UNIVERSIDAD CARLOS II DE MADRID

Trabajo Fin de Grado

Software para la enseñanza de resolución de problemas de TALF y PL

Autor/a: Natalia Arroyo Pérez

Tutores: D. Germán Gutiérrez Sánchez

D. Juan Manuel Alonso Weber

Fecha: Junio, 2015

Grado en Ingeniería Informática

Agradecimientos

En primer lugar quiero dar las gracias a mis padres y a mi hermano por tolerar mi mal humor, por ser mi apoyo, por darme confianza, ánimos cuando más lo necesito a lo largo de todos estos años. A pesar de los malos ratos y mi desesperación por sacar este trabajo, hoy en día, gracias a vosotros, estoy aquí.

A mi mejor amigo, Alfredo Alba Mansilla, el estar conmigo desde el primer día al último, tanto en la carrera como en el día a día, y todos los buenos momentos compartidos (más los que están por llegar). Gracias. No solo por ser un gran amigo, si no el mejor compañero de prácticas que se puede tener y sin el cual no hubiese sobrevivido todos estos años de carrera.

Gracias a Álvaro Díaz-Guerra Ruiz. No sé que hubiese sido de mí sin esos días en los que lograbas que los problemas y preocupaciones los dejase a un lado. No has dejado que perdiese la sonrisa, haciendo todo lo que estaba en tu mano para que la recuperase. No ha sido un camino fácil la ejecución de este proyecto, así que gracias por tu apoyo, tus ánimos y por estar ahí siempre que te he necesitado.

A mis compañeros de la carrera quiero agradecerles todos los buenos momentos pasados y las ayudas con las prácticas en momentos de desesperación. Con mención especial a Pablo Alberto Fernandes Fernandes, por ayudarme a hacer las primeras investigaciones sobre este trabajo en las prácticas de *Teoría avanzada de la computación*.

También a mis tutores, sin los cuales no hubiese podido crear este proyecto. A pesar de las dudas iniciales sobre mi motivación, creo que he conseguido haceros cambiar de idea. Gracias por haberme ayudado a enfocar los problemas de codificación cuando me ha sido imposible lograr los resultados esperados por mis propios medios. Concretamente a Juan Manuel, por su dedicación y gran exigencia en el trabajo me ha llevado a superarme día a día.

Resumen

En las asignaturas de *Teoría de Autómatas y Lenguajes Formales* y *Procesadores del Lenguaje* se realizan numerosos ejercicios a mano, pero no se cuenta con muchas herramientas, ya que son escasas y contienen limitaciones importantes, para ayudar a comprender los conceptos y cómo resolver los ejercicios que se plantean en las clases. Este motivo impulsó a crear una aplicación que ayudase en ese aspecto.

La materia de *Teoría de Autómatas y Lenguajes Formales* introduce los pilares base que luego se ven en mayor profundidad en *Procesadores del Lenguaje*. En las clases se enseña a resolver los ejercicios y problemas sobre papel, y son necesarias herramientas que resuelvan estos problemas para ayudar a los alumnos a resolver y comparar resultados de grandes y tediosos problemas en los que son fáciles de comer errores. Con las que sí se cuenta para esta tarea, poseen demasiadas limitaciones en el cómputo y en la definición de los problemas.

Posteriormente, en la asignatura de *Procesadores del Lenguaje* se requiere de conocimientos muy avanzados que exceden los obtenidos en *Teoría de Autómatas y Lenguajes Formales*. Este motivo impulsó a crear una aplicación que ayudase en ese aspecto y mejorase la preparación futura de los usuarios.

El objetivo es crear una aplicación necesaria para realizar derivaciones de palabras o sentencias de lenguaje, a modo de aplicación a un análisis sintáctico elemental, ciñéndose a las gramáticas de tipo 2, vistas en *Teoría de Autómatas y Lenguajes Formales*, y además ser capaz de realizar las transformaciones necesarias para convertirlas en Forma Normal de Greibach y Forma Normal de Chomsky.

Concretamente, el software a desarrollar trata de determinar la pertenencia de palabras para gramáticas en Forma Normal de Chomsky, en Forma Normal de Greibach o directamente mediante una derivación por Fuerza Bruta, así como obtener los árboles de derivación en caso de que sí perteneciese. Además, también realizará transformaciones de una gramática hasta dejarla en Forma Normal de Chomsky o en Forma Normal de Greibach.

Palabras clave: Forma Normal de Chomsky, Forma Normal de Greibach, algoritmo CYK, algoritmo Fuerza Bruta, Cocke-Younger-Kasami.

Índice de contenidos

Índice de Tablas

Índice de Ilustraciones

Índice de Gráficas

1. Introducción

En este apartado se aportará una visión general del proyecto dentro de su contexto así como la motivación y los objetivos de dicho proyecto. Además, se explicará la estructura que sigue este documento.

1.1. Motivación

El conocimiento de las asignaturas *Teoría de Autómatas y Lenguajes Formales* y *Procesadores del Lenguaje* es un pilar fundamental en la informática. La materia de *Teoría de Autómatas y Lenguajes Formales* introduce las bases que luego se ven en mayor profundidad en *Procesadores del Lenguaje*. Cuando se estudian los fundamentos de las gramáticas, la mayoría de los recursos disponibles para los estudiantes sobre estos temas son principalmente teóricos. Como es sabido, no se puede conocer algo realmente si no se pone en práctica; y precisamente en esta fase de aprendizaje no se dispone de los suficientes recursos tecnológicos para reforzar o poner en práctica los conocimientos adquiridos.

En las clases se enseña principalmente a resolver los ejercicios y problemas sobre papel, lo cual resulta un proceso tedioso en problemas grandes, en los cuales es fácil cometer errores. Además hay problemas muy interesantes pero inabordables si los tratamos de resolver a mano. Por este motivo, son necesarias herramientas que resuelvan estos problemas.

Posteriormente, en la asignatura de *Procesadores del Lenguaje* los alumnos disponen de muchas herramientas para resolver problemas, pero la asignatura requiere de conocimientos muy avanzados (gramáticas y lenguajes LL(1), LR(0), SLR, LR(1) y LALR) que exceden los que se pueden incluir en el año asignado al estudio de *Teoría de Autómatas y Lenguajes Formales* (gramáticas y lenguajes de tipo 2)., si no se han adquirido inicialmente unos conocimientos adecuados en la asignatura de *Teoría de Autómatas y Lenguajes Formales* dichos programas no pueden ser utilizados con todo el potencial que tienen.

En la asignatura de *Teoría de Autómatas y Lenguajes Formales* se utilizan herramientas de apoyo a la docencia, como por ejemplo *JFLAP*, pero poseen unos recursos computacionales limitados. Poniendo como ejemplo la aplicación mencionada, tan solo pueden utilizarse en la definición de gramáticas 26 *No terminales* (que comprenden los valores entre las letras A-Z en mayúsculas). Cuando ejecutamos una prueba de conversión a Forma Normal de Chomsky en este programa, dado que requiere el uso de muchos *No terminales* auxiliares, tan sólo se pueden usar entre 10 y 14 *No terminales* útiles. Los demás valores del alfabeto se necesitan para la generación de las producciones auxiliares. En el diseño de gramáticas con 10 *No terminales* supone una severa limitación cuando se quiere generar lenguajes complejos.

Ahora bien, algunas ejecuciones de gramáticas con menos de 10 reglas, al ejecutarlas en *JFLAP*, suponen un gran esfuerzo computacional para el programa. Un ejemplo es el problema

que conlleva generar expresiones de suma con números binarios expuesta en el apartado *9.2.Gramática de expresiones de suma de números binarios*.

Una aplicación necesaria para el refuerzo tendría que realizar derivaciones de palabras o sentencias de lenguaje ciñéndose a las gramáticas de tipo 2, vistas en *Teoría de Autómatas y Lenguajes Formales,* y ser capaz de realizar las transformaciones necesarias para convertirlas en Forma Normal de Greibach y Forma Normal de Chomsky. Aquí es donde radica el problema. Las herramientas disponibles para tratar esta transformación son escasas, y en la mayoría de las ocasiones no contemplan todos los requisitos. También poseen muchas restricciones en cuanto al tipo de gramáticas que pueden analizar, además de las restricciones temporales debidas de la complejidad exponencial de derivar gramáticas y lenguajes de tamaño creciente.

Este es el motivo que me llevó a desarrollar una herramienta que ayude en la docencia a los profesores, por ejemplo para plantear ejercicios más interesantes, y a los alumnos, para que puedan obtener la correcta solución de los ejercicios y sirva de refuerzo a lo que se imparta en las clases. Esto sería el equivalente a diseñar un analizador sintáctico al uso en un compilador, cuyo conocimiento he adquirido en la asignatura de *Procesadores del Lenguaje*, y que también puede ser utilizado en la resolución de problemas de dicha materia.

1.2. Objetivos del proyecto

El objetivo principal es construir un programa que resuelva algunos problemas planteados en las asignaturas *Teoría de Autómatas y Lenguajes Formales* y *Procesadores del Lenguaje* para mitigar el problema acerca de las escasas herramientas de ayuda a la docencia.

Concretamente, los ejercicios que resuelve el software a diseñar son:

- Dada una gramática, hallar la Forma normal de Greibach de dicha gramática.
- Dada una gramática, hallar la Forma normal de Chomsky de dicha gramática.
- Una vez obtenida la Forma normal de Chomsky, aplicar el algoritmo CYK que determine si la gramática proporcionada genera una cadena de símbolos de entrada.
- Obtenida la Forma normal de Greibach, determinar si la gramática proporcionada genera una cadena de símbolos de entrada mediante el algoritmo de Fuerza Bruta.
- Determinar, mediante el algoritmo de Fuerza Bruta y sobre una gramática de tipo 2, si la gramática proporcionada genera una cadena de símbolos de entrada.
- Obtener los árboles de derivación de la Forma normal de Greibach, la Forma normal de Chomsky (mediante el algoritmo CYK) y la gramática sin modificaciones estructurales.

1.3. Estructura del documento

El documento consta de distintos apartados. El principal tema de cada uno de ellos se describe a continuación.

- **Estado del arte**. Este apartado muestra la historia sobre el tema de la Teoría de Autómatas y de los Lenguajes Formales así como la explicación de los distintos algoritmos que se utilizaran en el desarrollo de la aplicación.
- **Modelo ciclo de vida**. Define el orden de las tareas o actividades involucradas, la coordinación entre ellas y su enlace y realimentación.
- **Planificación**. Es el proceso de establecer metas y escoge medios para alcanzar dichas metas. Además se podrán detallar los percances acontecidos a lo largo del desarrollo del proyecto y evaluar si la planificación ha sido la adecuada.
- **Presupuesto**. Determina los recursos necesarios y cuánto va a costar completar el proyecto.
- **Análisis del Sistema**. En esta sección se realiza una descripción del sistema y se incluyen los requisitos de usuario y de software, los casos de uso pertinentes y las matrices de trazabilidad derivadas.
- **Diseño del Sistema**. Define la arquitectura de hardware y software, componentes, módulos y datos del sistema para satisfacer ciertos requisitos.
- **Evaluación.** Muestra el plan seguido para realizar las pruebas necesarias para determinar que el sistema funciona adecuadamente así como los resultados obtenidos por cada una de ellas.
- **Estudio de la complejidad.** Este apartado muestra el estudio realizado sobre la complejidad de los algoritmos creados y del programa en general.
- **Especificación del Entorno Tecnológico**. Se describe los lenguajes de programación adecuados para la realización del proyecto y los posibles entornos en los que se puede desarrollar.
- **Legalidad**. Contempla todos los problemas que se pueden encontrar al realizar este proyecto en cuestión de legalidad.
- **Conclusiones y líneas futuras**. En este apartado se realizará una síntesis sobre el desarrollo del proyecto y las conclusiones obtenidas a lo largo del mismo, añadiendo un planteamiento de líneas futuras de mejora.
- **Bibliografía.** Recopilación de todas las fuentes de información que se han utilizado a lo largo del proyecto.
- **Manual de usuario.** Especificación de la guía de uso de la herramienta creada.
- **Anexo I.** En este apartado se mostrará la gramática y las cadenas analizadas para una de las pruebas de análisis de la complejidad.

2. Estado del arte

2.1. Contexto

El estudio de la *Teoría de Autómatas y Lenguajes Formales* se inscribe dentro del campo de la Informática Teórica. El área de la *Teoría de Autómatas y Lenguajes Formales* es independiente de los avances tecnológicos que se han hecho en otras ramas de la Informática y el estudio de esta proviene de la antigüedad [1]. Este campo de investigación tuvo origen antes de los avances de la Informática Teórica, aunque actualmente estén ambas unidas; por tanto, su evolución no depende de las mejoras tecnológicas producidas en el siglo XX.

Si nos centramos en el estudio de los lenguajes naturales, los primeros trabajos emergen en la India, durante el comienzo del primer milenio antes de Cristo, siendo el gramático Panini el causante de un gran apogeo dentro de ese campo. Algunos de los primeros trabajos, anteriores a los de Panini, son las experimentaciones de Akbar el Grande, emperador mogol de la India, que quería comprobar qué lenguaje llegarían a crear niños aislados totalmente, desde su nacimiento, del mundo exterior. Al mismo tiempo en Grecia se desarrollaba una corriente de investigación gramatical, cuyo representante sería Pitágoras [2]. Sin embargo, si nos atenemos al concepto de gramática desde el punto de vista formal, el origen de su estudio se da en investigaciones y avances que se hicieron en el siglo XX.

Varios progresos influyeron en la investigación de los lenguajes formales. En el año 1931, el lógico, matemático y filósofo Kurt Gödel realizó uno de los descubrimientos más importantes del siglo XX dentro de las Ciencias Matemáticas. En el artículo *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme* (en castellano *Sobre proposiciones formalmente indecidibles de Principia Mathematica y sistemas afines*) [3] publicó el teorema que dice: *Toda formulación axiomática, consistente en la teoría de números, contiene proposiciones indecidibles.* La demostración desarrollada afirma que todas las teorías matemáticas son incompletas dado que habrá afirmaciones que son imposibles de probar o de negar. Este concepto terminó con la idea que se tenía hasta el momento, y que investigaba David Hilbert en 1900 en la propuesta *Problema de la decisión* (*Entscheidungsproblem*), de encontrar un sistema general que demostrase cualquier fórmula lógica como verdadera o como falsa.

La propuesta de Gödel alentó a Alan Mathison Turing a investigar y desarrollar la idea. Turing formalizó la idea en el año 1937, considerada como el comienzo de la Informática Teórica. ¿Y por qué se puede considerar el inicio? En la publicación del trabajo que desarrolló Turing sobre el de Gödel se introdujo lo que se denomina *Maquina de Turing.* Con esta herramienta se descubrió un tipo de problema irresoluble, concretamente el *Problema de la*

parada de la máquina de Turing, y fue la predecesora de las máquinas de cálculo automáticas. Además, este trabajo también fue importante ya que se fijó el concepto de algoritmo.

