformación de autómata finito determinista a expresión regular.
- 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.

TALFi 2.0 respeta esta estructura y le añade una colección de ejemplos de ejercicios de autómatas de pila y máquinas de Turing. Sobre los primeros, podremos realizar un estudio a fondo del lenguaje independiente de contexto que generan, obteniendo los siguientes resultados:

- Lista de palabras aleatorias que pertenecen al lenguaje.
- Gramática independiente de contexto que genera un autómata de pila aplicando el algoritmo de transformación.
- Simplificación detallada por pasos de la gramática que se obtiene directamente de un autómata de pila.
- Transformación de una gramática independiente de contexto en forma normal de Greibach.^{3.4.2}
- Transformación de una gramática independiente de contexto en forma normal de Chomsky.^{3.4.2}

En lo que se refiere a máquinas de Turing, podremos:

- Saber si una cadena de entrada es reconocida por una máquina de Turing. ^{3.6.2}
- Obtener la cinta de salida generada por una máquina de Turing.
- Conocer si una cierta cinta de entrada hace que una máquina de Turing cicle.

En cuanto a los ejercicios, sobretodo en los de máquinas de Turing, los usuarios podrán comprobar fácilmente antes de mandar su solución al enunciado propuesto que realmente es correcto. El administrador de la aplicación, que será el creador de los ejercicios, determinará la dificultad de los mismos, ya que además de dibujar la solución, debe incluir listas de palabras que se simularán internamente, sirviendo como filtro para la corrección.

- En los autómatas de pila, se deben llenar la lista de palabras reconocidas por el autómata solución, y también la lista de palabras que deberán ser rechazadas.
- En máquinas de Turing, según sea necesaria la cinta resultante del cómputo o no, deberán especificarse la lista de palabras aceptadas con su correspondiente cinta de salida si procede, la lista de palabras rechazadas junto con la cinta de salida si es que hace falta, y por último una lista de palabras con las que la máquina nunca llegaría a detenerse.

Y destacamos la salida en L^AT_EX de cualquier tipo de autómata y máquinas de Turing, ya sea únicamente el diagrama que tienen asociados, como cualquier resultado de aplicación de algoritmos generada en HTML.

3.4. Conceptos teóricos de la ampliación

3.4.1. Autómata de pila

El autómata de pila es, en esencia, un autómata finito no determinista en el que se permiten transiciones- ϵ y con una capacidad adicional: una pila en la que se puede almacenar una cadena de “símbolos de pila”. La presencia de una pila significa que, a diferencia de los autómatas finitos, el autómata de pila puede “recordar” una cantidad infinita de información. Al autómata de pila se le permite observar el símbolo de entrada y el símbolo de lo alto de la pila. Alternativamente, podría hacer una transición “espontánea” usando ϵ como entrada en vez de leer un símbolo de entrada. En una transición, el autómata de pila:

1. Consumo de la entrada el símbolo que usa una transición. Si se usa ϵ como entrada no se consume ningún símbolo.
 2. Pasa a un nuevo estado, que podría ser o no el mismo que el estado anterior.
 3. Reemplaza el símbolo que está en lo alto de la pila por cualquier cadena. La cadena puede ser ϵ , que corresponde a una extracción de la pila. Podría ser el mismo símbolo que aparecía anteriormente en lo alto de la pila; es decir, no se hace ningún cambio en la pila. Podría también reemplazar el símbolo de lo alto de la pila por cualquier otro símbolo, lo que cambia la parte superior de la pila, pero no quita o añade ningún elemento. Finalmente el símbolo de lo alto de la pila podría ser reemplazado por dos o más símbolos, lo que tiene el efecto de cambiar o no el símbolo de lo alto de la pila e introducir uno o más símbolos adicionales.
- Definición formal de los autómatas de pila:

Nuestra notación formal para un autómata de pila incluye siete componentes. Escribimos la especificación de un autómata de pila P como sigue:

$P = (\mathcal{Q}, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ Las componentes tienen los siguientes significados:

- \mathcal{Q} : Un conjunto finito de estados, como los estados de un autómata finito.
- Σ : un conjunto finito de símbolos de entrada, también análogo a la correspondiente componente de un autómata finito.
- Γ : Un alfabeto de pila finito. Esta componente, que no tiene análogo en un autómata finito, es el conjunto de símbolos que se permite introducir en la pila.
- δ : La función de transición. Como para un autómata finito, δ goberna el comportamiento del autómata. Formalmente, δ toma como argumento $\delta(q, a, X)$, donde:
 1. q es un estado en \mathcal{Q} .
 2. a es, o bien un símbolo de entrada en Σ , o bien $a = \epsilon$, la cadena vacía, que se supone que no es un símbolo de entrada.
 3. X es un símbolo de pila, es decir, un elemento de Γ .

La salida de δ es un conjunto finito de pares (p, γ) , donde p es el nuevo estado y γ es la cadena de símbolos de pila por los que se reemplaza X en lo alto de la pila. Por ejemplo, si $\gamma = \epsilon$ se saca un elemento de la pila; si $\gamma = X$ la pila no se cambia, y si $\gamma = YZ$, X se reemplaza por Z y se mete Y en la pila.

- q_0 : El estado inicial. El autómata de pila está en este estado antes de realizar cualquier transición.
- Z_0 : El símbolo inicial. Inicialmente, la pila del autómata de pila consta de este símbolo y de nada más.
- F : El conjunto de estados de aceptación o estados finales.

La interpretación intuitiva de $\delta(q_0, a, Z_0) = \{(q_1, \gamma_1), (q_2, \gamma_2), \dots, (q_n, \gamma_n)\}$, con $q_i \in \mathcal{Q}$, $a \in (\Sigma \cup \{\epsilon\})$, $\gamma_i \in \Gamma^*$ es la siguiente:

Cuando el estado del autómata es q_0 , 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:

1. Si $a \in \Sigma$, es decir no es la palabra vacía, se avanza una posición la cabeza lectora para inspeccionar el siguiente símbolo.
 2. Se elimina el símbolo Z de la pila del autómata.
 3. Se selecciona un par (q_i, γ_i) de entre los existentes en la definición de $\delta(q_0, A, Z)$, la función de transición del autómata.
 4. 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.
 5. Se cambia el control del autómata al estado q_i .
- Ejemplo Sea el siguiente lenguaje independiente de 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 de 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:

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

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 ϵ 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 del alfabeto de entrada Σ , estos pasan a la pila en forma

de símbolos del alfabeto de pila. 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:

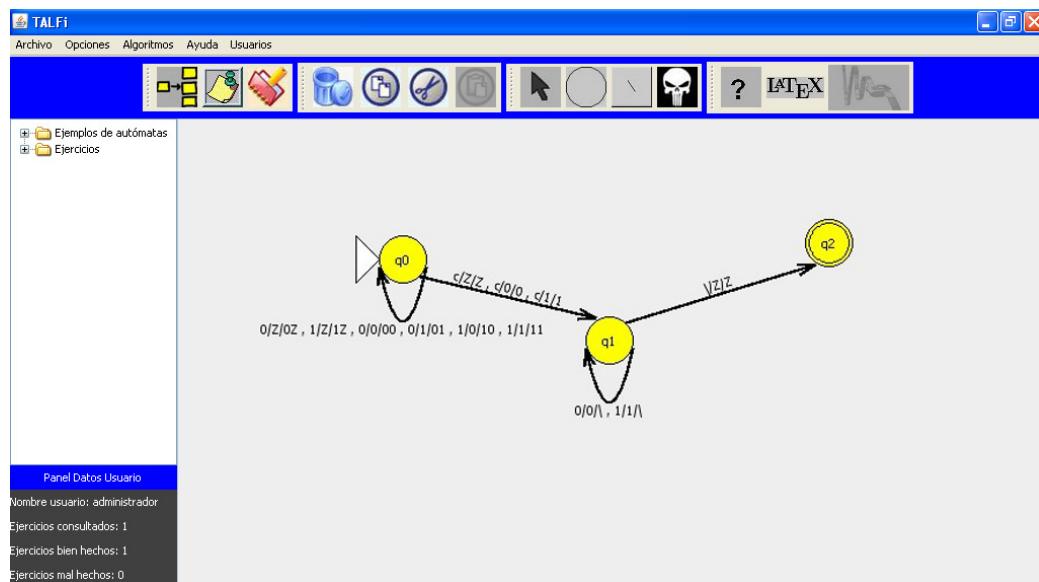
Aunque a los autómatas de pila se les permite, por definición, ser no deterministas, el subcaso determinista es bastante importante. En particular, los analizadores sintácticos se comportan generalmente como autómatas de pila deterministas, por lo que el conjunto de lenguajes que pueden ser aceptados por estos autómatas es interesante, por las ideas que nos da sobre qué construcciones es adecuado usar en los lenguajes de programación. En esta sección definiremos los autómatas de pila deterministas e investigaremos algunas de las cosas que pueden hacer y no hacer.

Definición de un autómata de pila determinista:

Intuitivamente, un autómata de pila es determinista si en ninguna situación puede elegir entre dos o más movimientos alternativos. Estas alternativas son de dos tipos. Si $\delta(q,a,X)$ contiene más de un par, el autómata de pila es no determinista, porque se puede elegir entre estos pares al decidir el siguiente movimiento. Sin embargo, incluso aunque $\delta(q,a,X)$ tenga siempre un único elemento, aun sería posible elegir entre usar un símbolo de entrada o hacer un movimiento sobre ϵ . Decimos que un autómata de pila $P = (\mathcal{Q}, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ es determinista (y lo llamamos APD, iniciales de autómata de pila determinista), si y solo si se cumplen las siguientes condiciones:

1. $\delta(q,a,X)$ tiene como máximo un elemento para cualquier q en \mathcal{Q} , a en Σ o $a = \epsilon$, y X en Γ .
2. Si $\delta(q,a,X)$ no está vacío para algún a en Σ , $\delta(q,\epsilon,X)$ debe estar vacío.

Ejemplo de un autómata de pila determinista en TALFi:



3.4.2. Gramática independiente de contexto

Una gramática independiente de contexto es una notación formal que sirve para expresar las definiciones recursivas de los lenguajes. Una gramática consta de una o más variables que representan las clases de cadenas, es decir, los lenguajes. Existen reglas que establecen cómo se construyen las cadenas de cada clase. La construcción puede emplear símbolos del alfabeto, cadenas que se sabe que pertenecen a una de las clases, o ambos elementos.

- Definición formal:

Existen cuatro componentes importantes en una descripción grammatical de un lenguaje:

1. Un conjunto finito de símbolos que forma las cadenas del lenguaje que se está definiendo. Denominamos a este conjunto alfabeto terminal o alfabeto de símbolos terminales.
2. Un conjunto finito de variables, denominado también en ocasiones símbolos no terminales o categorías sintácticas. Cada variable representa un lenguaje; es decir, un conjunto de cadenas.

3. Una de las variables representa el lenguaje que se está definiendo; se denomina símbolo inicial. Otras variables representan las clases auxiliares de cadenas que se emplean para definir el lenguaje del símbolo inicial.
4. Un conjunto finito de producciones o reglas que representan la definición recursiva de un lenguaje. Cada producción consta de:
 - Una variable a la que define (parcialmente) la producción. Esta variable a menudo se denomina cabeza de producción.
 - El símbolo de producción \rightarrow .
 - Una cadena formada por cero o más símbolos terminales y variables. Esta cadena, denominada cuerpo de la producción, representa una manera de formar cadenas pertenecientes al lenguaje de la variable de la cabeza. De este modo, dejamos los símbolos terminales invariables y sustituimos cada una de las variables del cuerpo por una cadena que sabemos que pertenece al lenguaje de dicha variable.

Los cuatro componentes que acabamos de describir definen una gramática independiente de contexto (GIC), o simplemente una gramática. Representamos una gramática independiente de contexto G mediante sus cuatro componentes, es decir, $G=(V,T,P,S)$, donde V es el conjunto de variables, T son los símbolos terminales, P es el conjunto de producciones y S es el símbolo inicial.

■ Ejemplos:

1. Una simple gramática independiente de contexto es

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

donde $|$ es un “o” lógico y es usado para separar múltiples opciones para el mismo símbolo no terminal, ϵ indica una cadena vacía.

Esta gramática genera el lenguaje no regular $\{a^n b^n : n \geq 0\}$.

2. Aquí hay una gramática independiente de contexto para expresiones enteras algebraicas sintácticamente correctas sobre las variables algebraicas, que son símbolos terminales en la gramática, x, 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 independiente 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 independiente de contexto.

$$\begin{aligned} S &\rightarrow aSc \mid B \\ B &\rightarrow bBc \mid E \end{aligned}$$

■ Formas normales:

- Forma normal de Greibach:

Una gramática independiente de 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:

$$\begin{aligned} A &\rightarrow BC \text{ ó} \\ A &\rightarrow a \end{aligned}$$

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

3.4.3. 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.

Los lenguajes de programación son lenguajes independientes del contexto. Existe una forma mecánica de convertir la descripción de un lenguaje como GIC en un analizador sintáctico: el componente del compilador descubre la estructura del programa fuente y representa dicha estructura mediante un árbol de derivación.

Otro ejemplo es el lenguaje HTML, o también el lenguaje XML, cuya parte fundamental es la DTD (Document Type Definition, definición de tipo de

documento), que principalmente es una gramática independiente de contexto que describe las etiquetas permitidas y las formas en que dichas etiquetas pueden anidarse.

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.

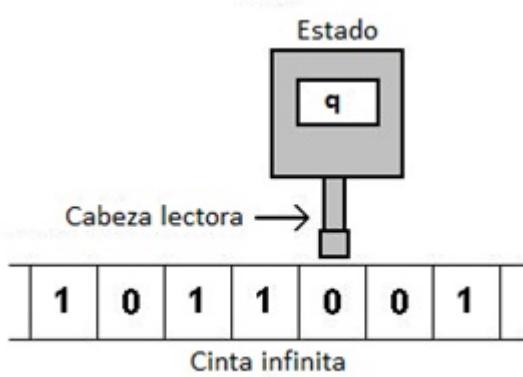
La idea subyacente es el concepto de que una máquina de Turing es una persona ejecutando un procedimiento efectivo definido formalmente, donde el espacio de memoria de trabajo es ilimitado, pero en un momento determinado sólo una parte finita es accesible. La memoria se divide en espacios de trabajo 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.

- Nociones preliminares:

Podemos imaginarnos una máquina de Turing de la manera siguiente: la máquina consta de una unidad de control, que puede estar en cualquier estado tomado de un conjunto finito. Hay una cinta dividida en casillas, que puede contener un símbolo, tomado de otro conjunto finito.



Inicialmente, se sitúa en la cinta de entrada, que es una cadena de símbolos de longitud finita, elegidos del alfabeto de entrada. El resto de las casillas de la cinta, que se extiende infinitamente hacia la derecha y la izquierda, contiene, inicialmente, un símbolo denominado espacio en blanco.

El espacio en blanco es un símbolo de cinta, pero no un símbolo de entrada, y puede haber también otros símbolos de cinta además de los símbolos de entrada y del espacio en blanco.

Existe una cabeza de cinta que siempre está situada sobre una de las casillas de la cinta. Se dice que la máquina de Turing está señalando dicha casilla. Al principio, la cabeza de la cinta se encuentra en la casilla de entrada que se encuentra más a la izquierda y no es un blanco.

Un movimiento de la máquina de Turing es una función del estado de la unidad de control, del símbolo de la cinta y de la dirección. En un movimiento, la máquina de Turing:

1. Cambiará de estado. Opcionalmente, el siguiente estado puede ser el mismo que el actual.
2. Escribirá un símbolo de cinta en la casilla señalada por la cabeza. Este símbolo de cinta sustituye al símbolo que estuviera anteriormente en la casilla. Opcionalmente, el símbolo escrito puede ser el mismo que había en dicha casilla.
3. Moverá la cabeza de la cinta hacia la izquierda o hacia la derecha. En nuestro formalismo, no se exige que haya un movimiento, y se permite que la cabeza permanezca en el mismo lugar.

■ Definición formal de una máquina de Turing:

La notación formal que utilizaremos para una máquina de Turing es similar a la utilizada para los autómatas finitos o para los autómatas de pila.

$M = (\mathcal{Q}, \Sigma, \Gamma, \delta, q_0, \#, F)$ cuyas componentes tienen el siguiente significado:

- \mathcal{Q} : El conjunto finito de estados de la unidad de control.
- Σ : El conjunto finito de símbolos de entrada.
- Γ : El conjunto completo de símbolos de cinta; Σ siempre es un subconjunto de Γ .
- δ : La función de transición. Los argumentos de $\delta(q, X)$ son un estado q y un símbolo de cinta X . El valor de $\delta(q, X)$, si está definido, es una tupla (p, Y, S) , donde:
 1. p es el estado siguiente de \mathcal{Q} .
 2. Y es el símbolo de Γ , que se escribe en la casilla señalada por la cabeza de la cinta y que sustituye al símbolo que se encontraba en dicha casilla.
 3. S es un sentido, I, D o N (“izquierda”, “derecha” o “ninguno” respectivamente), que nos indica el sentido en el que se mueve la cabeza.
- q_0 : El estado inicial (uno de los elementos de \mathcal{Q}) en el cual se encuentra inicialmente la unidad de control.
- $\#$: El símbolo de espacio en blanco. Este símbolo forma parte de Γ pero no de Σ ; es decir, no es un símbolo de entrada. El espacio en blanco aparece inicialmente en todas las casillas de la cinta, excepto en aquéllas que contienen los símbolos de la entrada.
- F : El conjunto de estados finales o de aceptación, que constituye un subconjunto de \mathcal{Q} .

■ 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 $\{q_1, q_2, q_3, q_4, q_5\}$ y el estado inicial es q_1 .

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

Estado	Símbolo leído	Símbolo escrito	Mov.	Estado sig.
q_1	1	0	D	q_2
q_1	1	0	D	q_2
q_2	1	1	D	q_2
q_2	0	0	D	q_3
q_3	0	1	I	q_4
q_3	1	1	D	q_3
q_4	1	1	I	q_4
q_4	0	0	I	q_5
q_5	1	1	I	q_5
q_5	0	1	D	q_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	q_1	11
2	q_2	01
3	q_2	010
4	q_3	0100
5	q_4	0101
6	q_5	0101
7	q_5	0101
8	q_1	1101
9	q_2	1001
10	q_3	1001
11	q_3	10010
12	q_4	10011
13	q_4	10011
14	q_5	10011
15	q_1	11011
Parada		

La máquina realiza su proceso por medio de un bucle, en el estado inicial q1, reemplaza el primer 1 con un 0, y pasa al estado q2, con el que avanza hacia la derecha, saltando los símbolos 1 hasta un 0 (que debe existir), cuando lo encuentra pasa al estado q3, 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 q4 vuelve a la izquierda saltando los 1, cuando encuentra un 0 (en el medio de la secuencia), pasa a q5 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 q1 de nuevo. Si es un símbolo 0, será el símbolo central, con lo que la máquina se detiene al haber finalizado su cómputo.

3.4.5. Algoritmo CYK

El algoritmo de Cocke-Younger-Kasami (CYK) determina si una cadena puede ser generada por una gramática independiente de contexto y, si es posible, cómo puede ser generada. Este proceso es conocido como análisis sintáctico de la cadena.

La versión estándar CYK reconoce lenguajes definidos por una gramática independiente de contexto escrita en la forma normal de Chomsky. Cualquier gramática independiente de contexto puede ser convertida a FNC sin mucha dificultad, CYK puede usarse para reconocer cualquier palabra de lenguaje independiente de contexto.

Consiste en crear una tabla, de tamaño como la cadena a reconocer, siempre que la cadena sea distinta de ϵ , pues en ese caso solo basta comprobar si es una producción de la variable inicial. Cada celda la rellenamos con la lista de variables que nos pueden permitir crear las subcadenas basadas en la cadena dada, hasta que llegamos a la celda de la primera columna y la fila de la longitud de la cadena. Si ésta contiene a la variable inicial, es que a partir de ella podríamos derivar la cadena original, y por tanto sería reconocida por la gramática, y en consecuencia por el autómata de pila.

En el peor caso asintótico la complejidad temporal de CYK es cúbica sobre la longitud de la cadena analizada. Esto hace a éste algoritmo uno de los más eficientes en el reconocimiento de cadenas de los lenguajes independientes de contexto.