Anterior a la época en la que Turing hizo sus investigaciones, otros matemáticos, como por ejemplo Abu Ja'far Mohammed ibn Musa al-Jowârizmî en el año 825, realizaron investigaciones sobre el tema de definir un concepto para algoritmo. En estas indagaciones ya se vislumbraba la formalización de la palabra algoritmo como un conjunto de reglas que permite obtener un resultado a partir de ciertos datos de partida [4].

Otra de las influencias de los lenguajes formales se dio dentro del campo de la ingeniería eléctrica por parte de Claude Elwood Shannon, matemático e ingeniero electrónico reconocido por su teoría matemática de la comunicación. En el artículo *A symbolic analysis of relay and switching circuits* [5], en castellano *Un análisis simbólico de circuitos de relé y de conmutación*, el matemático norteamericano fijó las bases de la lógica matemática a los circuitos combinatorios y secuenciales. Los principios que estableció, son pilares fundamentales en la teoría de las máquinas secuenciales y de los autómatas finitos.

Estas máquinas secuenciales, también denominadas autómatas, se consideran, en un amplio sentido de la palabra, sistemas que captan señales de su entorno y, como resultado de ello, cambian su estado o realizan una transformación de esta señal para enviar otra distinta como contestación. Las entradas se suelen denominar también estímulos, y las salidas como respuestas. Dentro de esta definición, todas las máquinas formarían parte de la clasificación de autómatas, por este motivo, fue necesario introducir restricciones a este enunciado y así poder realizar un estudio teórico al respecto [4].

Se está hablando de matemáticos, gramáticos, lógicos y filosóficos, entre muchos otros, que indagaban en la lingüística como teoría de los lenguajes y las gramáticas, pero siempre se investigaba como algo ligado a áreas de estudio diferente, por ejemplo en filosofía para asentar normas de la lógica. Hasta los años 40 no se consideró que fuese un campo de estudio concreto. Esta nueva área de estudio científico estuvo muy influida por Avram Noam Chomsky, quien en 1950 transformó las bases de la lingüística matemática con la teoría de las gramáticas transformacionales, y suministró una herramienta que facilitó el estudio y la formalización de los lenguajes naturales [6][7].

Finalmente, el último influjo de los lenguajes formales vino de la mano de Chomsky, cuya influencia en la descripción del concepto de los lenguajes formales fue notable. Clasificó las gramáticas y los lenguajes formales de acuerdo a una jerarquía de cuatro categorías [4], que se explican con mayor detenimiento en el apartado *2.2.2. Jerarquía de Chomsky*. Aún hoy en día se sigue tomando como referencia dicho precepto.

Se ha hablado de todas las influencias que tuvo la *Teoría de Autómatas y Lenguajes Formales* hasta considerarlo un campo de estudio y tener un asentamiento en los conceptos a tratar. Actualmente, dicha teoría influye en diversas áreas [2][4], tales como:

- Lingüística.
- Matemáticas.
- Teoría de la Comunicación.
- Teoría de Control.
- Lógica de los circuitos secuenciales.
- Ordenadores.
- Redes de comunicación y codificadoras.
- Robótica.
- Teoría lógica de los sistemas evolutivos y auto-reproductivos.
- Reconocimiento de patrones.
- Teoría algebraica de lenguajes.
- Fisiología del sistema nervioso.
- Etc.

Todos estos campos tienen en común el manejo de conceptos como el *control*, la *acción* y la *memoria*. Generalmente, los objetos que se controlan o memorizan son símbolos, palabras o frases de algún tipo, de ahí que tengan influencia de la *Teoría de Autómatas y Lenguajes Formales*, cuya área trabaja dichos conceptos.

2.2. Lenguajes y Gramáticas

2.2.1. Lenguaje

La noción de lenguaje se define, según la Real Academia Española, como "*conjunto de señales que dan a entender algo*". Y concretamente para lenguaje informático, "*conjunto de signos y reglas que permite la comunicación con un ordenador*" [8]. Hay similitud entre ambas definiciones. Las dos dicen que integran una serie de normas que tienen un significado.

No solo hay similitud en cuanto a definición, sino también en los componentes. Un lenguaje se compone de un **diccionario**, que indica los significados de las palabras de ese lenguaje, y un **conjunto de reglas**, que describen las sentencias válidas del lenguaje. Este conjunto de reglas es a lo que se le denomina **gramática del lenguaje**.

En el apartado anterior hemos hablado del estudio de los lenguajes naturales. El análisis de los lenguajes se divide en dos: el estudio de las frases (gramática) y su significado (semántica). También, se puede clasificar como: análisis de la forma de las palabras (morfología), la combinación de estas para formar frases correctas (sintaxis) y los distintivos del

lenguaje hablado (fonética). Si escogemos esta última forma de clasificación, hay que destacar que la fonética no se aplica al lenguaje de los ordenadores. Pese a que no se emplee, sí que se estudia mediante el análisis de la voz que disponen las nuevas tecnologías [4].

La definición de lenguaje máquina según la Real Academia Española es "*Conjunto de instrucciones codificadas que una computadora puede interpretar y ejecutar directamente*" [8]. Siguiendo con la línea del lenguaje en los ordenadores, la lingüística es primordial para el diseño y compilación de lenguajes de programación. El objetivo de esta es lograr que las sentencias que se programen tengan sentido claro e inalterable. Estas sentencias son importantes ya que con ellas se construyen gramáticas formales capaces de generar las sentencias deseadas.

2.2.2. Jerarquía de Chomsky

En el punto *2.1. Contexto* se hizo mención a Chomsky por clasificar gramáticas y los lenguajes formales de acuerdo a una jerarquía. A continuación se explicará en qué consiste dicha clasificación.

En 1956 Noam Chomsky publicó el trabajo *Three models for the description of language* [6] en el que explica las propiedades de los distintos tipos de lenguajes formales con sus correspondientes gramáticas. Para Chomsky los tipos de lenguajes formales pueden clasificarse en cuatro grupos: recursivamente enumerable, independientes del contexto, dependientes del contexto y recursivos. Dichos lenguajes también se reconocen como lenguajes de tipo 0, 1, 2 y 3. Tal clasificación es conocida como la Jerarquía de Chomsky [9].

La imagen siguiente muestra cómo es la pertenencia de cada uno de los lenguajes dentro del conjunto de todos los lenguajes posibles y que cada uno de los tipos de gramáticas es más restringido que el anterior.

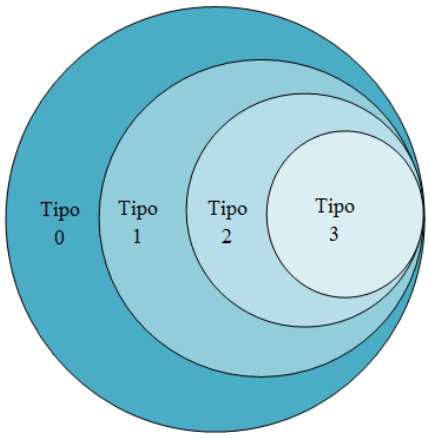


Ilustración 1. Tipos de lenguajes formales

Cada nivel contiene al anterior. Por ejemplo, cualquier lenguaje de tipo3 es a su vez un lenguaje de tipo2. Sin embargo, lo contrario no es cierto, es decir, $L3 \subsetneq L2 \subsetneq L1 \subsetneq L0 \subsetneq$ [2].

La clasificación de los lenguajes es equivalente a la realizada con las gramáticas. Una gramática tiene un conjunto de reglas que describen palabras pertenecientes a un lenguaje específico. Según qué tipo de lenguaje generen, se incluirán dentro de un tipo o de otro.

¿Y cómo podemos determinar qué tipo de lenguaje producen? Para contestar a esta pregunta se utilizan diversos autómatas que son capaces de reconocer los distintos tipos de lenguajes. Estos son Máquina de Turing, Máquina de Turing no determinista, Autómata a pila y Autómata finito.

Para entender mejor esta tipificación se explicara cada una de las posibles clasificaciones.

- **Gramáticas de Tipo 0.**

Estas gramáticas no tienen restricciones. Esta categoría contiene a todas las gramáticas formales, gramáticas que tienen los lenguajes que son aceptados por una máquina de Turing.

Las gramáticas de **tipo 0** son todas las que cumplan $G = (V, \Sigma, Q_0, P)$, con V el alfabeto de los símbolos *Terminales*, Σ conjunto de *No terminales*, Q_0 el axioma y P conjunto de reglas de producción, tal que todas las producciones de P son del tipo siguiente:

$$\gamma \rightarrow \omega, \text{ donde } \gamma, \omega \in (\Sigma \cup V)^*$$

- **Gramáticas de Tipo 1.**

Las gramáticas de **tipo 1** son todas las que cumplan $G = (V, \Sigma, Q_0, P)$, con V el alfabeto de los símbolos *Terminales*, Σ conjunto de *No terminales*, Q_0 el axioma y P conjunto de reglas de producción, tal que todas las producciones de P son del tipo siguiente:

$$\gamma A \delta \rightarrow \gamma \omega \delta, \text{ donde } A \in V \text{ y } \gamma, \delta \in (\Sigma \cup V)^*, \omega \in (\Sigma \cup V)^+.$$

$$S \rightarrow \lambda, \text{ siempre que S sea el axioma}$$

A es un *No terminal* y **γ**, **δ** y **ω** son cadenas de *Terminales* y *No terminales*. **γ** y **δ** pueden estar vacíos, pero **ω** debe contener información.

La palabra vacía λ sólo se puede generar mediante una producción específica del Axioma. Las reglas compresoras $\gamma A \delta \rightarrow \gamma \delta$ solo se admite en una Gramática de Tipo 0.

Estos lenguajes son todos los que pueden ser aceptados por autómatas linealmente acotados.

- **Gramáticas de Tipo 2.**

Las gramáticas de **tipo 2** son todas las que cumplan G = (V, Σ, Q_0, P), con V el alfabeto de los símbolos *Terminales*, Σ conjunto de *No terminales*, Q_0 el axioma y P conjunto de reglas de producción, tal que todas las producciones de P son del tipo siguiente:

$$A \rightarrow \omega, \text{ donde } A \in V \text{ y } \omega \in (\Sigma \cup V)^*$$

$$S \rightarrow \lambda, \text{ siempre que S sea el axioma}$$

A es un *No terminal* y **ω** es una cadena de *Terminales* y *No terminales*.

En caso de tener una producción que genera ω = λ, la gramática se considerará de tipo 0, pero podrá ser formulada como tipo 2 mediante un proceso de limpieza. La palabra vacía λ sólo se puede generar mediante una producción específica del axioma.

La diferencia respecto a las gramáticas de tipo 1 consiste en que se suprime el contexto de forma que en la parte izquierda de la producción sólo puede figurar un único símbolo, que deberá ser un *No terminal*.

Estos son todos los lenguajes que pueden ser aceptados por un autómata a pila.

- **Gramáticas de Tipo 3.**

Las gramáticas de **tipo 3** son todas las que cumplan G = (V, Σ, Q_0, P), con V el alfabeto de los símbolos *Terminales*, Σ conjunto de *No terminales*, Q_0 el axioma y P conjunto de reglas de producción, tal que todas las producciones de P son de uno de los dos tipos siguientes:

$$A \rightarrow a, \text{ donde } A \in V \text{ y } a \in \Sigma$$

$$A \rightarrow aB \text{ ó } A \rightarrow Ba, \text{ donde } A, B \in V \text{ y } a \in \Sigma \text{ [10].}$$

$$S \rightarrow \lambda, \text{ siempre que S sea el axioma}$$

B es un *No terminal* y **a** es un *Terminal*.

Estos lenguajes son los que pueden ser reconocidos por un autómata finito.

La diferencia respecto a las gramáticas de tipo 2 consiste en que se cada producción va a generar un único *Terminal*, que puede ir acompañado o no por otro *No terminal*. Esta configuración la hace adecuada para que se pueda establecer una equivalencia entre gramáticas de tipo 3, los lenguajes que generan y los autómatas finitos que son capaces de reconocer dichos lenguajes.

La siguiente tabla *Tabla 1. Lenguajes con sus autómatas* se muestran un resumen en el que se ve qué lenguajes se dan con cada uno de los tipos y los autómatas que resuelven dichos lenguajes.

Gramática Tipo	Lenguaje	Autómata
0	Recursivamente enumerable	Máquina de Turing
1	Dependiente del contexto	Autómatas linealmente acotados
2	Independiente del contexto	Autómata a pila
3	Regular	Autómata finito

Tabla 1. Lenguajes con sus autómatas

Al igual que ocurría con los tipos de lenguaje, un Autómata Finito puede considerarse como un caso particular de Autómata a Pila y este como una Máquina de Turing.

2.2.3. Formas Normales

Hemos hablado de lenguajes, de gramáticas y, en última instancia, de los autómatas. En este apartado, cuando hablemos de gramáticas, hará referencia a las clasificadas dentro del **tipo 2,** que además son las que pueden ser reconocidas por un autómata.

Los autómatas reconocen gramáticas, pero hay que hacer una especificación. Las gramáticas suelen estar formadas por un amplio conjunto de reglas. El problema básico es que, sin modificación alguna, no parece posible otra opción que probar con todas las combinaciones de producciones para generar las distintas palabras pertenecientes al lenguaje.

Este problema se puede solventar realizando la transformación de la gramática inicial "sucia" a una gramática "limpia". Es necesario el uso de Formas Normales para simplificar las gramáticas libres de contexto y conseguir que los autómatas las reconozcan. Las más importantes son la Forma Normal de Chomsky y la Forma normal de Greibach, las cuales explicaremos en los subapartados.

Una gramática en Forma Normal está compuesta por **símbolos Terminales**, **símbolos No terminales**, **reglas de producción** y un **axioma inicial**. Para analizarla, hay que sustituir los símbolos *No terminales* de parte derecha de cada una de las reglas de producción, con el contenido de las reglas que tengan a la izquierda ese *No terminal* a sustituir. Realizando estas sustituciones, la gramática define un lenguaje formal compuesto por todas las sentencias que están constituidas por símbolos *Terminales* generados de la derivación de las reglas a partir del axioma inicial.

Forma Normal de Chomsky

Una forma de simplificación de las gramáticas aplicable a las gramáticas de tipo 2 y de tipo 3, tal y como hemos dicho anteriormente, es la Forma Normal de Chomsky (FNC). La

gramática que posee esta forma, tiene cada una de las reglas de producción con uno de los siguientes formatos:

$$A \rightarrow B\ C$$

$$A \rightarrow d$$

siendo ***A***, ***B*** y ***C*** símbolos *No terminales*, y ***d*** símbolo *Terminal* [11].