Podemos ver una codificación simplificada en pseudocódigo del algoritmo:

- Entrada : $G=(V, T, P, S)$ en FNC y $w = w_1 w_2 \dots w_n$ ($w \neq \epsilon$)
- Salida : Cierto (si $w \in L(G)$) o Falso (si $w \notin L(G)$)
- Método :

Para $i=1$ hasta n

$$V_{i1} = \{ A : A \rightarrow w_i \in P \}$$

finPara

Para $j=2$ hasta n

Para $i=1$ hasta $n-j+1$

$$V_{ij} = \emptyset$$

Para $k=1$ hasta $j-1$

$$V_{ij} = V_{ij} \cup \{ A : A \rightarrow BC \in P, B \in V_{ik}, C \in V_{i+k}, j-k \}$$

finPara

finPara

Si $S \in V_{1n}$ devolver Cierto

sino devolver Falso

3.5. Lenguaje generado por un autómata de pila

Esta parte fue la que nos llevó más tiempo, pues como hemos explicado antes, seguimos una línea de trabajo que desafortunadamente nos llevó 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. Pasos de simplificación antes y después de realizar transformaciones FNC o FNG

Estamos acostumbrados en los ejemplos a ver gramáticas con pocos símbolos de variable, y en general bastante simplificadas. Al aplicar el algoritmo que nos da la gramática independiente de contexto asociada, vimos que aún teniendo pocos estados, en el momento que el número de aristas comienza a crecer, obtenemos unas gramáticas bastante grandes, y nos vimos en la obligación de reducirlas adaptando las reglas de simplificación de una gramática:

- Eliminación de variables sin producciones.
- Sustitución de una variable por su producción si sólo tiene asociada a ella una única producción y es ϵ .
- Sustitución de una variable por su producción si sólo tiene asociada a ella una única producción.
- Eliminación de producciones para variables inalcanzables.
- Eliminación de producciones ϵ que cuya cabeza no sea la variable inicial.
- Renombramiento de variables.

Las tres primeras normas se aplican antes de comentar la transformación en forma normal de una gramática. Nos permiten reducir y dejar mucho más claro el lenguaje que definen. Las dos siguientes las aplicamos para simplificar el resultado final, pues las formas normales tienen unas características estrictas, por tanto en algunos casos no hará ni falta aplicarlas, y en otros es necesario porque no se aplicaron al principio.

A continuación mostramos algunos de estos resultados que aparecen en el archivo HTML que se genera con la simplificación de gramáticas independientes de contexto.

Quitamos variables sin producciones:

```
[[S0XS0], [S1XS0], [auxXS0], [S0XS1], [S1XS1],
[auxXS1], [S0Xiniaux], [S1Xiniaux], [auxXiniaux],
[S0Xaux], [S1ZS0], [iniauxZS0], [auxZS0],
[S1ZS1], [iniauxZS1], [auxZS1], [S1Ziniaux],
[iniauxZiniaux], [auxZiniaux], [iniauxZaux],
[S1AS0], [iniauxAS0], [auxAS0], [S1AS1],
[iniauxAS1], [auxAS1], [S1Ainiaux],
[iniauxAiniaux], [auxAiniaux], [iniauxAaux],
[S1BS0], [iniauxBS0], [auxBS0], [S1BS1],
[iniauxBS1], [auxBS1], [S1Biniaux],
[iniauxBiniaux], [auxBiniaux], [iniauxBaux]]
```

Quitamos variables con una única producción:

```
[[iniauxXS0], [iniauxXS1], [iniauxXiniaux],
[S0AS1], [S0Ainiaux], [S0BS1], [S0Biniaux],
[S1Xaux], [S1Baux], [S1Aaux], [S1Zaux],
[auxXaux], [auxBaux], [auxAaux], [auxZaux]]
```

Quitamos variables no alcanzables:

```
[[S0ZS0]]
```

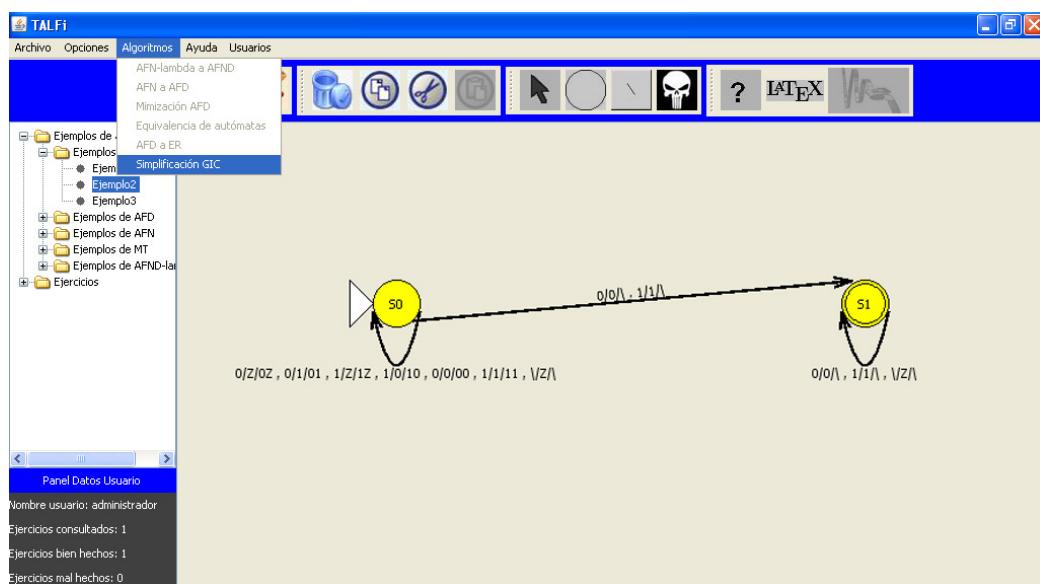
Puede verse una breve descripción del paso de simplificación a realizar, y sobre qué variables se hace, y a continuación mostramos la gramática resultante. Así hasta renombrar las variables para que la gramática quede a la forma en la que estamos habituados a tratar.

Gramática de entrada simplificada totalmente

Variables: [S, A, B, C, D, E, F, G, H]
Variable Inicial: S
Símbolos Terminales: [a, b]
Producciones: {D=[a,E,D, a,F, b,G,D, b,H], E=[a,E,E, b], F=[a,E,F, a,F], G=[b,G,G, a], A=[C,A, a,E,B, b,G,B, \, a,E,D, a,F, b,G,D, b,H], B=[a,E,B, b,G,B, \,], S=[A], C=[a,E,C, b,G,C], H=[b,G,H, b,H]}

3.5.2. Algoritmo de paso de AP \rightarrow gramática

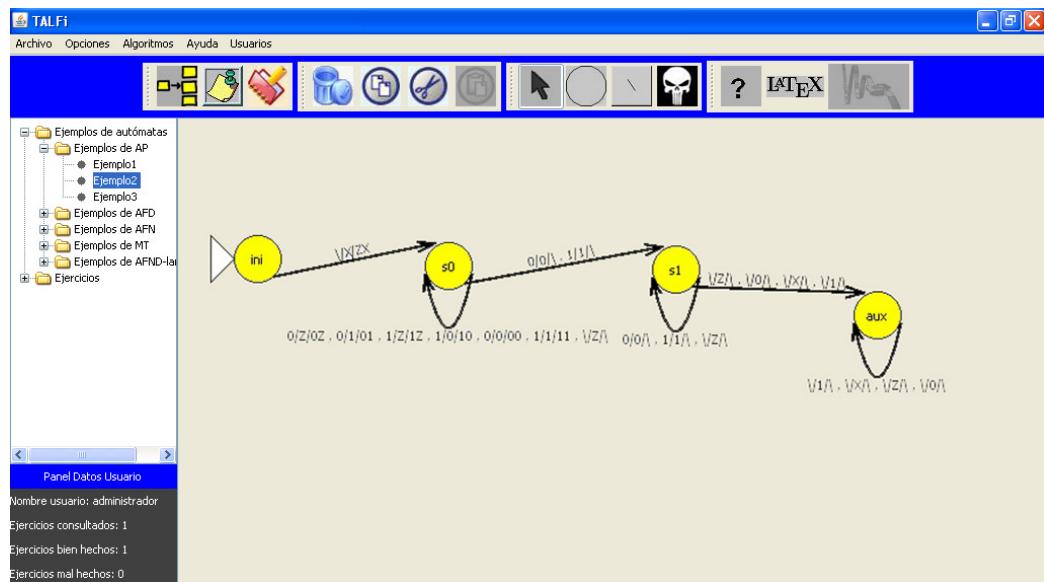
Encontramos muchas dificultades, pues este algoritmo viene detallado muy formalmente y los ejemplos que pudimos encontrar no eran demasiado aclaratorios. Para realizar esta nueva función, hemos incluido en el menú de algoritmos el botón “Simplificación GIC”:



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.



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 p del autómata, siendo q_0 el estado inicial tendremos la producción
 $S \rightarrow [q_0 Z p]$
- 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 ϵ .
- 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 ϵ .

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.

3.5.3. Algoritmo de simplificación gramática independiente de contexto a FNG

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 comienzan 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										
Variables: [S, A, B, C, D, E, F, G, H, I, J]										
Variable Inicial: S										
Símbolos Terminales: [0, 1]										
Producciones: {D=[0,H,D, 0,I, 0,J, 1,E,D, 1,F, 1,G], E=[0,H,E, 1,E,E], F=[0,H,F, 0,I,1, 1,E,F, 1,F,1, 1], G=[0,H,G, 0,I, 0,J, 1,E,G, 1,F, 1,G], A=[C,A, 0,H,B, 0,I, 1,E,B, 1,F, 0,H,D, 0,J, 1,E,D, 1,G], B=[0,H,B, 0,I, 1,E,B, 1,F], S=[A], C=[0,H,C, 1,E,C], H=[1,E,H, 0,H,H], I=[1,E,I, 1,F,0, 0,H,I, 0,I,0, 0], J=[1,E,J, 1,F, 1,G, 0,H,J, 0,I, 0,J]}										

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

-	S	A	B	C	D	E	F	G	H	I	J
S	-	X	-	-	-	-	-	-	-	-	-
A	-	-	-	X	-	-	-	-	-	-	-
B	-	-	-	-	X	-	-	-	-	-	-
C	-	-	-	-	-	X	-	-	-	-	-
D	-	-	-	-	-	-	X	-	-	-	-
E	-	-	-	-	-	-	-	X	-	-	-
F	-	-	-	-	-	-	-	-	X	-	-
G	-	-	-	-	-	-	-	-	-	X	-
H	-	-	-	-	-	-	-	-	-	-	X
I	-	-	-	-	-	-	-	-	-	-	-
J	-	-	-	-	-	-	-	-	-	-	-

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:

Gramática										
Variables: [S, A, B, C, D, E, F, G, H, I, J]										
Variable Inicial: S										
Símbolos Terminales: [0, 1]										
Producciones: {D=[0,H,D, 0,I, 0,J, 1,E,D, 1,F, 1,G], E=[0,H,E, 1,E,E], F=[0,H,F, 0,I,1, 1,E,F, 1,F,1, 1], G=[0,H,G, 0,I, 0,J, 1,E,G, 1,F, 1,G], A=[C,A, 0,H,B, 0,I, 1,E,B, 1,F, 0,H,D, 0,J, 1,E,D, 1,G], B=[0,H,B, 0,I, 1,E,B, 1,F], S=[C,A, 0,H,B, 0,I, 1,E,B, 1,F, 0,H,D, 0,J, 1,E,D, 1,G], C=[0,H,C, 1,E,C], H=[1,E,H, 0,H,H], I=[1,E,I, 1,F,0, 0,H,I, 0,I,0, 0], J=[1,E,J, 1,F, 1,G, 0,H,J, 0,I, 0,J]}										

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.