Inicialmente, las gramáticas que no tienen forma suelen estar constituidas por un amplio conjunto de reglas. Si nos adentramos en el análisis de estas reglas, pueden producirse problemas como ver reglas que producen *No terminales* que a su vez no aparecen en la parte izquierda de ninguna otra regla, que nunca se llegue a reglas que generen *Terminales*, recursividad infinita, etc. Por este motivo, antes de realizar la transformación a FNC, hay que realizar una limpieza de dicha gramática. Las transformaciones realizadas para la limpieza se explican en el apartado *2.4.3. Limpieza de gramáticas*.

Cuando la gramática ya tenga una cierta estructura y sin reglas que nos lleven a error, realizamos la transformación a la Forma Normal de Chomsky. Dicha transformación se procederá aplicando uno de los dos pasos siguientes a cada una de las reglas.

- **Reemplazo de Terminales por No terminales**. Si en la parte derecha de alguna regla hay más de un *Terminal*, ya sea una composición de varios *Terminales* o de *Terminales* con *No terminales*, hay que sustituir cada uno de esos *Terminales* por un nuevo *No terminal*. Una vez sustituido, se añadirá una regla para que ese nuevo *No terminal* produzca el *Terminal* reemplazado. Por ejemplo, la regla $S \rightarrow aBCd$ (donde ***a*** y ***d*** son *Terminales*, y ***B*** y ***C*** *No terminales*) se transformaría en las siguientes reglas:

  $$S \rightarrow A\ B\ C\ D$$

  $$A \rightarrow a$$

  $$D \rightarrow d$$

  Si por el contrario, la regla tiene en su parte derecha un solo *Terminal* se dejaría la regla tal y como está. Por ejemplo, la regla $S \rightarrow a$ se dejaría sin modificar.

- **Reemplazo de Producciones de 3 o más No terminales**. Se trata de transformar todas las producciones con 3 o más *No terminales* en la parte derecha de la regla, por una que contenga solo dos. Por ejemplo la regla $S \rightarrow BCD$ (con ***B***, ***C*** y ***D*** *No terminales*) se transformaría en las siguientes reglas [12]:

  $$S \rightarrow B\ A$$

  $$A \rightarrow C\ D$$

Terminadas de transformar todas las reglas, la gramática que se obtenga estará ya en Forma Normal de Chomsky.

Una ventaja de la transformación a Forma Normal de Chomsky es aplicar algoritmos, como por ejemplo el Cocke-Younger-Kasami (CYK), para el reconocimiento de cadenas y realizar una derivación de forma más eficiente.

Forma normal de Greibach

Otra forma de representación de las gramáticas es la Forma Normal de Greibach (FNG). En la FNG todas las reglas de la gramática poseen la forma:

$$Y \rightarrow a\beta$$

Donde a es un donde ***Y*** es un símbolo *No terminal*, a es un símbolo *Terminal* y ***β*** es un conjunto de símbolos *No terminales*.

La Forma Normal de Greibach también requiere de una previa limpieza de la gramática (explicada en el apartado siguiente, es decir, el *2.4.3.*). Hay que realizar dicha limpieza y a continuación efectuar los siguientes cambios para dejarla en FNG [12].

- **Enumerar las variables en un orden arbitrario pero fijo durante el procedimiento.** El axioma principal debe ser la variable con orden 1.
- **Eliminar la recursividad a izquierdas.** Para cualquier regla que tenga, en su parte derecha, el primer símbolo igual que el *No terminal* de la parte izquierda de la regla, se realiza lo siguiente:

  Coger la regla con recursividad, y otra producida por el mismo *No terminal* pero sin recursividad.

$$\begin{cases} A \rightarrow A\alpha 1 \mid A\alpha 2 \mid \ldots \mid A\alpha n \\ A \rightarrow \beta 1 \mid \beta 2 \mid \ldots \mid \beta m \end{cases}$$

  donde $\alpha_i, \beta_i \in (V \cup \Sigma)^*$ y el primer símbolo β_i es diferente de *A*. Las producciones se transformarán así:

$$\begin{cases} A \rightarrow \beta 1 \mid \beta 2 \mid \ldots \mid \beta m \mid \beta 1\, Z \mid \beta 2\, Z \mid \ldots \mid \beta m\, Z \\ A \rightarrow \alpha 1 \mid \alpha 2 \mid \ldots \mid \alpha n \mid \alpha 1\, Z \mid \alpha 2\, Z \mid \ldots \mid \alpha n\, Z \end{cases}$$

  donde Z es una variable completamente nueva.

  Y en una producción $A \rightarrow uBv$ se puede reemplazar por:

$$A \rightarrow uw_1v \mid uw_2v \mid \ldots \mid uw_nv$$

  siendo $B \rightarrow w_1 \mid w_2 \mid \ldots \mid w_n$ todas las producciones de B.

- **Conversión a FNG.** Se trata de transformar todas las producciones de tal manera que todas las reglas empiecen en su parte derecha por un *Terminal* seguido de tantos *No terminales* como variables tenga esa regla. Por ejemplo, la regla $S \rightarrow aBCd$ (donde ***a*** y ***d*** son *Terminales*, y ***B*** y ***C*** *No terminales*) se transformaría en las siguientes reglas:

$$S \rightarrow A\ B\ C\ D$$

$$D \rightarrow d$$

Otro ejemplo sería las reglas $S \rightarrow ABC$ *y* $A \rightarrow a$, que se transformarían en:

$$S \rightarrow a\ B\ C\ D$$

eliminándose $A \rightarrow a$ porque sería una regla inaccesible [13].

Una ventaja de la transformación a FNG es la facilidad para pasar de la gramática a un autómata a pila equivalente. Además, con la FNG la derivación de palabras es algo más eficiente dado que, al tener el primer símbolo de la derecha un *Terminal*, puedes guiar la búsqueda de manera que lo primero a reconocer sea la coincidencia de ese símbolo *No terminal* con la palabra.

Limpieza de gramáticas

La limpieza de las gramáticas permite establecer restricciones necesarias para que las producciones se formen de tal manera que no generen problemas a la hora de derivar palabras. Si no se estableciese estas limitaciones, los árboles de derivación podrían generarse y tener ramas que carezcan de sentido o llegar a un bucle infinito en alguna de las producciones.

La limpieza consiste en la realización de los siguientes pasos:

- **Eliminar las reglas innecesarias.** Estas son las que siguen el formato $X \rightarrow X$. Como se puede ver, generar una producción sobre sí misma sin más contenido es inservible.
- **Quitar los símbolos inaccesibles.** Las reglas que no son accesibles desde el axioma inicial no se generan nunca.
- **Transformar las reglas no generativas.** Dichas reglas son las que generan la palabra vacía. Por ejemplo la eliminación de $X \rightarrow \lambda$, siempre que X no sea el axioma principal:

$$Y \rightarrow abX$$

$$X \rightarrow \lambda$$

Se duplicaría la regla pero no incluyendo valor donde el *Terminal* que genere la palabra vacía. La regla se duplicaría en $Y \rightarrow ab$. Si hay otra regla generada por el mismo *No terminal* que el que genera la palabra vacía, entonces también se dejaría la regla $Y \rightarrow abX$.

Otro ejemplo es:

$$Y \rightarrow aCbbD$$

$$C \rightarrow dE$$

$$C \rightarrow \lambda$$

$$D \rightarrow \lambda$$

$$D \rightarrow Bd$$

que se quedaría con las siguientes reglas:

$$Y \rightarrow aCbbD \mid aCbb \mid abbD \mid abb$$

La convención es no permitir producciones que generen lambda, salvo que el lenguaje incluya la palabra vacía, en cuyo caso se incluye como caso especial del axioma.

- **Modificación de las reglas de redenominación.** Estas reglas son la que un *No terminal* da otro *No terminal*. Para transformar estas producciones hay que sustituir la parte derecha de la regla por todas las producciones que tenga el *No terminal* de la derecha. Por ejemplo:

$$Y \rightarrow C$$

$$C \rightarrow dE$$

$$C \rightarrow aBCd$$

Quedaría como resultado:

$$Y \rightarrow dE$$

$$Y \rightarrow aBCd$$

2.3. Algoritmo Cocke-Younger-Kasami

Con el algoritmo Cocke-Younger-Kasami (CYK) podemos determinar, mediante un análisis sintáctico, si una cadena de símbolos de entrada se puede generar mediante una gramática libre de contexto. La importancia teórica del algoritmo CYK reside en que crea un problema de decisión, es decir, un problema donde las respuestas posibles sean sí o no. Este algoritmo se aplica siempre con una gramática en Forma Normal de Chomsky. El CYK está diseñado con esa restricción, y si se quiere emplear con otro tipo de gramáticas habría que realizar cambios para adaptar el algoritmo.

Con esta gramática en FNC, se efectúa el análisis de la cadena mediante la construcción de una tabla que define el algoritmo CYK [14]. Si la cadena de entrada tiene longitud ***N***, esta

tabla tendrá como tamaño ***NxN***, es decir, el número de columnas y de filas serán tantos *Terminales* como compongan la cadena de entrada. Por ejemplo, si se tuviese una entrada que fuese ***baaba***, siendo ***a*** y ***b*** *Terminales*, la tabla tendría 5 filas y 5 columnas. En cada una de las celdas de la tabla se incluyen los *No terminales* de la parte izquierda de las reglas que den las distintas combinaciones que comprueba el algoritmo. Hay que destacar que esta tabla solo se rellena triangularmente. El seudocódigo a seguir para rellenar esta tabla es el siguiente [15]:

Algoritmo CYK

Entrada:

Gramática G en FNC y cadena de n *Terminales* $w = a_1 a_2 \dots a_n$

Inicializar:

$j = 1$. Para cada i, $1 \leq i \leq n$,

$X_{ij} = X_{i1}$:= conjunto de variables A tales que $A \rightarrow a_i$

es una producción de G.

Repetir:

$j := j + 1$. Para cada i, $1 \leq i \leq n - j + 1$,

X_{ij} := conjunto de variables A tales que $A \rightarrow BC$ es una producción de G, con $B \in X_{ik}$ y $C \in X_{i+k,j-k}$, considerando todos los k tales que $1 \leq k < j - 1$.

Hasta:

$j = n$.

Salida:

$w \in L(G)$ si y sólo si $S \in X_{1n}$

Ilustración 2. Algoritmo CYK

Para explicar este seudocódigo, a continuación se irá realizando paso a paso un ejemplo que ilustre qué es lo que hay que ir completando en cada una de las celdas de la tabla.

Inicialmente tendremos una gramática y una cadena de entrada. La cadena de entrada que vamos a usar para el ejemplo va a ser ***baaba***, y gramática que vamos a emplear es:

S → AB | BC

A → BA | a

B → CC | b

C → AB | a

La tabla inicial para este ejemplo es la siguiente:

Cadena de entrada	**b**	**a**	**a**	**b**	**a**
	i=1	**i=2**	**i=3**	**i=4**	**i=5**
j=1					
j=2					
j=3					
j=4					
j=5					

Tabla 2. Tabla inicial

Primero se rellena la fila j=1. Para completar cada una de sus columnas tenemos que identificar en la gramática qué reglas dan directamente cada uno de los *Terminales* que componen la cadena. Los valores a introducir en la celda (i=1, j=1) serán la regla o reglas que dan *b*. La regla que cumple esta condición es B→b, por lo que incluimos ***B*** en la tabla. En la celda (i=2, j=1) serán la regla o reglas que dan ***a***. Las reglas que cumplen esta condición son A→a y C→a, así que incluimos ***A*** y ***C*** en la tabla. El resto de la fila se completa siguiendo el mismo procedimiento, quedando la primera fila de la siguiente manera.

Cadena de entrada	**b**	**a**	**a**	**b**	**a**
	i=1	**i=2**	**i=3**	**i=4**	**i=5**
j=1	B	A, C	A, C	B	A, C
j=2					
j=3					
j=4					
j=5					

Tabla 3. Tabla primer paso

Ahora se completa la segunda fila, es decir, para j=2. El valor que escogemos de ***K*** es 1. Para completar la fila nos fijamos en las columnas i=1 y i=2 pero con la j=1, que corresponden con X_{ik} y $X_{i+k,j-k}$. Tenemos ***B*** para i=1, y ***A*** y ***C*** para i=2. Ahora lo que hay que buscar son reglas que den las combinaciones de un *No terminal* de X_{ik} junto con otro de $X_{i+k,j-k}$.

Una de las composiciones es ***B*** y ***A***. Con ella miramos qué regla da exactamente esa combinación en su parte derecha. En este caso hemos encontrado A→BA. Otra producción que nos queda por comprobar si existe es la combinación de ***B*** y ***C***. En este caso está la regla es S→BC.

Hemos terminado de analizar todas las combinaciones posibles de este par de posiciones de la tabla, así que en la celda (i=1, j=2) ponemos ***A*** y ***S*** que son las reglas que hemos obtenido.

Si seguimos rellenando las columnas de esta fila y nos queda lo siguiente:

Cadena de entrada	**b**	**a**	**a**	**b**	**a**
	i=1	**i=2**	**i=3**	**i=4**	**i=5**
j=1	B	A, C	A, C	B	A, C
j=2	S, A	B	S, C	S, A	
j=3					
j=4					
j=5					

Tabla 4. Tabla segundo paso

La siguiente fila se complica algo más. En ella seguimos mirando las posiciones de X_{ik} y $X_{i+k,j-k}$. Para (i=1, j=3) hay que hacer el mismo análisis con (i=1, j=1) y (i=2, j=2) y después con (i=1, j=2) y (i=3, j=1). El que se haga así se debe al aumento o disminución de ***j*** e ***i*** y también al aumento en cada ciclo de ***K***.

Terminadas todas las operaciones posibles de esta fila, queda la tabla de la siguiente manera.

Cadena de entrada	**b**	**a**	**a**	**b**	**a**
	i=1	**i=2**	**i=3**	**i=4**	**i=5**
j=1	B	A, C	A, C	B	A, C
j=2	S, A	B	S, C	S, A	
j=3	-	B	B		
j=4					
j=5					

Tabla 5. Tabla tercer paso

El aumento o disminución de ***j*** e ***i*** y el aumento en cada ciclo de ***K*** hace que se vayan rellenando las posiciones de X_{ik} y $X_{i+k,j-k}$ en cada una de las filas. Siguiendo con el proceso, la tabla finalmente quedaría rellena como se muestra a continuación.

Cadena de entrada	b	a	a	b	a
	i=1	i=2	i=3	i=4	i=5
j=1	B	A, C	A, C	B	A, C
j=2	S, A	B	S, C	S, A	
j=3	-	B	B		
j=4	-	S, C, A			
j=5	S, C, A				

Tabla 6. Tabla final

El algoritmo Cocke-Younger-Kasami determina que la cadena de entrada se puede generar con esta gramática si en la celda (i=1, j=N) se encuentra el axioma inicial. En este caso es ***S*** el axioma, y como se encuentra en (i=1, j=5) entonces la palabra se reconoce.

Un problema de su análisis es que se tiene que crear una tabla de tamaño ***NxN,*** siendo ***N*** el valor de la longitud de la cadena. A medida que se aumenta el tamaño, se incrementa en gran medida la memoria necesaria para realizar el análisis. El algoritmo tiene una complejidad polinomial de $O(n^3)$.