-	S	A	B	C	D	E	F	G	H	I	J
S	-	-	-	-	-	-	-	-	-	-	-
A	-	-	-	-	-	-	-	-	-	-	-
B	-	-	-	-	-	-	-	-	-	-	-
C	-	-	-	-	-	-	-	-	-	-	-
D	-	-	-	-	-	-	-	-	-	-	-
E	-	-	-	-	-	-	-	-	-	-	-
F	-	-	-	-	-	-	-	-	-	-	-
G	-	-	-	-	-	-	-	-	-	-	-
H	-	-	-	-	-	-	-	-	-	-	-
I	-	-	-	-	-	-	-	-	-	-	-
J	-	-	-	-	-	-	-	-	-	-	-

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

Gramática final simplificada
Variables: [S, A, B, C, D, E, F, G, H, I, J]
Variable Inicial: S
Símbolos Terminales: [0, 1]
Producciones: {D=[0,H,D, 0,I, 0,J, 1,E,D, 1,F, 1,G], E=[0,H,E, 1,E,E], F=[0,H,F, 0,I,1, 1,E,F, 1,F,1, 1], G=[0,H,G, 0,I, 0,J, 1,E,G, 1,F, 1,G], A=[0,H,C,A, 1,E,C,A, 0,H,B, 0,I, 1,E,B, 1,F, 0,H,D, 0,J, 1,E,D, 1,G], B=[0,H,B, 0,I, 1,E,B, 1,F], S=[0,H,C,A, 1,E,C,A, 0,H,B, 0,I, 1,E,B, 1,F, 0,H,D, 0,J, 1,E,D, 1,G], C=[0,H,C, 1,E,C], H=[1,E,H, 0,H,H], I=[1,E,I, 1,F,0, 0,H,I, 0,I,0, 0], J=[1,E,J, 1,F, 1,G, 0,H,J, 0,I, 0,J]}

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.
3. Nos deshacemos de variables que no fueran alcanzables desde el símbolo inicial, repitiendo el proceso para comprobar si su eliminación ha provocado algún cambio en la gramática.

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 ϵ . 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.

3.5.4. Algoritmo de simplificación gramática independiente de contexto a FNC

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 ϵ 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: $A \rightarrow A_1A_2A_3 \dots A_n$. Creamos una nueva variable, $[D0]$, que contendrá la producción $A_2A_3 \dots A_n$, y modificamos la producción que acabamos de encontrar suplantándola por $A \rightarrow A_1[D0]$.

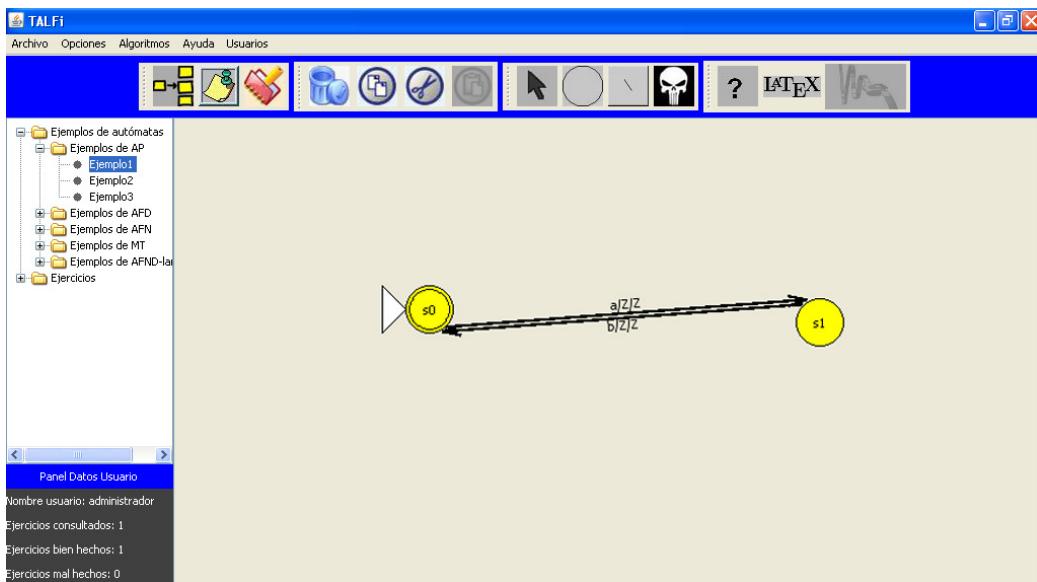
Esta fase 2 se repite hasta que todas las producciones tengan longitud menor que 3, o lo que es lo mismo, que ninguna cumpla la condición para ser modificada.

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:



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

*****FASE 1*****

Gramatica

Variables: [S, A, [Ca], [Cb]]

Variable Inicial: S

Símbolos Terminales: [a, b]

Producciones: {[Ca]=[a], A=[[Ca],[Cb],A, \], S=[A], [Cb]=[b]}

- La salida obtenida de terminar con la segunda fase:

*****FASE 2*****

Gramatica

Variables: [S, A, [Ca], [Cb], [D0]]

Variable Inicial: S

Símbolos Terminales: [a, b]

Producciones: {[Ca]=[a], A=[[Ca],[D0], \], S=[A], [D0]=[[Cb],A], [Cb]=[b]}

- Observemos la producción $S \rightarrow A$ y la producción de [D0] cómo cambia al terminar con la simplificación y conseguir una gramática en FNC:

Gramática final simplificada

Variables: [S, A, [Ca], [Cb], [D0]]

Variable Inicial: S

Símbolos Terminales: [a, b]

Producciones: {[Ca]=[a], A=[[Ca],[D0]],
S=[[Ca],[D0], \], [D0]=[[Cb],A, b], [Cb]=[b]}

3.5.5. 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.

3.5.6. Palabras reconocidas por un autómata de pila

Ya explicamos al comienzo cómo fue nuestro primer intento de verificar cuándo una cadena es reconocida por un autómata de pila, y cómo al final optamos por utilizar el algoritmo CYK.

La salida de la tabla que se genera se puede ver por consola únicamente pues consideramos que a el usuario no le importa cómo se obtiene el resultado, sino el valor del mismo.

Para cada palabra que se va a comprobar en la corrección de ejercicios de autómata de pila, se guarda el resultado del CYK, por si en un futuro quiere devolverse esta información para ampliar la funcionalidad de TALFi. Nosotros mismos pensamos que sería interesante incluir algún aparato en el menú con esta función, pero finalmente consideramos que la generación aleatoria de palabras era suficiente para comprender las cadenas aceptadas por un autómata de pila.

3.6. Máquinas de Turing

3.6.1. Diferencias entre máquinas de Turing con y sin estados finales

Hemos decidido un tratamiento diferente en las máquinas de Turing que contienen estados finales de las que no, al responder a la cuestión de qué información devolver cuando el cómputo para una determinada entrada de cinta es terminante.

Para aquellas máquinas de Turing que no contienen estados finales, devolvemos cómo queda la cinta al finalizar el cómputo, pues asumimos que ésta información es importante, ya que la simulación no se detiene en un estado final, también llamado de aceptación. El usuario debe comprobar si su contenido es el esperado o no.

En cambio, si existen estados finales, consideramos que el estado en el que quede la cinta no es relevante, y únicamente mostramos un mensaje al usuario

informándole de que su cinta de entrada se ha procesado con éxito si ha finalizado la simulación en un estado final, o que ha fallado su reconocimiento en caso contrario.

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 cálculo, es decir, cicle o no sea terminante. Este problema no es decidable, 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 de complemento para que se detectase el caso de ciclo en el cálculo 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. Cambios destacables en TALFi 2.0

3.7.1. 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.

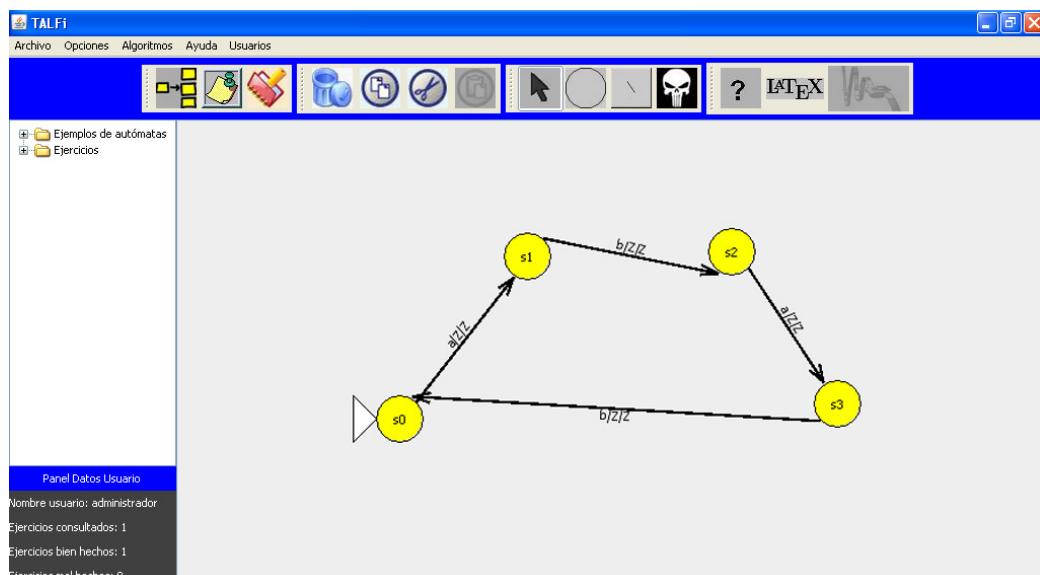
Notación gráfica para automátas de pila

Un diagrama que generaliza el diagrama de transición de un autómata finito aclara determinados aspectos del comportamiento de un autómata de pila. Por lo tanto, introduciremos y usaremos a partir de ahora un diagrama de transición para autómatas de pila en el que:

1. Los nodos corresponden a los estados del autómata de pila.
2. Una triángulo blanco indica el estado inicial, y los estados con un doble círculo son de aceptación, igual que en la primera versión de TALFi cuando se dibujan los autómatas finitos.