Es fácil, una vez obtenida la tabla, sacar el árbol sintáctico. La construcción de este se realiza mediante el almacenamiento de los nodos del árbol, que son los *No terminales* obtenidos de las reglas en cada una de las comprobaciones, como elementos de un *array* [16]. La unión de estos nodos se realiza con el mismo proceso que se sigue mientras se van analizando las combinaciones de los distintos elementos de la tabla.

2.4. Aplicaciones

Una vez tratados los procesos que siguen los algoritmos, nos centraremos en este apartado en analizar los usos que tienen. Como se ha citado hasta ahora en este documento, principalmente se emplean en el ámbito de la informática, concretamente dentro del **análisis sintáctico**.

Hemos hablado del uso de la Forma Normal de Chomsky dentro del análisis sintáctico en informática y las investigaciones de Avram Noam Chomsky, pero ¿y el análisis sintáctico de los demás campos de la lingüística? Noam Chomsky se centró en la lingüística en general por lo que también transformó los conceptos de este campo de investigación. Si se indaga sobre la forma de generar las gramáticas, tal y como las hemos aprendido en asignaturas como Lengua y Literatura, o el mismo concepto de gramática, se ve que la tarea del lingüista es encontrar un sistema de expresión en reglas de manera que produzca el conjunto de resultados que se desee, y cuyo conjunto de reglas formen lo denominado gramática [17]. Se ve claramente que la idea de gramática que tenía Avram Noam Chomsky se aplica también a las ideas de esta fuera del

campo de la informática. Al igual que ocurre con las reglas de codificación y las palabras aceptadas, las gramáticas dentro de la Lengua y Literatura también pueden ser analizadas por el mismo algoritmo.

También podemos encontrar a Chomsky como autoridad en Robótica. Para entender tal influencia expliquemos en primer término que la robótica tiene como uno de los principales estudios la conexión entre el cerebro y el lenguaje. Es una idea muy interesante que ya abordó Chomsky en sus investigaciones, las cuales hoy en día se siguen desarrollando.

Actualmente es imposible saber si podría existir el cerebro y el lenguaje, uno sin el otro. Experimentos con personas, por ejemplo con personas sordomudas de nacimiento, han demostrado que el cerebro desarrolla instintivamente un lenguaje en sus estructuras más elementales (esta idea ya la hemos mencionado anteriormente con las investigaciones del emperador mogol de la India Akbar el Grande). Que el cerebro desarrolle un lenguaje, aunque sea básico, cuando no se puede aprender ni aplicar el lenguaje que se transmite oralmente, es decir, aprender la lengua materna, que se crea en a la posibilidad de crear dispositivos robóticos que imiten al ser humano en ese aspecto. Si una persona puede crear desde cero y tan solo con la influencia del entorno un lenguaje elemental para vivir, ¿por qué no puede tener un robot ese tipo de lenguaje esencial? Pues todas estas investigaciones tienen un pilar en el que basarse, y es el que hizo Chomsky al demostrar que el lenguaje y las estructuras cerebrales tienen una correspondencia entre sí, derivada de los estímulos que se reciben del mundo exterior [18]. Y cómo no, estas estructuras, o gramática base, tendrán que seguir una serie de reglas que son posibles de analizar con los algoritmos tratados en entre trabajo.

Centrándonos en un uso particular que se haga a alguno de los algoritmos, podemos ver que el Cocke-Younger-Kasami (CYK) puede aplicarse para distintos fines.

Uno de los sitios donde utilizan el CYK es en estudios de bioinformática, como por ejemplo en las investigaciones de las secuencias de ADN. Herón Molina-Lozano publicó un artículo en el que se explica el proceso que se sigue, usando una variación el CYK, para realizar un análisis con gramáticas de lógica Fuzzy (son reglas lógicas pero que los resultados dan valores continuos en vez de discretos) codificadas para detectar subcadenas de ADN.

Para detectar los componentes de las subcadenas se realizan varias etapas de modificación, tanto en la forma de representar la gramática reconocedora como del algoritmo CYK, y llegar a identificar una secuencia concreta del ADN (proteína, carbohidrato, etc.) [19].

Otro ámbito donde emplean el algoritmo CYK es en el reconocimiento de formas bidimensionales. Un ejemplo concreto es la detección, mediante códigos cadena y basándose la naturaleza cíclica, de los contornos triangulares de figuras.

Para definir el triángulo, se usa como medida unitaria de los lados la variable ***a*** y como definición de un ángulo la variable ***b***. Tomando esta definición, un triangulo isósceles tendría como fórmula $a^nba^nba^kb$, con $n \geq 1$ $k \geq 1$ y $k \neq n$. O un triángulo equilátero sería $a^nba^nba^nb$, con $n \geq 1$. Basándonos en cómo se expresan estas figuras, las cadenas *ababaab*, *babaaba* y *abaabab* representarían el mismo triángulo solo que tomando el inicio de la representación desde distintos puntos de su área. Con esta representación nos referimos a que el mismo triángulo puede empezar a representarse por cualquier punto de sus aristas o de sus ángulos.

Poder reconocer este tipo de cadenas con el algoritmo CYK, supone hacer modificaciones en este, como hemos mencionado anteriormente, teniendo en cuenta la naturaleza cíclica y la representación del área a determinar.

Las modificaciones realizadas sobre el CYK hacen que el algoritmo quede de la siguiente manera:

Inicio.

for i=l t o n

ti,l = {A|A → ai ϵ P}

for j=2 to n

for i=l to n

ti,j = {A|A → BC ϵ P,B ϵ ti,k , C ϵ t(i+k-1)mod+1, j-k $1 \leq k < j$}

Fin.

Este cambio produciría una transformación en la forma de completar la tabla, encontrando en esta no sólo si la cadena es válida, si no hasta qué punto se puede dar repeticiones de *a*. Esta conclusión se ve mejor con el ejemplo siguiente:

Tenemos la cadena *abaabaaba*. Con el algoritmo modificado, la tabla resultante es la que se muestra en *Tabla 7. Resultado CYK modificado*.

		S			S			S
		C			C			C
E	D		E	D		E	D	
C		F	C		F	C		F
F	E		F	E		F	E	
A	B, C, F	A	A	B, C, F	A	A	B, C, F	A
a	**b**	**a**	**a**	**b**	**a**	**a**	**b**	**a**

Tabla 7. Resultado CYK modificado

Este triangulo es detectado como verdadero, ya que se trata de un triángulo equilátero. Se reconoce como resultado positivo porque en la esquina superior derecha se ve el símbolo del axioma principal. Además, también podemos detectar cuales son los puntos en los que puede aumentarse el lado del triangulo para dar otros triángulos, porque aparece el axioma principal en la fila superior [20].

2.5. Herramientas similares

Anteriormente a la realización del proyecto, se hizo un estudio de los programas existentes actualmente en el mercado capaces de resolver parte de los problemas o los mismos problemas que se afrontan en este proyecto a realizar. Dichas cuestiones son: la resolución de la transformación de una gramática a Forma normal de Chomsky, la transformación a Forma normal de Greibach, determinar si una palabra pertenece a una gramática mediante el algoritmo Cocke-Younger-Kasami, pertenece a una gramática en Forma normal de Greibach o determinar su pertenencia mediante fuerza bruta. Todos estos puntos se engloban dentro el campo de la Teoría de Autómatas y lenguajes formales, por lo que se especificó el estudio a los programas similares dentro de ese tema.

A continuación, se muestra una descripción de cada uno de los programas encontrados y en qué medida se ajustan a las características buscadas, así como una conclusión de los datos recopilados comparándolos con este proyecto.

2.5.1. JFLAP

JFLAP (acrónimo de an Interactive Formal Languages and Automata Package) es un software, programado en *Java*, diseñado para trabajar con temas relacionados con la *Teoría de autómatas y lenguajes formales* [21]. Se trata de un software de código abierto que no requiere instalación ya que se utiliza mediante un ejecutable [22].

JFLAP es una herramienta práctica muy fácil de usar. Los temas con los que se puede trabajar son: autómatas finitos no deterministas, autómatas a pila, máquinas de Turing y diversos tipos de gramáticas y análisis sintácticos. *JFLAP* consta de las siguientes características:

- Se centra en varios tipos de análisis, entre ellos, análisis de fuerza bruta, gramáticas LL (1) y gramáticas SLR (1).
- Se puede construir máquinas de Moore y Mealy.
- Permite analizar cadenas y convertirlas en árboles sintácticos.
- *JFLAP* realiza diagramas de autómatas y guardarlo en un archivo de imagen en los formatos: jpeg, jpg, gif o bmp, o exportar a formato SVG. Mientras realizas el diagrama puedes mover gráficamente a la posición de los estados por toda la pantalla sin que afecte a los arcos de las uniones de las transiciones.
- Permite pasar de un autómata finito no determinista a un autómata finito determinista.
- Los autómatas a pila se reconocen como aceptados si la pila se queda vacía o si hay un símbolo de final de pila.
- La etiqueta de transición para un autómata finito tiene cero, uno o varios símbolos. Estos símbolos se pueden definir como rengo de valores, por ejemplo [1-9] para aceptar las entradas 1, 2, 3, 4, 5, 6, 7, 8 y 9 o con los caracteres de la a *a* la *z*, que se representan como [a-z] y [A-Z]. Pero el autómata solo permite un carácter por entrada.
- Reconoce el uso de la palabra vacía, que se representa con el símbolo λ.
- Se puede rehacer y deshacer las acciones realizadas.
- Puedes definir de antemano el tipo de gramática a usar o crear una gramática y analizarla desde distintos tipos de autómatas.

Pese a las numerosas características que tiene JFLAP, hay que destacar que carece de otras. La lista con los puntos carentes o errores es la siguiente:

- JFLAP no tiene un número ilimitado de caracteres para definir los *Terminales* y No *terminales*. Esto es un problema, y delimita el número de reglas que puede tener una gramática.
- Los *Terminales* y los *No terminales* tan solo se pueden definir con un símbolo.
- Acepta palabras, en cuanto al algoritmo CYK (el cual solo lo llevan las nuevas versiones), que no debería de darlas como buenas.
- Genera gramáticas distintas si realizas la transformación a FNC paso a paso que si la realizas automáticamente.

2.5.2. Jaccie.

Jaccie es una herramienta que se utiliza para visualizar las técnicas de compilación. El nombre de *Jaccie* viene determinado por el acrónimo de ***Ja****va-based* ***c****ompiler* ***c****ompiler with* ***i****nteractive* ***e****nvironment*. Es el sucesor de *SIC* (acrónimo de ***S****malltalk-based* ***I****nteractive* ***C****ompiler*) que fue implementado entre 1989 y 1995 por lo que el desarrollo de Jaccie se inició después de este último, en 1995. La creación de este programa se llevó a cabo mediante la composición de distintos proyectos de estudiantes [23].

La herramienta se compone de dos principales componentes: un generador de analizadores léxicos y una variedad de generadores sintácticos para LL(1), LALR(1), SLR(1).

Proporciona un entorno integrado con editores especiales para las definiciones del scaner y parser, herramientas para ver información de los iniciales (first) y los siguientes (follow) [24].

La estructura de *Jaccie* se puede englobar en lo que se muestra en la siguiente imagen.

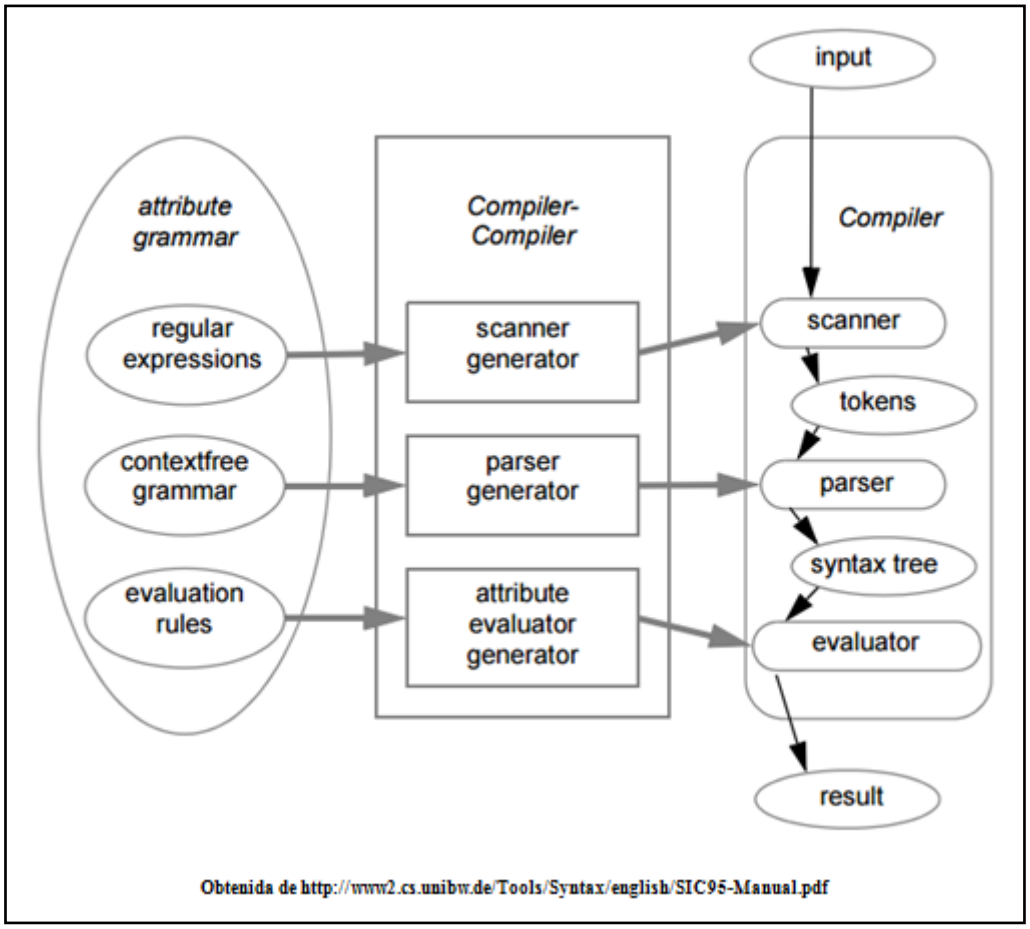


Ilustración 3. Estructura Jaccie

Las características, en cuanto a las acciones que puede realizar el programa, son las siguientes [24]:

- Puede leer ficheros externos con los parámetros necesarios en cada caso.
- Hay que definir de antemano los *Terminales* y *No terminales* junto con la gramática.
- Genera el árbol kantorovich (o árbol sintáctico) para un *token* introducido.

2.5.3. Lex

Lex genera programas para analizar lexicalmente un texto. Los archivos de entrada contienen, por un lado, expresiones regulares para ser encontradas, y por otro, las acciones escritas en *C* para ser ejecutadas (estas acciones son las reglas de la gramática que pueden o no generar las expresiones regulares). En la ejecución de dichas acciones, se determina si las expresiones de entrada son encontradas; en caso de ser así, se mostrará la lista de reglas que ha llevado a cabo para demostrar la afirmación [25]. El reconocimiento de las expresiones se lleva a cabo internamente por un autómata finito determinista que es generado automáticamente por el programa.

Una característica de este analizador, es que admite gramáticas con ambigüedad. Y además, dentro de esta ambigüedad, se puede escoger el camino más largo o más corto que lleva a dar la expresión regular analizada.

También hay que destacar que si una expresión regular no es aceptada, *Lex* guarda las ejecuciones seguidas y que demuestran que no se ha podido obtener [26].

2.5.4. Yacc

Yacc, acrónimo de Yet Another Compiler-Compiler (en español *Otro generador de compiladores más*), es un generador de analizadores sintácticos LALR. Fue creado por Stephen C. Johnson en el laboratorio *Bell Labs*. Se trata de un ejecutable, programado en *C*, en el que el usuario especifica las estructuras de entrada y un código, o gramática, que al que atenerse para reconocer cada dicha estructura es correcta o no [26]. Las gramáticas son descritas usando una variante de Forma de Backus Naur (BNF) [27].

Sus características son las siguientes:

- *Yacc* permite analizar gramáticas con ambigüedad. Para romper dicha ambigüedad en el análisis, hay que establecer unas reglas de prioridad.
- La gramática de entrada puede ser tan compleja como un lenguaje de programación, o tan simple como solo unas pocas líneas.
- Con *Yacc* puedes obtener archivos de salida en los cuales se contemple la información especificada de los resultados obtenidos (por ejemplo, los conflictos derivados de las ambigüedades, para diagnósticos, etc.) [28].

2.5.5. Bison

El programa *Bison*, perteneciente al proyecto *GNU* (Iniciado por Richard Stallman con el objetivo de crear un sistema operativo completamente libre), es un generador de analizadores sintácticos. Su objetivo es convertir una gramática independiente del contexto, escrita como LALR(1), en un programa que describa la misma gramática pero en lenguaje *C*, *C++* o *Java* [29]. El ejecutable generado permite determinar si una cadena introducida por parámetro pertenece o no al lenguaje generado por la gramática. El reconocimiento se hace mediante la construcción del árbol sintáctico [30].

En cuanto a la generación del fichero de entrada, se debe diferenciar los *Terminales* de los *No terminales* de la gramática. Los símbolos *Terminales* se denominan tokens. Por norma general, los tokens se escriben en mayúsculas y los *No terminales* en minúsculas. También, en la definición de estos se puede añadir dígitos (aunque no al principio de la definición) y puntos (únicamente en *No terminales*).

Otras características de *Bison* son las siguientes:

- Si se usa *Bison* junto a *Flex*, se crea una herramienta para construir compiladores de lenguajes.
- Es compatible al 100% con *Yacc*.

2.5.6. Comparativa y conclusiones

Para poder realizar la comparativa entre las distintas herramientas encontradas, se han comparado las características de la herramienta de este proyecto con las particularidades de cada herramienta encontrada.

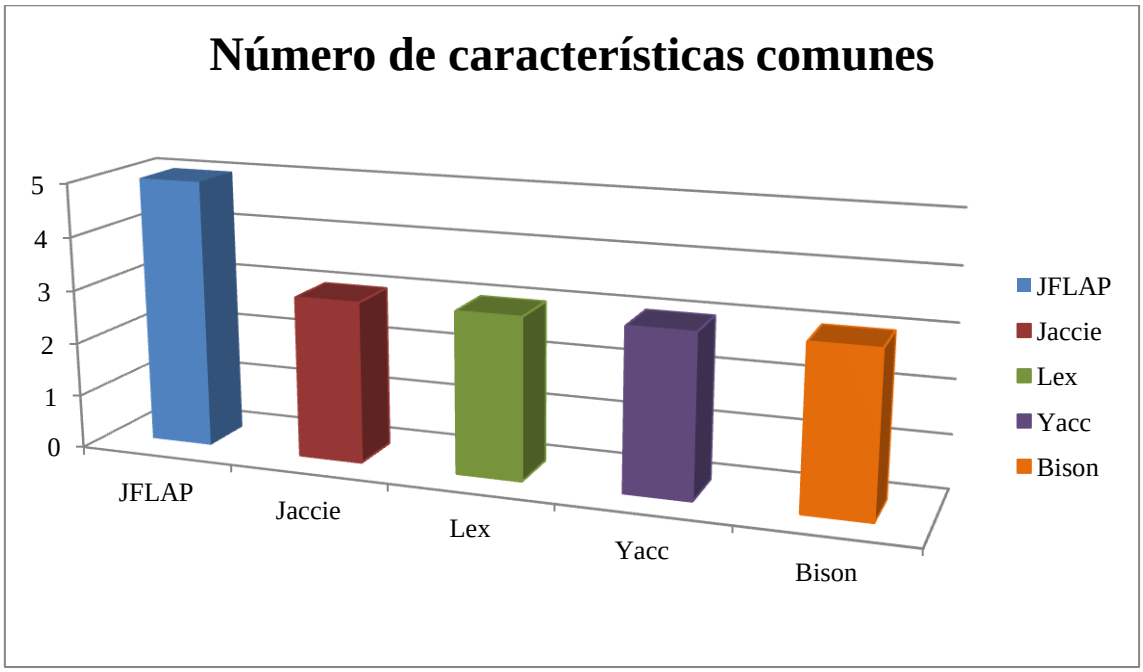
características Competencia					
características	**JFLAP**	**Jaccie**	**Lex**	**Yacc**	**Bison**
Lectura gramática	SI	SI	SI	SI	SI
Lectura cadena	SI	SI	SI	SI	SI
Transformar a FNG	NO	NO	NO	NO	NO
Transformar a FNC	SI	NO	NO	NO	NO
Análisis por fuerza bruta	SI	SI	SI	SI	SI
Análisis con gramática FNG	NO	NO	NO	NO	NO
Análisis con gramática FNC	SI	NO	NO	NO	NO

Tabla 8. Tabla comparativa herramientas

Los distintos significados de los puntos tratados en la tabla se explican a continuación para tratar las ambigüedades o confusiones producidas por la corta definición.

- **Lectura gramática**. El programa es capaz de leer un archivo externo que contenga la gramática a analizar.
- **Lectura cadena**. La herramienta está capacitada para leer un archivo externo que contenga las distintas cadenas que se van a analizar.
- **Transformar a FNG**. El programa realiza una transformación automática, de una gramática dada, a FNC.
- **Transformar a FNC**. La herramienta es capaz de realizar una transformación automática, de una gramática dada, a FNC.
- **Análisis por fuerza bruta**. El programa analiza una cadena con su respectiva gramática mediante fuerza bruta.
- **Análisis con gramática FNG**. El programa puede analizar una cadena con su respectiva gramática, siendo esta en formato FNG.
- **Análisis con gramática FNC**. La herramienta puede analizar una cadena con su respectiva gramática, siendo esta en formato FNG.

A continuación se muestra un análisis gráfico de lo que se muestra en la tabla, para poder observar mejor qué partes cumplen las distintas herramientas.



Gráfica 1. Comparativa de herramientas

Dentro de todos los programas encontrados, el que más se ajusta a las características del proyecto desarrollado y mayor competidor de este es *JFLAP*. A continuación se concretará más la comparativa.

Algunos de los programas analizados pueden usar parte del conjunto de gramáticas que forman las clasificadas como FNC. Esto quiere decir, que habrá gramáticas que cumplan con las restricciones de FNC pero no sean admitidas por estos.

JFLAP permite las mismas gramáticas que acepta el programa del proyecto. Además, al igual que ocurre con la herramienta creada, a su vez es capaz de realizar las transformaciones necesarias a la gramática hasta contener una estructura con formato FNC. En cambio, la aplicación programada es capaz de realizar cambios para que estén las gramáticas en FNG, cosa que el JFLAP no contempla dicha opción. En este sentido, la herramienta es mejor pues posee ambas opciones.

JFLAP, cuanto a la gestión gráfica, tiene ventaja. Siempre es más agradable trabajar con una interfaz bien desarrollada. Si nos fijamos bien en tiempo de ejecución, una interfaz gráfica, cuanto más compleja sea, tardará más que una que no lo tenga. Al carecer de esta, la herramienta de este proyecto juega a favor en cuanto a tiempo de ejecución. Depende de cómo se evalúe el tener o no interfaz, puede ganar una u otra. Quiere decir, que ganaría JFLAP si se tiene en cuenta el poseer una interfaz, pero ganaría el proyecto al realizar el cómputo en menor tiempo de ejecución.

JFLAP puede crear y guardar el autómata en diferentes formatos, y almacenar los pasos realizados para transformar la gramática que le pasen por parámetro. En este punto *JFLAP* gana. Es cierto que el proyecto realizado guarda en formato .txt los distintos pasos realizados para la transformación de las gramáticas y la determinación de pertenencia o no de las palabras analizadas. Por tanto, aunque *JFLAP* posea el punto ganador en un principio, si nos fijamos solo en las características que comparten (pues en el proyecto no se ha incluido nada de realizar autómatas), *JFLAP* y la herramienta están empatados.

La herramienta creada tiene una gran ventaja sobre el punto flaco de *JFLAP*, y es la definición ilimitada de distintos *Terminales* y *No terminales*. Como se ha mencionado con anterioridad, *JFLAP* delimita el tamaño de la gramática al no disponer de todos los *Terminales* y *No terminales* que se deseen. El programa implementado carece de esas restricciones.

Concluimos que la herramienta implementada no anda tan alejada de las características de las aplicaciones en el actual mercado e incluso tiene alguna mejora con respecto a su mayor competidora.

3. Modelo ciclo de vida

Es necesario establecer una metodología de trabajo que permita una gestión correcta de los recursos y los tiempos de entrega. Tras estudiar los modelos de ciclo de vida disponibles, el esquema a seguir a lo largo del proyecto, para realizar la construcción del mismo, es el Modelo de Ciclo de Vida en Cascada con realimentación. Este esquema identifica el conjunto de fases que comprenden el desarrollo del proyecto en su totalidad pudiendo regresar a las anteriores para llevar a cabo los cambios pertinentes, y así poder continuar con las siguientes etapas. En la imagen *Ilustración 4. Ciclo de vida* se muestra de forma visual como son los pasos de este modelo y la conexión entre cada uno ellos.

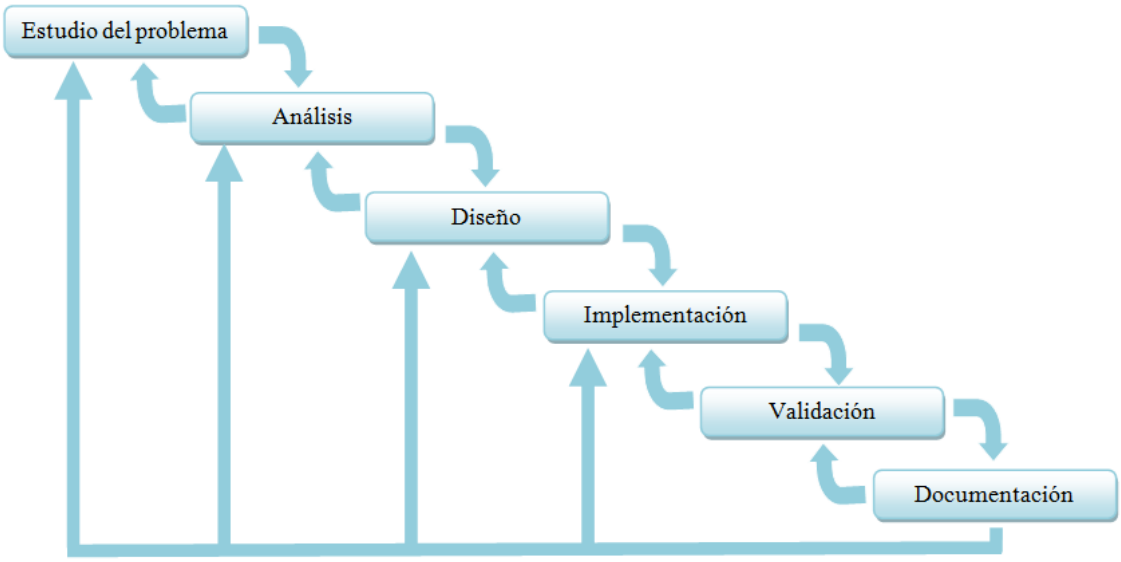


Ilustración 4. Ciclo de vida

Los distintos significados de las fases de la ilustración anterior, representadas por una corta definición, se explican a continuación.

- **Estudio del problema:** realizar un estudio inicial del problema planteado, necesidades a solventar, estudio de plataformas, herramientas a utilizar, planteamiento de objetivos y límites temporales.
- **Análisis del sistema:** plasmar los casos de uso y requisitos.
- **Diseño del sistema:** definir la arquitectura del sistema y componentes que lo forman.
- **Implementación del sistema:** desarrollar el sistema, adecuándolo a lo definido en las fases anteriores.
- **Validación del sistema:** realización de diversas pruebas que verifiquen la integridad, usabilidad y eficiencia.
- **Documentación:** redacción de la memoria del proyecto.

4. Planificación

Se ha elaborado un plan de trabajo para realizar el proyecto satisfactoriamente en el periodo de tiempo establecido. En caso de que los límites de tiempo se vean irremediablemente afectados, los miembros del proyecto han de asumir responsabilidades, puesto que el plazo de entrega es inamovible.

4.1. Planificación inicial

En la tabla que se muestra al final de este apartado, *Tabla 9. Planificación inicial*, se han especificado todas las tareas que deben ser realizadas desde el comienzo hasta el fin del proyecto así como la estimación en días y horas de cada una de ellas.

El Trabajo Fin de Grado, según el reglamento de la Universidad Carlos III de Madrid, equivale a 12 créditos ECTS. Cada crédito equivale a 25 horas de trabajo de dedicación del estudiante, por lo que el total de horas dedicadas al TFG asciende a 300 [31].

El Trabajo Fin de Grado comenzó el 1 de Septiembre y se acordó terminarlo en 31 de Mayo. Según estas especificaciones, se establece que se dedique al proyecto 2 horas diarias. Estos cálculos se obtienen de dividir las horas totales del proyecto entre:

22(días laborables cada mes de 30 días) x 8 (total de meses menos todos los días no lectivos y los días de estudio de los exámenes) + 4 (meses con 31 días).

Estos cálculos sacan un total de 1,66666… que se redondea a 2, resultado de dividir las 300 horas entre los 180 días. Queda establecido que la dedicación sean 2 horas diarias al día.