3. Los arcos corresponden a las transiciones del autómata de pila en el sentido siguiente:

un arco con la etiqueta $a/X/\alpha$ desde el estado q al estado p significa que $\delta(q,a,X)$ contiene el par (p,α) , quizás entre otros pares. Es decir, la etiqueta del arco dice qué entrada se usa y también indica los elementos de lo alto de la pila, antiguos y nuevos. La única cosa que el diagrama no nos dice es qué símbolo de la pila es el símbolo inicial. Por convenio, será Z_0 , a menos que se tenga que transformar un autómata de pila que acepte un lenguaje por estado final en otro que acepte el mismo lenguaje por pila vacía, siendo en ese caso X_0 .

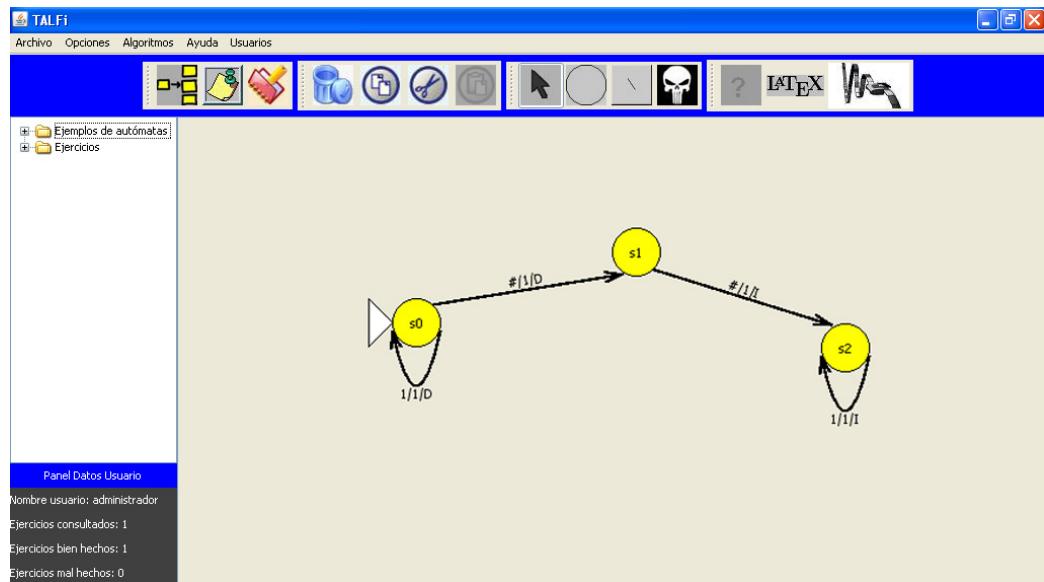


Notación gráfica para máquinas de Turing

Las transiciones de una máquina de Turing pueden representarse visualmente de una forma muy parecida como se hace para los autómatas de pila. Un diagrama de transición está formado por un conjunto de nodos que corresponden a los estados de la máquina de Turing. En un arco que vaya del estado q al estado p , aparecerán una o varias etiquetas de la forma $X/Y/S$, donde X e Y son símbolos de cinta, y S indica un sentido, que puede ser I, D o N. Es decir, si $\delta(q,X) = (p,Y,S)$, en el arco que va de q a p se encontrará la etiqueta $X/Y/S$. Como en el resto de diagramas de transición, el estado inicial se indica mediante la colocación de un triángulo blanco a su izquierda, y los estados de aceptación se indican mediante círculos dobles.

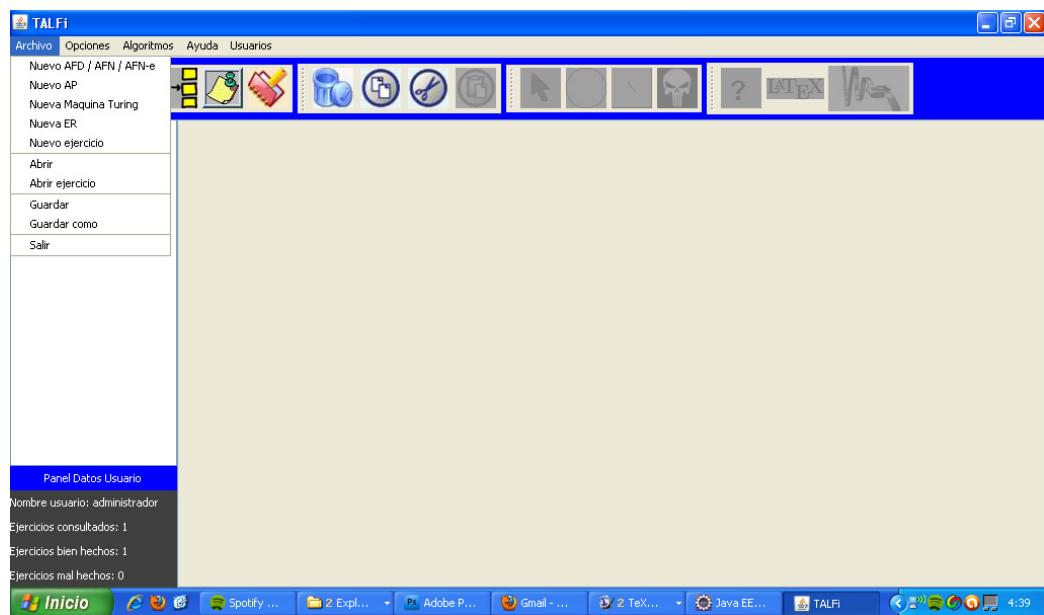
Por tanto, la única información sobre la máquina de Turing que habría que añadir al diagrama es el símbolo que se utiliza para denotar el espacio

en blanco. Dicho símbolo es #.



Crear un nuevo autómata 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 autómata 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:



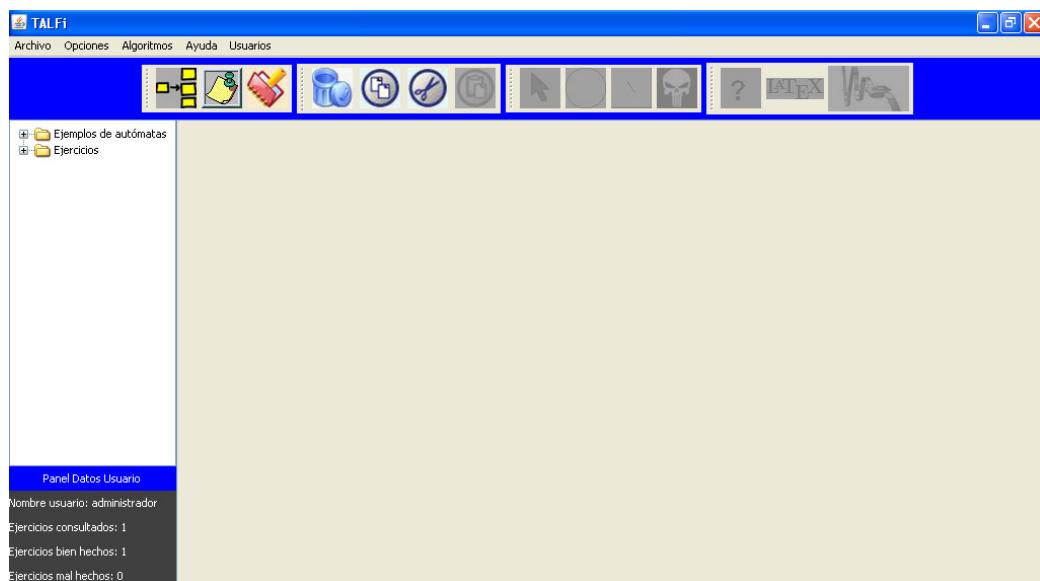
Botones

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 L^AT_EX del autómata que visualizamos en el canvas.

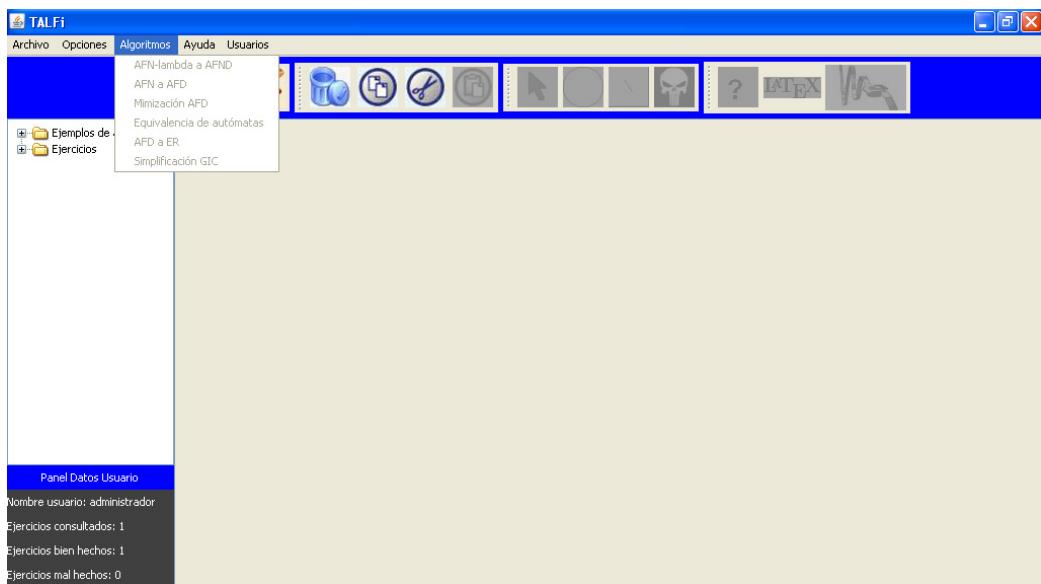
Al comienzo de la aplicación aparecen los tres desactivados, y según las áreas que trabajemos se irán activando si corresponde.