Id	Nombre de la Tarea	Fecha de comienzo	Fecha de fin	Duración en días*	Duración en horas
1	**Investigación**				
	Estudio tecnologías	9/9/2014	12/9/2014	4	8
	Estado del arte	15/9/2014	19/9/2014	5	10
2	**Análisis**				
	Requisitos de usuario	22/9/2014	23/9/2014	1	2
	Requisitos del software	24/9/2014	25/9/2014	1	2
	Casos de uso	26/9/2014	29/9/2014	1	2
	Identificación de los usuarios principales	29/9/2014	30/9/2014	1	2
3	**Diseño**				
	Definición arquitectura	1/10/2014	7/10/2014	5	10
	Definición sistema	8/10/2014	24/10/2014	13	26

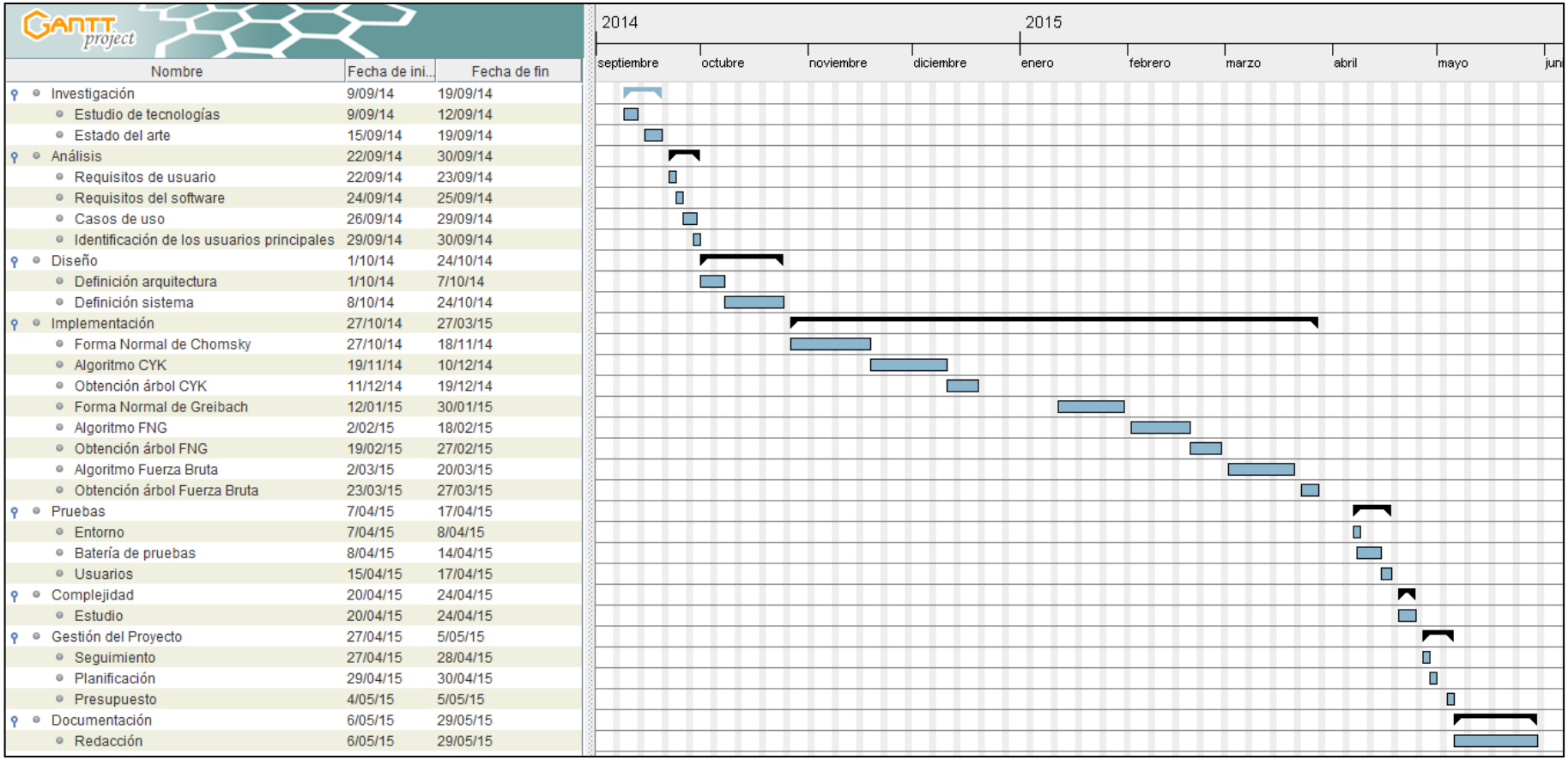
4	**Implementación**				
	Forma Normal de Chomsky	27/10/2014	18/11/2014	16	32
	Algoritmo CYK	19/11/2014	10/12/2014	14	28
	Obtención árbol CYK	11/12/2014	19/12/2014	7	14
	Forma Normal de Greibach	12/1/2014	30/1/2015	15	30
	Algoritmo FNG	2/2/2015	18/2/2015	12	24
	Obtención árbol FNG	19/2/2015	27/2/2015	7	14
	Algoritmo Fuerza Bruta	2/3/2015	20/3/2015	15	30
	Obtención árbol Fuerza Bruta	23/3/2015	27/3/2015	5	10
5	**Pruebas**				
	Entorno	7/4/2015	8/4/2015	1	2
	Batería de pruebas	8/4/2015	14/4/2015	4	8
	Usuarios	15/4/2015	17/4/2015	2	4
6	**Complejidad**				
	Estudio	20/4/2015	24/4/2015	4	8
7	**Gestión del Proyecto**				
	Seguimiento	27/4/2015	28/4/2015	1	2
	Planificación	29/4/2015	30/4/2015	1	2
	Presupuesto	4/5/2015	5/5/2015	1	2
8	**Documentación**				
	Redacción	6/5/2015	29/5/2015	14	28

Tabla 9. Planificación inicial

*Los cálculos realizados son los días laborables, es decir, se excluyen fin de semana y festivos.

4.2. Diagrama de Gantt planificación inicial

En este apartado mostraremos el Diagrama de Gatt, en el cual se expone el tiempo de dedicación previsto para diferentes tareas o actividades a lo largo del proyecto. Este se ha realizado con la herramienta *Gantt proyect*.



Nombre	Fecha de ini...	Fecha de fin
Investigación	9/09/14	19/09/14
Estudio de tecnologías	9/09/14	12/09/14
Estado del arte	15/09/14	19/09/14
Análisis	22/09/14	30/09/14
Requisitos de usuario	22/09/14	23/09/14
Requisitos del software	24/09/14	25/09/14
Casos de uso	26/09/14	29/09/14
Identificación de los usuarios principales	29/09/14	30/09/14
Diseño	1/10/14	24/10/14
Definición arquitectura	1/10/14	7/10/14
Definición sistema	8/10/14	24/10/14
Implementación	27/10/14	27/03/15
Forma Normal de Chomsky	27/10/14	18/11/14
Algoritmo CYK	19/11/14	10/12/14
Obtención árbol CYK	11/12/14	19/12/14
Forma Normal de Greibach	12/01/15	30/01/15
Algoritmo FNG	2/02/15	18/02/15
Obtención árbol FNG	19/02/15	27/02/15
Algoritmo Fuerza Bruta	2/03/15	20/03/15
Obtención árbol Fuerza Bruta	23/03/15	27/03/15
Pruebas	7/04/15	17/04/15
Entorno	7/04/15	8/04/15
Batería de pruebas	8/04/15	14/04/15
Usuarios	15/04/15	17/04/15
Complejidad	20/04/15	24/04/15
Estudio	20/04/15	24/04/15
Gestión del Proyecto	27/04/15	5/05/15
Seguimiento	27/04/15	28/04/15
Planificación	29/04/15	30/04/15
Presupuesto	4/05/15	5/05/15
Documentación	6/05/15	29/05/15
Redacción	6/05/15	29/05/15

Ilustración 5. Diagrama planificación inicial

En la tabla se han mostrado los días exactos planificados. Se han eliminado, cuando ha sido necesario, los días de fiesta. En el diagrama de Gantt esta acción no ha sido posible contemplarla, por tanto, la duración visual de determinadas tareas puede ser mejor.

4.3. Método de seguimiento y control de desviaciones

Como método de seguimiento se realizará, al menos, una reunión quincenal en la que estén presentes todos los miembros del proyecto y los tutores de este. En ella se identificarán los puntos que se han de realizar, especificando qué deberán contener, y una evaluación de los contenidos que se han realizado hasta la fecha.

Aparte de la reunión semanal, se podrán poner en contacto (de forma personal u online) con los tutores del proyecto si en algún momento se produce algún problema.

4.4. Planificación final

En la tabla *Tabla 10. Planificación final*, se han especificado todas las tareas que han sido realizadas desde el comienzo hasta el fin del proyecto así como la duración, tanto en días como horas, de cada una de ellas.

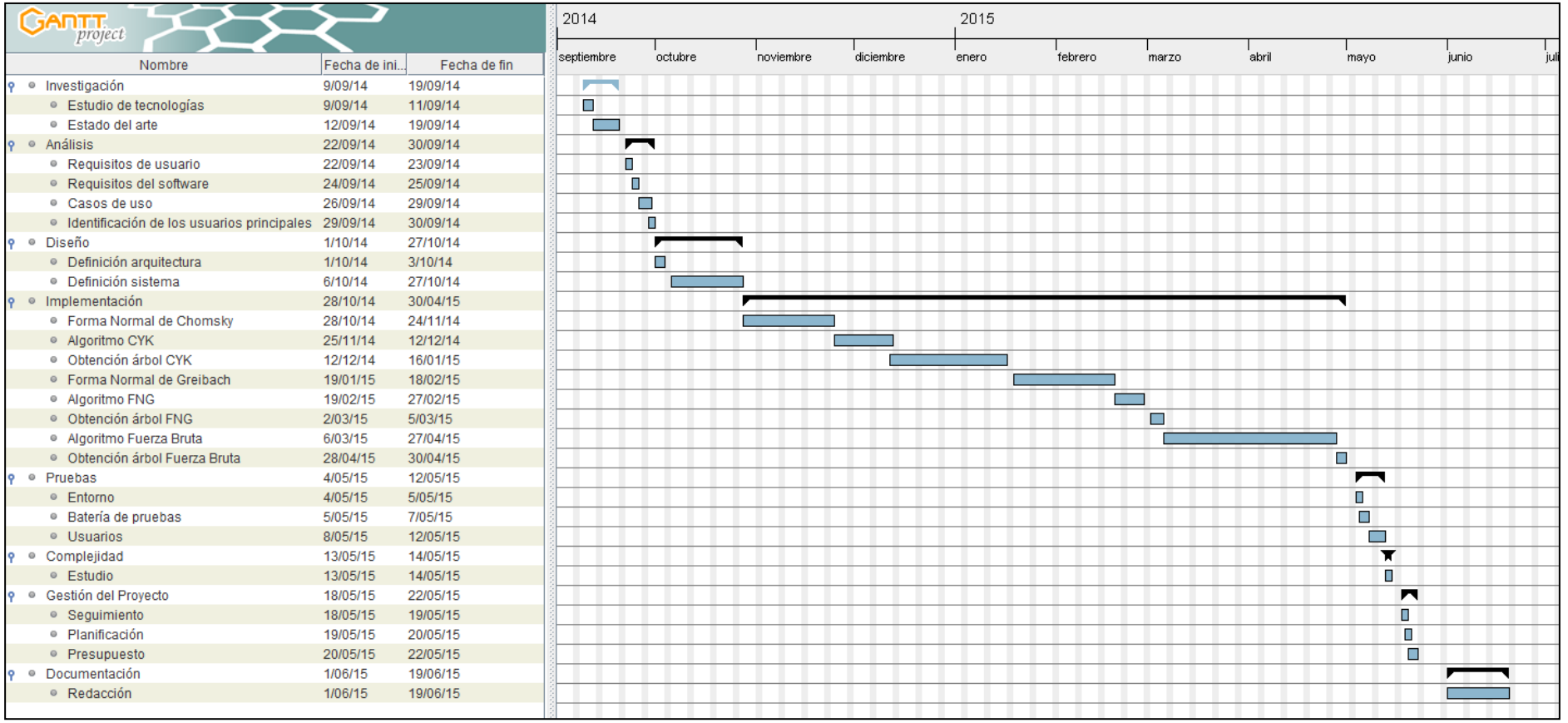
Id	Nombre de la Tarea	Fecha de comienzo	Fecha de fin	Duración en días	Duración en horas
1	**Investigación**				
	Estudio tecnologías	9/9/2014	11/9/2014	3	6
	Estado del arte	12/9/2014	19/9/2014	6	12
2	**Análisis**				
	Requisitos de usuario	22/9/2014	23/9/2014	1	2
	Requisitos del software	24/9/2014	25/9/2014	1	2
	Casos de uso	26/9/2014	29/9/2014	1	2
	Identificación de los usuarios principales	29/9/2014	30/9/2014	1	2
3	**Diseño**				
	Definición arquitectura	1/10/2014	3/10/2014	3	6
	Definición sistema	6/10/2014	27/10/2014	16	32
4	**Implementación**				
	Forma Normal de Chomsky	28/10/2014	24/11/2014	19	38
	Algoritmo CYK	25/11/2014	19/12/2014	17	34
	Obtención árbol CYK	12/1/2014	16/1/2014	5	10
	Forma Normal de Greibach	19/1/2014	18/2/2014	22	44

	Algoritmo FNG	19/2/2014	27/2/2014	7	14
	Obtención árbol FNG	2/3/2014	5/3/2014	4	8
	Algoritmo Fuerza Bruta	6/3/2014	27/4/2015	30	60
	Obtención árbol Fuerza Bruta	28/4/2015	30/4/2015	3	6
5	**Pruebas**				
	Entorno	4/5/2015	5/5/2015	1	2
	Batería de pruebas	5/5/2015	7/5/2015	3	6
	Usuarios	8/5/2015	12/5/2015	3	6
6	**Complejidad**				
	Estudio	13/5/2015	14/5/2015	2	4
7	**Gestión del Proyecto**				
	Seguimiento	18/5/2015	19/5/2015	1	2
	Planificación	19/5/2015	20/5/2015	1	2
	Presupuesto	20/5/2015	22/5/2015	2	4
8	**Documentación**				
	Redacción	1/6/2015	19/6/2015	15	30

Tabla 10. Planificación final

4.5. Diagrama de Gantt planificación final

En este apartado mostraremos el Diagrama de Gatt, en el cual se expone el tiempo de dedicación de las diferentes tareas o actividades a lo largo del proyecto. Este se ha realizado con la herramienta *Gantt proyect*.



Nombre	Fecha de ini...	Fecha de fin
Investigación	9/09/14	19/09/14
Estudio de tecnologías	9/09/14	11/09/14
Estado del arte	12/09/14	19/09/14
Análisis	22/09/14	30/09/14
Requisitos de usuario	22/09/14	23/09/14
Requisitos del software	24/09/14	25/09/14
Casos de uso	26/09/14	29/09/14
Identificación de los usuarios principales	29/09/14	30/09/14
Diseño	1/10/14	27/10/14
Definición arquitectura	1/10/14	3/10/14
Definición sistema	6/10/14	27/10/14
Implementación	28/10/14	30/04/15
Forma Normal de Chomsky	28/10/14	24/11/14
Algoritmo CYK	25/11/14	12/12/14
Obtención árbol CYK	12/12/14	16/01/15
Forma Normal de Greibach	19/01/15	18/02/15
Algoritmo FNG	19/02/15	27/02/15
Obtención árbol FNG	2/03/15	5/03/15
Algoritmo Fuerza Bruta	6/03/15	27/04/15
Obtención árbol Fuerza Bruta	28/04/15	30/04/15
Pruebas	4/05/15	12/05/15
Entorno	4/05/15	5/05/15
Batería de pruebas	5/05/15	7/05/15
Usuarios	8/05/15	12/05/15
Complejidad	13/05/15	14/05/15
Estudio	13/05/15	14/05/15
Gestión del Proyecto	18/05/15	22/05/15
Seguimiento	18/05/15	19/05/15
Planificación	19/05/15	20/05/15
Presupuesto	20/05/15	22/05/15
Documentación	1/06/15	19/06/15
Redacción	1/06/15	19/06/15

Ilustración 6. Diagrama Gantt planificación final

En la tabla se han mostrado los días exactos planificados. Se han eliminado, cuando ha sido necesario, los días de fiesta. En el diagrama de Gantt esta acción no ha sido posible contemplarla, por tanto, la duración visual de determinadas tareas puede ser mejor.

4.6. Comparativa de las planificaciones

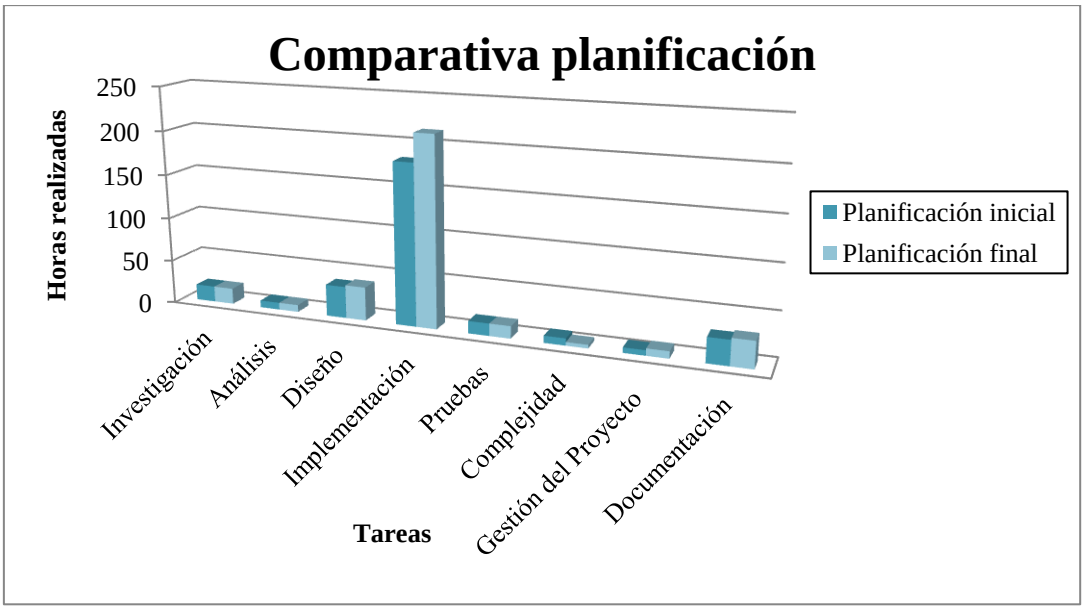
En esta sección se evaluarán las desviaciones de tiempo producidas en cada fase del proyecto con respecto a la planificación inicial. La siguiente tabla llamada *Tabla 11. Comparativa de las planificaciones* muestra la diferencia en días que existe entre la planificación inicial de cada tarea y el tiempo que al final se ha requerido para completarlas.

Id	Nombre de la Tarea	Duración en horas estimada	Duración en horas real	Desfase de tiempo
1	**Investigación**	18	18	0
2	**Análisis**	8	8	0
3	**Diseño**	36	38	2
4	**Implementación**	182	214	32
5	**Pruebas**	14	14	0
6	**Complejidad**	8	4	-4
7	**Gestión del Proyecto**	6	8	2
8	**Documentación**	28	30	2
	TOTAL	300	334	34

Tabla 11. Comparativa de las planificaciones

La diferencia negativa en el desfase de tiempo indica un desarrollo en menor tiempo sobre la planificación inicial, mientras que una positiva muestra un retraso. Según los datos de la tabla, la terminación del proyecto se ha realizado en un total de 334 horas.

A continuación, en el gráfico se muestra visualmente la desviación para cada una de las actividades realizadas y entregadas.



Gráfica 2. Gráfica comparativa

Como se puede observar en la tabla anterior, en la mayoría de tareas ha habido un desfase positivo, es decir, se ha tenido que emplear más tiempo del estimado. Al igual que ha ocurrido en términos de tiempo, también se ha producido un desfase en la fecha esperada para concluir el proyecto.

Ha concluido su realización 19 días más tarde de lo planificado inicialmente. Esto se debe principalmente a que durante la realización del proyecto han surgido distintas dificultades.

El proyecto dio comienzo, tal y como estaba planificado inicialmente, el día 9 de Septiembre de 2014 (dado que hasta esa fecha los anteriores días de Septiembre no eran lectivos). Las tareas de *Investigación* y *Análisis* se realizaron según lo previsto. La parte de *Diseño*, pese a tener una desviación de un día, con sus respectivas 2 horas de trabajo, está dentro de los límites aceptables en cuanto a retraso.

No obstante, la tarea de *Implementación* fue la que peor predicción tuvo. El grado de dificultad de comprensión de los diversos algoritmos fue:

- Algoritmo CYK.
- Algoritmo de transformación a Forma Normal de Greibach.
- Algoritmo de derivación por Fuerza Bruta.

Como queda reflejado en las horas de trabajo, el algoritmo de derivación por Fuerza Bruta y el algoritmo de transformación a Forma Normal de Greibach fueron los que más tardaron en realizarse. El aumento de trabajo fue debido a la programación de la eliminación de la recursividad a izquierdas.

Hay que señalar que se produjo un ligero desfase en el estudio con usuarios, dado que no todos los participantes podían realizar las pruebas en los días planteados. Tan solo hubo que retrasar algunas pruebas un día más que se vieron compensadas por las horas ganadas en otras tareas, pero es un punto a tener en cuenta. No se previó el margen que hay que dar a los usuarios por los problemas que puede generar la coordinación de todos ellos en unos días establecidos.

A pesar de todo ello, el proyecto se ha terminado dentro del plazo de entrega, es decir, no se ha sobrepasado la fecha límite del 22 de Junio del 2015 establecida por la universidad, aunque sí que se vio superada la fecha límite planificada.

5. Presupuesto

En esta sección se estimarán los costes asociados al proyecto. En las siguientes tablas se muestran de forma desglosada cada uno de los salarios.

5.1. Costes de personal

Según las etapas en las que se divide el proyecto el precio por cada una de ellas es diferente.

Costes de personal				
Rol	**Rol dentro del equipo**	**Salario/hora**	**Salario/mes***	**Salario/proyecto****
Jefe de proyecto	Principal responsable del planteamiento y la ejecución acertados del proyecto.	30,00 €	1.320,00 €	5.400,00 €
Diseño	Responsable del diseño de la aplicación	18,00 €	792,00 €	3.240,00 €
Responsable de Pruebas	Gestor de pruebas de software	18,00 €	792,00 €	3.240,00 €
Analista	Encargado de recopilar y especificar los requisitos de usuario y los requisitos software a desarrollar	14,00 €	616,00 €	2.520,00 €
Programador	Responsable de escribir, depurar y mantener el código fuente del programa	13,00 €	572,00 €	2.340,00 €
Documentador	Encargado de recopilar información y realizar la redacción del estado del arte	13,00 €	572,00 €	2.340,00 €
TOTAL	-	-	-	**19.080,00 €**

Tabla 12. Costes de personal

*Los cálculos realizados por vez equivalen a los 22 días laborables correspondientes a un mes con 30 días en total, trabajando 2 horas diarias.

**Los cálculos de este salario son los equivalentes a todo el tiempo de duración del proyecto.

5.2. Costes de la tasa de prestaciones complementarias

A continuación se detalla los costes de las tasas de prestaciones complementarias que se tiene por cada empleado.

Prestaciones	Salario proyecto/persona	Número empleados	Salario/proyecto
Seguro médico	1.000,00 €	6	6.000,00 €
Seguro odontológico	500,00 €	6	3.000,00 €
Seguro invalidez	500,00 €	6	3.000,00 €
TOTAL	-	-	**12.000,00 €**

Tabla 13. Costes prestaciones

Al tratarse de un proyecto compuesto por 6 personas, este salario hay que multiplicarlo por esas 6.

5.3. Costes administrativos

Se va proceder a detallar los costes administrativos que vienen asociados con la creación de un nuevo proyecto y que se adquieren externamente al proyecto al depender de agentes externos. Estos costes se pagan por cada proyecto que se realice. En este caso al tratarse de uno solo, tan solo se pagan una vez.

Producto	Salario
Asesoría legal	1.000,00 €
Asesoría Administrativa	800,00 €
TOTAL	**1.800,00 €**

Tabla 14. Costes administrativos

5.4. Costes Hardware

Dentro de los cálculos de hardware se encuentra las compras de los diferentes ordenadores necesarios para el proyecto, así como impresoras y demás material hardware necesario.

Costes de hardware				
Producto	**Cantidad**	**Precio unitario**	**Periodo amortización**	**Precio Total**
Ordenador portátil para desarrollo y pruebas	1	699,00 €	3 años	233,00 €
Impresora	1	90,00 €	3 años	30,00 €
TOTAL	-	-	-	**263,00 €**

Tabla 15. Costes de hardware

5.5. Costes Software

De forma análoga a los costes de hardware, en este apartado se recogen los costes de software necesarios para la ejecución del proyecto.

Costes de software				
Producto	**Cantidad**	**Precio unitario**	**Periodo amortización**	**Precio Total**
Sistema Operativo Windows 8	1	0,00 € *	3 años	0,00 €
Microsoft Office 2010	1	90,00 €	3 años	30,00 €
Plataforma Eclipse	1	0,00 €**	3 años	0,00 €
TOTAL	-	-	-	**30,00 €**

Tabla 16. Costes de software

*La licencia del sistema operativo viene incluida al realizar la compra del ordenador portátil.

**La plataforma Eclipse posee licencia gratuita para uso público de la plataforma.

5.6. Material fungible

Evaluando las características de este proyecto, se ha realizado una estimación de gastos en material fungible.

Costes de material fungible			
Producto	**Cantidad**	**Precio unitario**	**Precio Total**
Papel DIN-A 4	3 cajas	13,00 €	39,00 €
Tóner	4 packs	28,50 €	114,00 €
Material oficina	1 pack	20,00 €	20,00 €
Libros de consulta	2 libros	0,00 € *	0,00 €
TOTAL	-	-	**173,00 €**

Tabla 17. Costes material fungible

*Los libros de consulta utilizados en este proyecto los proporciona gratuitamente la Universidad Carlos III de Madrid.

5.7. Costes fijos

Costes fijos			
Producto	**Cantidad**	**Precio unitario**	**Precio Total**
Electricidad	8 meses	22,50 €	180,00 €
Internet	8 meses	30,00 €	240,00 €
Local trabajo	8 meses	150,00 €	120,00 €
Local reunión	8 meses	0,00 €	0,00 €*
TOTAL	-	-	**420,00 €**

Tabla 18. Costes de luz e internet

*El local de reunión lo proporciona gratuitamente la Universidad Carlos III de Madrid.

5.8. Coste total del proyecto

Se detallan en este punto el coste total de realizar el proyecto planteado.

COSTES RECURSOS Y PRESTACIONES	
RECURSOS HUMANOS	
Coste de sueldos	19.080,00 €
TOTAL RECURSOS HUMANOS	
	19.080,00 €
PRESTACIONES	
Seguro médico	6.000,00 €
Seguro odontológico	3.000,00 €
Seguro invalidez a largo plazo	3.000,00 €
TOTAL PRESTACIONES	
	12.000,00 €
RECURSOS EXTERNOS	
Asesoría legal	1.000,00 €
Asesoría Administrativa	800,00 €
TOTAL RECURSOS EXTERNOS	
	1.800,00 €
RECURSOS FÍSICOS	
Hardware	263,00 €
Software	30,00 €

Material fungible	173,00 €
Costes fijos	1.620,00 €
TOTAL RECURSOS FÍSICOS	
	2.086,00 €
TOTAL	
	34.966,00 €

Tabla 19. Costes recursos y prestaciones

A lo anterior descrito hay que sumarle el margen de riesgo, el beneficio y el IVA del proyecto. La suma de todo ello se muestra en la siguiente tabla, la cual determina el coste total del proyecto.

COSTES TOTAL PROYECTO	
CONCEPTO	
Costes recursos y prestaciones	34.966,00 €
TOTAL CONCEPTO	
	34.966,00 €
COSTES DIRECTOS	
Margen de riesgo (15%)	5.244,90 €
Beneficio (15 %)	5.244,90 €
IVA (21 %)	7.342,86 €
TOTAL COSTES DIRECTOS	
	17.832,66 €
TOTAL PROYECTO	
	52.798,66 €

Tabla 20. Costes totales del proyecto

Por tanto, el coste final asociado a este proyecto es de 52.798,66 €.

6. Análisis del Sistema

6.1. Descripción general del sistema

La herramienta a desarrollar facilitará al usuario la realización de problemas de las asignaturas *Teoría de autómatas y lenguajes formales* y de *Procesadores del lenguaje*. Este software está adaptado para el uso en los sistemas operativos Windows, el cual se realizará como una aplicación descargable y ejecutable mediante la consola de comandos.

Como se ha especificado en el apartado *1.2. Objetivos del proyecto*, esta herramienta resolverá unos determinados tipos de ejercicios, los cuales se ven cómo resolverlos en las asignaturas para las que está diseñado este proyecto.

La clase de usuarios a la que va dirigida dicha aplicación es cualquier persona que tenga conocimiento en Lenguajes formales.

6.2. Requisitos de usuario

En este apartado se especifican los requisitos de usuario. Con ellos se extrae las especificaciones de la aplicación, para conocer la funcionalidad que espera el usuario, así como las limitaciones. Se ha separando los requisitos en cuanto a si tratan de la funcionalidad como si son de limitaciones, es decir, se han separado en Requisitos de Usuario y Requisitos de Restricción.

Cada uno de los requisitos seguirá este esquema.