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

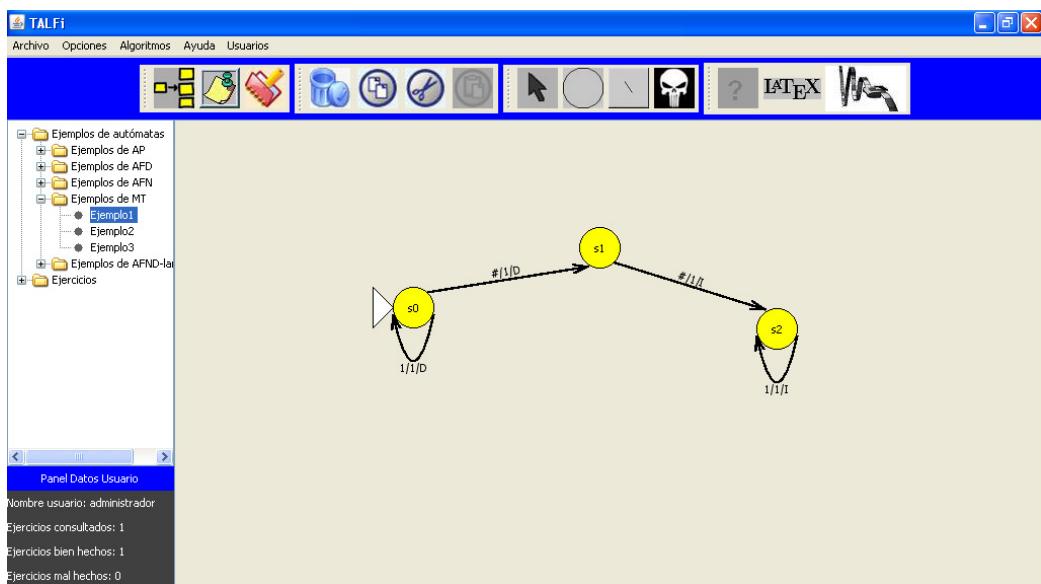
Aspecto de la aplicación al arrancarla:



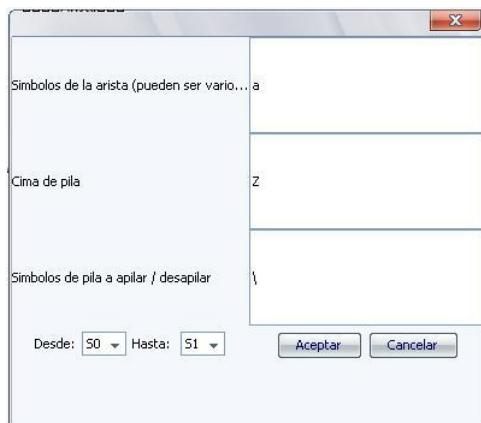
Menú Algoritmos al iniciar la aplicación:



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

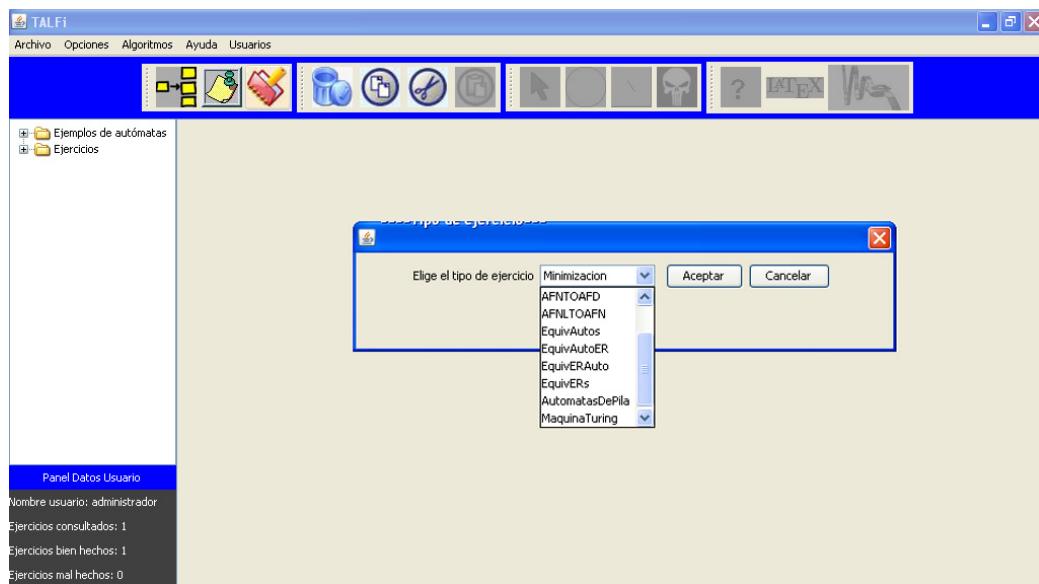


Cuadro de diálogo que aparece al modificar una arista de un autómata de pila:



3.7.2. Ejercicios

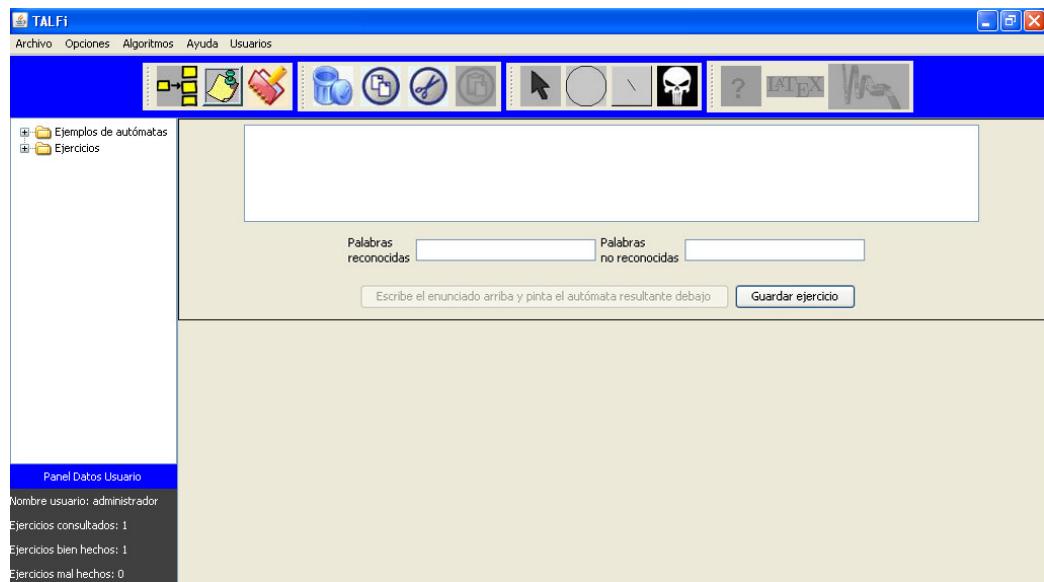
Ahora también es posible crear ejercicios de autómatas de pila y máquinas de Turing. Hemos incluido una pequeña colección, pero el administrador puede ampliarla creando sus propios ejercicios y añadiéndolos a la base de datos de los mismos. A continuación detallaremos como crearlos, que información interna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



Hasta el momento, el administrador solamente tenía que dibujar el autómata o escribir 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. A continuación describimos los cambios introducidos.

Interfaz gráfica para ejercicios de autómatas de pila

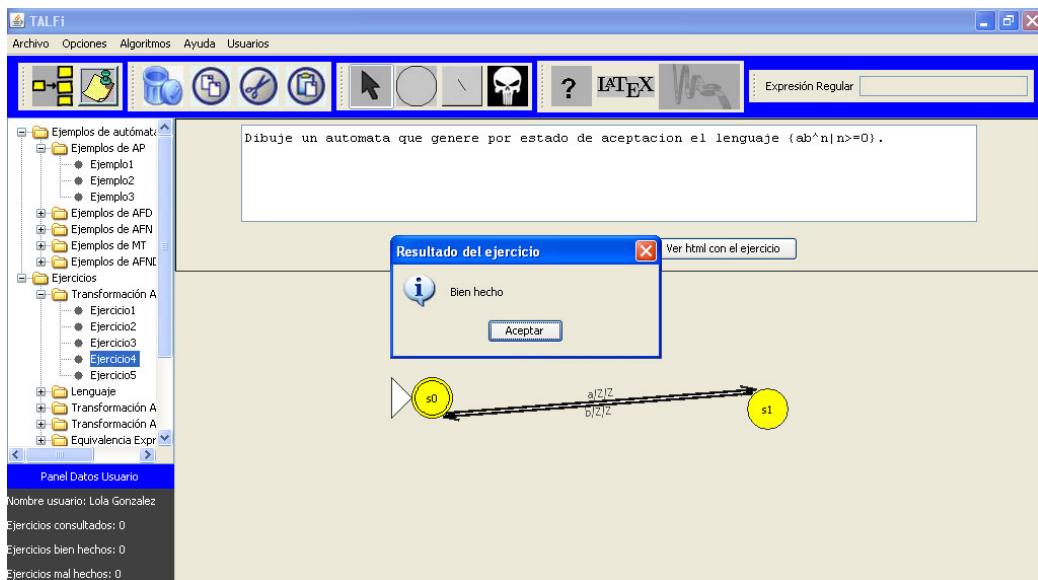
Un lenguaje independiente de contexto puede generarse con distintas gramáticas independientes de contexto, y por tanto con diferentes autómatas de pila. Por este motivo, necesitamos tener una lista de palabras que tendremos que comprobar que son aceptadas el autómata de pila dibujado como solución, resultado que obtenemos fácilmente mediante el resultado que obtenemos al aplicar el algoritmo CYK. Para afinar más aún el resultado, pedimos otra lista de palabras que no deben ser reconocidas por el autómata. Ninguna de las dos tiene límite, pero si es necesario que alguna de ellas contenga alguna cadena, pues con ambas vacías no podríamos comparar la solución del ejercicio con la que nos proporcione el usuario.



Al autómata que pinte el usuario se le aplicará el algoritmo de CYK, y si coinciden todos los resultados, se considerará el ejercicio como apto, en otro caso no se dará como no válido.

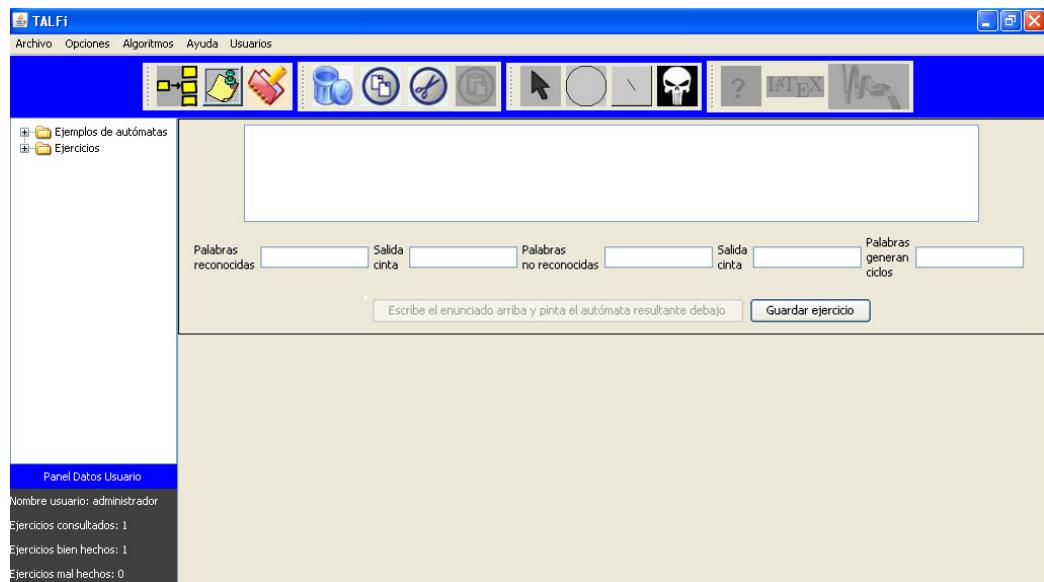
El usuario que vaya a enviar la corrección del ejercicio en ningún momento tendrá conocimiento de cuales son las palabras que pertenecen a estas listas.

Simulación de la corrección del ejercicio 4 de la colección de ejercicios de autómatas de pila de TALFi



Interfaz gráfica para ejercicios de máquinas de Turing

La corrección de ejercicios dependerá de si la solución tiene estados finales o no, puesto que en el primer caso, en los cómputos que finalizan no nos importa el contenido de la cinta, pero sin embargo en el segundo sí. Como no sabemos si se va a crear un ejercicio cuya solución tenga estados finales, incluimos cinco casillas, en las cuáles se podrán escribir las cintas de entrada que serán aceptadas, las salidas de cinta generadas por las cintas de entrada aceptadas, las cintas de entrada que no serán aceptadas, las salidas de cinta generadas por las cintas de entrada que no se aceptarán, y las cintas de entrada que provocarán ciclos en la simulación. No es necesario llenar toda esta información, pero si alguno, pues sino es imposible corregir el ejercicio.



Si la solución del ejercicio no contiene estados finales, comprobaremos que las cintas de entrada con sus cintas de salidas asociadas contienen el mismo número de elementos. Si son distintos avisamos al administrador para que las revise.

No se generarán las cintas de salida obtenidas en el caso de que pare la simulación, ya que sólo necesitamos saber si se ha obtenido la misma salida en la solución que en la simulación de la respuesta dibujada por el usuario.

Capítulo 4

Entorno LATEX

4.1. LATEX

La opción de que TALFi imprimiese cualquier autómata que se dibujase en 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 LATEX

LATEX es un sistema de composición de textos, orientado especialmente a la creación de libros, documentos científicos y técnicos que contengan fórmulas matemáticas.

LATEX está formado por un gran conjunto de macros de TEX, escrito por Leslie Lamport en 1984, con la intención de facilitar el uso del lenguaje de composición tipográfica 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 LATEX es comparable a la de una editorial científica de primera línea.

LATEX es software libre bajo licencia LPPL.

4.1.2. Descripción:

LATEX es un sistema de composición de textos que está formado mayoritariamente por órdenes (macros) construidas a partir de comandos de 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 *LATEX* utilizando comandos propios del *TeX* descritos en *The TeXbook*”. Esto es lo que convierte a *LATEX* en una herramienta práctica y útil pues, a su facilidad de uso, se une toda la potencia de *TeX*. Estas características hicieron que *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 *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 reestandardización completa de *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 *LATEX* 2.09. Esta tarea la realizaron Frank Mittelbach, 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 *LATEX2_ε* (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 reestandardización 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, *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.

4.2. Uso

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, *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 L^AT_EX, 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 L^AT_EX interpreta la “forma” que debe tener el documento es mediante etiquetas. Por ejemplo, “documentclass{article}” le dice a L^AT_EX 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 L^AT_EX 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 quieras decir”).

Una de las ventajas de L^AT_EX 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 LaTe_X 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 Mac OS 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.

4.2.1. Uso de LATEXen la herramienta TALFi

Aunque LATEXes 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 LATEX. Gracias a esta amplitud de posibilidades para crear gráficos complejos en LATEX, el trabajo inicial de poder imprimir los autómatas o las simplificaciones acabó dando como resultado 3 posibles representaciones en 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.

4.3. LatexCodeConverter

Esta clase, es la encargada de traducir cualquier autómata de TALFi en un archivo TEX totalmente listo para ser compilado en cualquier entorno para LATEX.

- Algoritmo conversor:

La implementación del algoritmo consiste básicamente en traducir los distintos campos de los que consta un autómata en comandos 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 generaremos con LATEX.

3. Generamos las aristas viendo desde que estado parten y a cual se dirigen.
4. Cerramos el archivo.

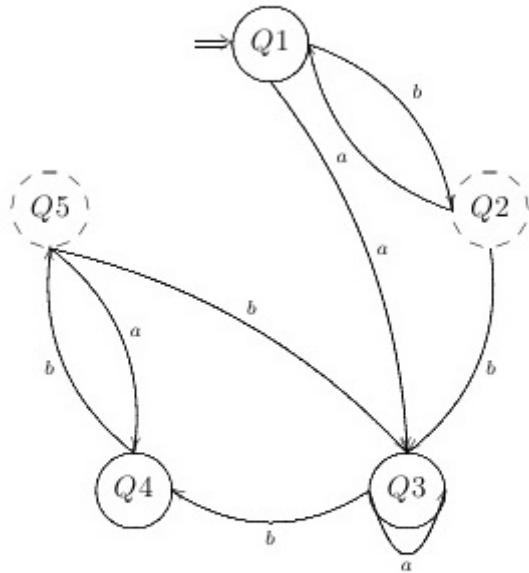
- Representaciones:

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 L^AT_EX será prácticamente idéntica.

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

\[
\begin{array}{l}
\text{\textbackslash xy} \\
(87,83)*{Q1}; \\
(116,61)*{Q2}; \\
(105,24)*{Q3}; \\
(69,24)*{Q4}; \\
(58,61)*{Q5}; \\
(87,83)*\xycircle(5.00,5.00){}=Q1"; \\
(116,61)*\xycircle(5.00,5.00){-}=Q2"; \\
(105,24)*\xycircle(5.00,5.00){}=Q3"; \\
(69,24)*\xycircle(5.00,5.00){-}=Q4"; \\
(58,61)*\xycircle(5.00,5.00){-}=Q5"; \\
\ar@{=>}(77,83)*{};(82,83)*{} \\
\ar@/^1pc/^{b}(92,83)*{};(111,61)*{} \\
\ar@/^1pc/^{a}(87,78)*{};(105,29)*{} \\
\ar@/^1pc/^{b}(116,56)*{};(105,29)*{} \\
\ar@/^1pc/^{a}(111,61)*{};(92,83)*{} \\
\ar@/^1pc/^{b}(100,24)*{};(74,24)*{} \\
\ar@/_2pc/_a(100,24)*{};(110,24)*{} \\
\ar@/^1pc/^{b}(69,29)*{};(58,56)*{} \\
\ar@/^1pc/^{b}(58,56)*{};(105,29)*{} \\
\ar@/^1pc/^{a}(58,56)*{};(69,29)*{} \\
\end{array}
\]
\end{xy}
\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 (inlcuso pordía 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 L^AT_EX de manera manual, este es el mejor método posible.

Lo comprobamos en un ejemplo:

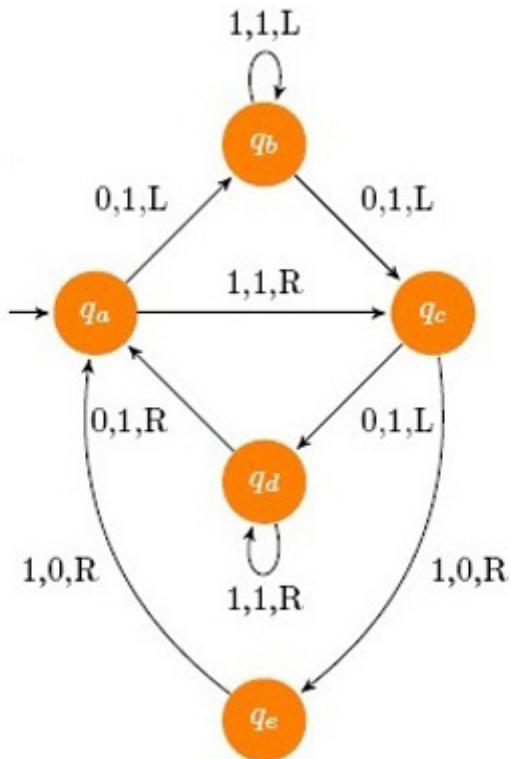
```
\documentclass{article}

\usepackage{pgf}
\usepackage{tikz}
\usetikzlibrary{arrows,automata,positioning}
\usepackage[latin1]{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} (C)
      (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:

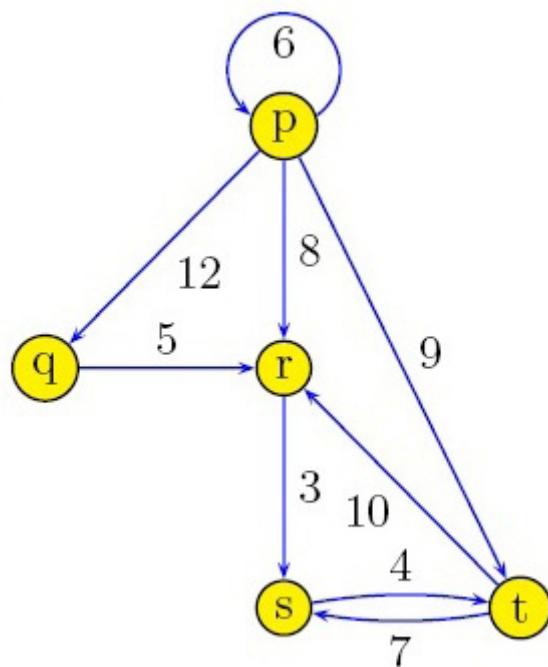
En esta implementación, no cabe posibilidad de solapamiento de estados, puesto que cada uno ocupará una “celda” de una matriz virtual que se crea en el lienzo.

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}
\nccircle{1,2}{0.5cm}^{\{6\}}
\nccline{1,2}{2,1}^{\{12\}}
\nccline{1,2}{2,2}^{\{8\}}
\nccline{1,2}{3,3}^{\{9\}}
\nccline{2,1}{2,2}^{\{5\}}
\nccline{2,2}{3,2}^{\{3\}}
\nccline{3,3}{2,2}^{\{10\}}
\ncarc[arcangle=10]{3,3}{3,2}^{\{7\}}
\ncarc[arcangle=10]{3,2}{3,3}^{\{4\}}
\end{document}
```



4.4. 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 LATEX.

Para ello usamos los comandos LATEX **tabular** y **hline** para dar un formato parecido a las tablas que se generaban en html. De esta manera podemos crear líneas verticales y horizontales para dar aspecto de tabla en 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]

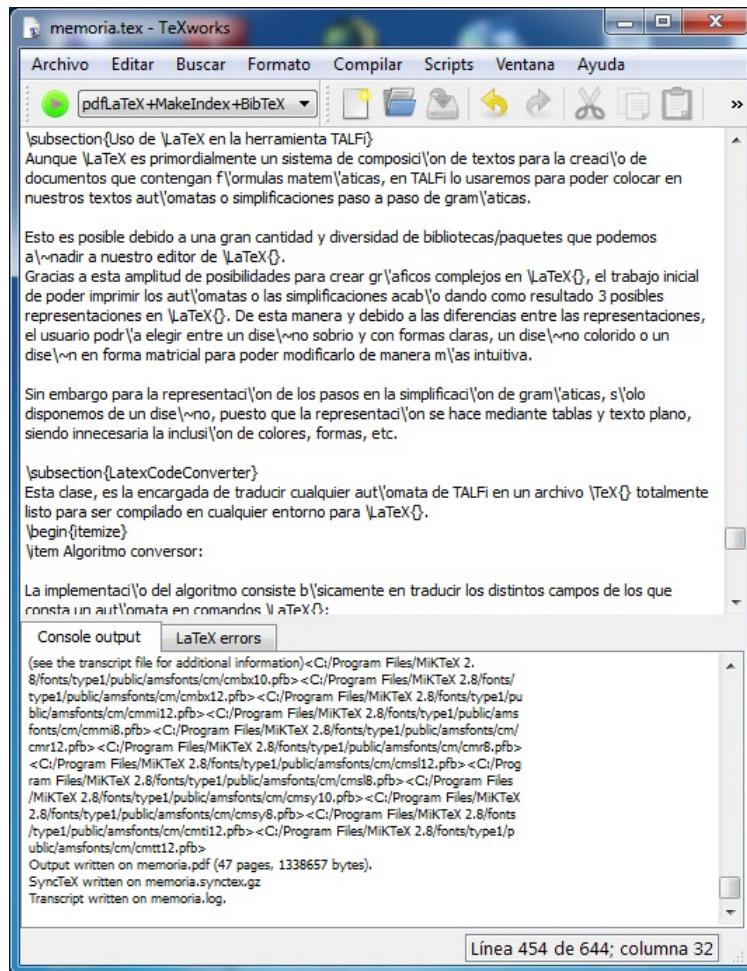
-	A
S	-
A	-
-	A
S	-
A	-

Gramática final simplificada

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

4.5. Entornos LATEX utilizados

- Miktex:



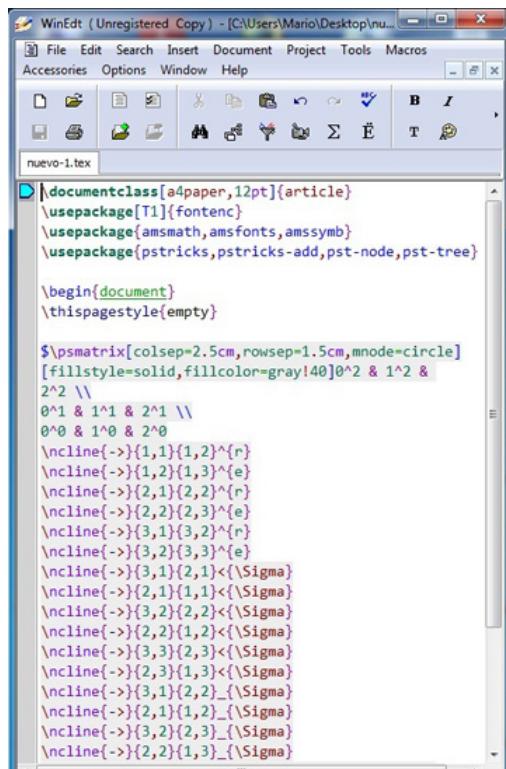
MiKTeX es una distribución TeX/LATEX para Microsoft Windows que fue desarrollada por Christian Schenk. Las características más apreciables de MiKTeX son su habilidad de actualizarse por sí mismo descargando nuevas versiones de componentes y paquetes instalados previamente, y su fácil proceso de instalación.

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 T_EX y L^AT_EX, 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 es un editor de textos de gran alcance y versatilidad para Windows, con una fuerte predisposición hacia la creación de documentos L^AT_EX. La página de descargas del sitio web tiene más información sobre WinEdt, T_EX, y enlaces a otros programas necesarios para hacer operativo WinEdt en la plataforma Windows.

Capítulo 5

Anexos

5.1. 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 ϵ en las transiciones, tendrá que hacerlo de forma distinta que en TALFi 1.0, con el carácter \.
- Por defecto el símbolo de fondo de pila es Z.
- 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”.
- Los enunciados de los ejercicios no pueden contener acentos ni el carácter ‘ñ’.
- Para especificar una cinta de blancos en los ejercicios de máquinas de Turing, tendrá que escribir #.

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.

5.2. Cambios en la implementación

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

Ampliación de paquetes

- Modelo.algoritmos:

- 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.

- C_Y_K:

Implementa el algoritmo CYK y guarda la lista de palabras que van a ejecutarse sobre él, al igual que la salida que generan, y sobre qué gramática independiente de contexto lo hacen.

- TuringResultado:

Contiene la máquina de Turing a corregir junto con la información necesaria para ello y la lista de resultados que se obtienen.

- 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.

5.3. Estructura XML de los ejercicios de autómatas de pila

Incluimos la estructura interna de los ejercicios de autómatas de pila, donde se apreciar la similitud con los ejercicios de la primera versión de TALFi y las novedades incluidas en su ampliación.

```
<ejercicio>
```

```
  <tipo>TransformacionAPs</tipo>
```

```
  <enunciado>
```

Dibuje un automata que genere por estado de aceptacion el lenguaje $\{ab^n \mid n \geq 0\}$.

```
  </enunciado>
```

5.3. ESTRUCTURA XML DE LOS EJERCICIOS DE AUTÓMATAS DE PILA73

```
<output>

<authomata>

<type>
    <item>AutomataPila</item>
</type>

<alphabet>
    <item>a</item>
    <item>b</item>
</alphabet>

<alphabetP>
    <item>Z</item>
</alphabetP>

<states>
    <state>s2</state>
    <state>s1</state>
</states>

<init>
    <state>s1</state>
</init>

<finals>
```

```
<state>s1</state>

</finals>

<arrows>

<arrow>

    <state>s1</state>

    <state>s2</state>

    <item>a</item>

    <cima>Z</cima>

    <trans>Z</trans>

</arrow>

<arrow>

    <state>s2</state>

    <state>s1</state>

    <item>b</item>

    <cima>Z</cima>

    <trans>Z</trans>

</arrow>

</arrows>

<coordenadas>

<estadoCoord>
```

5.3. ESTRUCTURA XML DE LOS EJERCICIOS DE AUTÓMATAS DE PILA75

```
<nombre>s2</nombre>

<x>505</x>

<y>104</y>

</estadoCoord>

<estadoCoord>

<nombre>s1</nombre>

<x>304</x>

<y>139</y>

</estadoCoord>

</coordenadas>

</authomata>

<listaPalabras>

<item>ab</item>

<item>abab</item>

</listaPalabras>

</output>

</ejercicio>
```

5.4. Estructura XML de los ejercicios de máquinas de Turing

Adjuntamos un ejemplo sencillo para mostrar de qué manera almacenamos los datos para un ejercicio de este tipo.

```
<ejercicio>
<tipo>EjMT</tipo>
<enunciado>
Dibuje una maquina de Turing que realice la funcion n = n+2
</enunciado>
<output>
<authomata>
<type>
<item>MaquinaTuring</item>
</type>
<alphabet>
<item>1</item>
</alphabet>
<alphabetP>
<item>1</item>
<item>#</item>
```

5.4. ESTRUCTURA XML DE LOS EJERCICIOS DE MÁQUINAS DE TURING77

```
</alphabetP>

<states>

    <state>s0</state>

    <state>s1</state>

    <state>s2</state>

</states>

<init>

    <state>s0</state>

</init>

<finals>

</finals>

<arrows>

    <arrow>

        <state>s0</state>

        <state>s0</state>

        <item>1</item>

        <scinta>1</scinta>

        <direc>D</direc>

    </arrow>

    <arrow>
```

```
<state>s0</state>

<state>s1</state>

<item>#</item>

<scinta>1</scinta>

<direc>D</direc>

</arrow>

<arrow>

<state>s1</state>

<state>s2</state>

<item>#</item>

<scinta>1</scinta>

<direc>I</direc>

</arrow>

<arrow>

<state>s2</state>

<state>s2</state>

<item>1</item>

<scinta>1</scinta>

<direc>I</direc>

</arrow>
```

5.4. ESTRUCTURA XML DE LOS EJERCICIOS DE MÁQUINAS DE TURING79

```
</arrows>

<coordenadas>

<estadoCoord>

<nombre>s0</nombre>

<x>217</x>

<y>110</y>

</estadoCoord>

<estadoCoord>

<nombre>s1</nombre>

<x>401</x>

<y>51</y>

</estadoCoord>

<estadoCoord>

<nombre>s2</nombre>

<x>576</x>

<y>131</y>

</estadoCoord>

</coordenadas>

</authomata>

<listaPalabras>
```

```
<item>111</item>

</listaPalabras>

<listaCintaPalabras>

<item>11111</item>

</listaCintaPalabras>

<listaPalabrasNo>

<item>1121</item>

</listaPalabrasNo>

<listaCintaPalabrasNo>

<item>21</item>

</listaCintaPalabrasNo>

<listaPalabrasBucle>

</listaPalabrasBucle>

</output>

</ejercicio>
```

Capítulo 6

Palabras clave

Autómata, Turing, pila, Chomsky, Greibach, cinta, simplificación, L^AT_EX,
pila vacía, determinista.

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>