RU/RUR-XX			
Título			
Fuente			
Prioridad		**Necesidad**	
Verificabilidad		**Claridad**	
Estabilidad			
Descripción			

Tabla 21. Plantilla requisitos usuario

- **RUC/RUR-XX**. Requisito de Usuario Capacidad/Requisitos Usuario de Restricción – Dígito#1 Dígito#2, esta nomenclatura permitirá identificar cada requisito de manera unívoca. Además, con esta identificación se facilita la trazabilidad de cada uno de los requisitos.
- **Título**. Descripción inicial del requisito.
- **Fuente**. Origen de cada uno de los requisitos.

- **Prioridad**. Medida a tener en cuenta de cara a realizar la planificación. Los valores contemplados en este proyecto son Alta, Media y Baja.
- **Necesidad**. Mide es grado de importancia de los requisitos. Estos pueden ser Esencial, Deseable u Opcional.
- **Verificabilidad**. Es necesario que se pueda comprobar de forma fehaciente que los requisitos de usuario han sido implementados. En este proyecto los atributos válidos para este campo son Alta, Media y Baja.
- **Claridad**. Medición la ambigüedad de cada uno de los requisitos. Es por ello que se ha decidido medir la claridad como Alta, Media y Baja.
- **Estabilidad**. Contempla la posibilidad de que los requisitos de usuario permanezcan estables o no a lo largo de todo el proyecto. La especificación se realizará por escrito, es decir, no se contempla con un rango de valores.
- **Descripción**. Proporciona una definición detallada para cada uno de los requisitos.

6.2.1. Identificación de Requisitos de Capacidad

La lista de requisitos de usuario es la siguiente.

- **RUC-01.** Introducir gramática.
- **RUC-02.** Introducir cadena.
- **RUC-03.** Ruta gramática.
- **RUC-04.** Ruta cadena.
- **RUC-05.** Cambiar gramática.
- **RUC-06.** Cambiar cadena.
- **RUC-07.** Forma Normal de Chomsky.
- **RUC-08.** Cadena Forma Normal de Chomsky.
- **RUC-09.** Especificación Forma Normal de Chomsky.
- **RUC-10.** Especificación árbol FNC.
- **RUC-11.** Especificación pertenencia cadenas FNC.
- **RUC-12.** Tabla FNC.
- **RUC-13.** Cadena Forma Normal de Greibach.
- **RUC-14.** Especificación Forma Normal de Greibach.
- **RUC-15.** Especificación árbol FNG.
- **RUC-16.** Especificación pertenencia cadenas FNG.
- **RUC-17.** Cadena Fuerza Bruta.
- **RUC-18.** Especificación árbol FB.
- **RUC-19.** Especificación pertenencia cadenas FB.
- **RUC-20.** Salir del programa.

RUC-01			
Título	Introducir gramática		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá introducir una gramática dentro de un .txt.		

Tabla 22. RUC-01

RUC-02			
Título	Introducir cadena		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá introducir una cadena a comparar dentro de un .txt.		

Tabla 23. RUC-02

RUC-03			
Título	Ruta gramática		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario tendrá la opción de introducir la ruta del archivo .txt que contenga la gramática a analizar.		

Tabla 24. RUC-03

RUC-04			
Título	Ruta cadena		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario tendrá la opción de introducir la ruta del archivo .txt que contenga la cadena a analizar.		

Tabla 25. RUC-04

RUC-05			
Título	Cambiar gramática		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	La herramienta permitirá al usuario realizar cambios en la gramática mientras el sistema esté en funcionamiento.		

Tabla 26. RUC-05

RUC-06			
Título	Cambiar cadena		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	La herramienta permitirá al usuario realizar cambios en la cadena mientras el sistema esté en funcionamiento.		

Tabla 27. RUC-06

RUC-07			
Título	Forma Normal de Chomsky		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá realizar una transformación de una gramática a Forma Normal de Chomsky.		

Tabla 28. RUC-07

RUC-08			
Título	Cadena Forma Normal de Chomsky		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá realizar una comparación de una cadena con una gramática en Forma Normal de Chomsky.		

Tabla 29. RUC-08

RUC-09			
Título	Especificación Forma Normal de Chomsky		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema.		
Descripción	El usuario podrá contemplar en un fichero .txt los pasos seguidos para la realización de la Forma Normal de Chomsky.		

Tabla 30. RUC-09

RUC-10			
Título	Especificación árbol FNC		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá visualizar en un fichero .txt los árboles generados al comparar una cadena con una gramática en Forma Normal de Chomsky.		

Tabla 31. RUC-10

RUC-11			
Título	Especificación pertenencia cadenas FNC		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá contemplar en un fichero .txt la pertenencia de las palabras comprobadas en gramática en Forma Normal de Chomsky.		

Tabla 32. RUC-11

RUC-12			
Título	Tabla FNC		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá visualizar en un fichero .txt las tablas generadas por el algoritmo CYK al comprobar una cadena con una gramática en Forma Normal de Chomsky.		

Tabla 33. RUC-12

RUC-13			
Título	Cadena Forma Normal de Greibach		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá contemplar una comparación de una cadena con una gramática en Forma Normal de Greibach.		

Tabla 34. RUC-13

RUC-14			
Título	Especificación Forma Normal de Greibach		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá visualizar en un fichero .txt los pasos seguidos para la realización de la Forma Normal de Greibach.		

Tabla 35. RUC-14

RUC-15			
Título	Especificación árbol FNG		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá contemplar en un fichero .txt los árboles generados al comparar una cadena con una gramática en Forma Normal de Greibach.		

Tabla 36. RUC-15

RUC-16			
Título	Especificación pertenencia cadenas FNG		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá contemplar en un fichero .txt la pertenencia de las palabras comprobadas en gramática en Forma Normal de Greibach.		

Tabla 37. RUC-16

RUC-17			
Título	Cadena Fuerza Bruta		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá visualizar una comparación de una cadena con una gramática, realizando el proceso de comparación en Fuerza Bruta.		

Tabla 38. RUC-17

RUC-18			
Título	Especificación árbol FB		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá contemplar en un fichero .txt los árboles generados al comparar una cadena con una gramática, realizando el proceso de comparación en Fuerza Bruta.		

Tabla 39. RUC-18

RUC-19			
Título	Especificación pertenencia cadenas FB		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá contemplar en un fichero .txt la pertenencia de las palabras comprobadas con una gramática, realizando el proceso de comparación en Fuerza Bruta.		

Tabla 40. RUC-19

RUC-20			
Título	Salir del programa		
Fuente	Usuario		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El usuario podrá terminar la ejecución del programa cuando desee.		

Tabla 41. RUC-20

6.2.2. Identificación de Requisitos de Restricción

Para determinar unívocamente las especificaciones no funcionales que debe cumplir el proyecto, se han clasificado los requisitos de restricción en requisitos de seguridad, escalabilidad, uso y de formación a usuarios.

La lista de requisitos de restricción es la siguiente.

- **RUR-01.** Manejo de información.
- **RUR-02.** Idioma
- **RUR-03.** Tecnología Windows.
- **RUR-04.** Formación usuarios.
- **RUR-05.** Usabilidad.

Requisitos de escalabilidad

RUR-01			
Título	Manejo de información		
Fuente	Usuario		
Prioridad	Media	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema deberá manejar volúmenes altos de información.		

Tabla 42. RUR-01

Requisitos de uso

RUR-02			
Título	Idioma		
Fuente	Usuario		
Prioridad	Media	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema estará disponible, exclusivamente, en castellano.		

Tabla 43. RUR-02

RUR-03			
Título	Tecnología Windows		
Fuente	Usuario		
Prioridad	Media	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema deberá funcionar y visualizarse correctamente en todas las versiones del sistema operativo Windows.		

Tabla 44. RUR-03

Requisitos de formación a usuarios

RUR-04			
Título	Formación usuarios		
Fuente	Usuario		
Prioridad	Media	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	Se otorgará a los usuarios de los conocimientos necesarios para el aprovechamiento del sistema mediante un manual de usuario.		

Tabla 45. RUR-04

RUR-05			
Título	Usabilidad.		
Fuente	Usuario.		
Prioridad	Media.	**Necesidad**	Deseable.
Verificabilidad	Alta.	**Claridad**	Alta.
Estabilidad	Durante toda la vida del sistema.		
Descripción	El sistema seguirá unos criterios de usabilidad para facilitar el uso de la herramienta.		

Tabla 46. RUR-05

6.3. Requisitos de software

Cada uno de los requisitos seguirá este esquema.

RSF/RSNF-XX			
Título			
Fuente			
Prioridad		**Necesidad**	
Verificabilidad		**Claridad**	
Estabilidad			
Descripción			

Tabla 47. Plantilla requisitos de software

- **RSF/RSNF-XX**. Requisito de software funcionales/Requisito de software funcionales – Dígito#1 Dígito#2, esta nomenclatura permitirá identificar cada requisito de software de manera unívoca.
- **Título**. Descripción inicial del requisito.
- **Fuente**. Origen de cada uno de los requisitos.
- **Prioridad**. Medida a tener en cuenta de cara a realizar la planificación. Los valores contemplados en este proyecto son Alta, Media y Baja.
- **Necesidad**. Mide es grado de importancia de los requisitos. Estos pueden ser Esencial, Deseable u Opcional.
- **Verificabilidad**. Es necesario que se pueda comprobar de forma fehaciente que los requisitos de usuario han sido implementados. En este proyecto los atributos válidos para este campo son Alta, Media y Baja.
- **Claridad**. Medición la ambigüedad de cada uno de los requisitos. Es por ello que se ha decidido medir la claridad como Alta, Media y Baja.
- **Estabilidad**. Contempla la posibilidad de que los requisitos de usuario permanezcan estables o no a lo largo de todo el proyecto. La especificación se realizará por escrito, es decir, no se contempla con un rango de valores.
- **Descripción**. Proporciona una definición detallada para cada uno de los requisitos.

6.3.1. Identificación de Requisitos Funcionales

La lista de estos requisitos de software funcionales es la siguiente:

- **RSF-01.** Forma Normal de Chomsky.
- **RSF-02.** Guardado Forma Normal de Chomsky.
- **RSF-03.** Pertenencia Forma Normal de Chomsky.
- **RSF-04.** Guardada pertenencia Forma Normal de Chomsky.
- **RSF-05.** Árbol pertenencia Forma Normal de Chomsky.
- **RSF-06.** Forma Normal de Greibach.
- **RSF-07.** Guardado Forma Normal de Greibach.
- **RSF-08.** Pertenencia Forma Normal de Greibach.
- **RSF-09.** Guardada pertenencia Forma Normal de Greibach.
- **RSF-10.** Árbol pertenencia Forma Normal de Greibach.
- **RSF-11.** Pertenencia Fuerza Bruta.
- **RSF-12.** Guardada pertenencia Fuerza Bruta.
- **RSF-13.** Árbol pertenencia Fuerza Bruta.

RSF-01			
Título	Forma Normal de Chomsky		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe realizar una transformación de una gramática a FNC.		

Tabla 48. RSF-01

RSF-02			
Título	Guardado Forma Normal de Chomsky		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe guardar los pasos de la transformación de la gramática a FNC.		

Tabla 49. RSF-02

RSF-03			
Título	Pertenencia Forma Normal de Chomsky		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe determinar la pertenencia de una palabra dentro de una gramática en FNC.		

Tabla 50. RSF-03

RSF-04			
Título	Guardada pertenencia Forma Normal de Chomsky		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe guardar la determinación de la pertenencia de una palabra dentro de una gramática en FNC.		

Tabla 51. RSF-04

RSF-05			
Título	Árbol pertenencia Forma Normal de Chomsky		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe guardar la el árbol generado por la pertenencia de una palabra dentro de una gramática en FNC.		

Tabla 52. RSF-05

RSF-06			
Título	Forma Normal de Greibach		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe realizar una transformación de una gramática a FNG.		

Tabla 53. RSF-06

RSF-07			
Título	Guardado Forma Normal de Greibach		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe guardar los pasos de la transformación de la gramática a FNG.		

Tabla 54. RSF-07

RSF-08			
Título	Pertenencia Forma Normal de Greibach		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe determinar la pertenencia de una palabra dentro de una gramática en FNG.		

Tabla 55. RSF-08

RSF-09			
Título	Guardada pertenencia Forma Normal de Greibach		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe guardar la determinación de la pertenencia de una palabra dentro de una gramática en FNG.		

Tabla 56. RSF-09

RSF-10			
Título	Árbol pertenencia Forma Normal de Greibach		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe guardar la el árbol generado por la pertenencia de una palabra dentro de una gramática en FNG.		

Tabla 57. RSF-10

RSF-11			
Título	Pertenencia Fuerza Bruta		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe determinar la pertenencia de una palabra dentro de una gramática sin transformación.		

Tabla 58. RSF-11

RSF-12			
Título	Guardada pertenencia Fuerza Bruta		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe guardar la determinación de la pertenencia de una palabra dentro de una gramática sin transformación.		

Tabla 59. RSF-12

RSF-13			
Título	Árbol pertenencia Fuerza Bruta		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe guardar la el árbol generado por la pertenencia de una palabra dentro de una gramática sin transformación.		

Tabla 60. RSF-13

6.3.2. Identificación de Requisitos No Funcionales

La lista de estos requisitos de software no funcionales es la siguiente:

- **RSNF-01.** Sistema compatible Windows.
- **RSNF-02.** Escritura de operaciones.
- **RSNF-03.** Distinción ficheros.
- **RSNF-04.** Lectura cadena o cadenas.
- **RSNF-05.** Modificación ficheros.
- **RSNF-06.** Modificación ruta ficheros.
- **RSNF-07.** Idioma.
- **RSNF-08.** Mostrar información.
- **RSNF-09.** Accesibilidad de ficheros.
- **RSNF-10.** Instrucciones.
- **RSNF-11.** Detección NT y T no incluidos.
- **RSNF-12.** Detección comentarios.
- **RSNF-13.** Estructura ficheros.

Requisitos de operación

RSNF-01			
Título	Sistema compatible Windows		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe ser compatible con la consola de mandos del sistema operativo Windows.		

Tabla 61. RSNF-01

RSNF-02			
Título	Escritura de operaciones		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema escribirá los resultados de cada una de las operaciones que permite realizar en un fichero .txt con un nombre predefinido por el programador.		

Tabla 62. RSNF-02

RSNF-03			
Título	Distinción ficheros		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema hará distinción entre un fichero para la gramática y otro para la cadena o cadenas a analizar.		

Tabla 63. RSNF-03

RSNF-04			
Título	Lectura cadena o cadenas		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema permitirá al usuario analizar una sola cadena o varias cadenas a la vez.		

Tabla 64. RSNF-04

RSNF-05			
Título	Modificación ficheros		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	Los ficheros se podrán modificar sin cerrar el programa.		

Tabla 65. RSNF-05

RSNF-06			
Título	Modificación ruta ficheros		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	La ruta de los ficheros se podrá modificar sin cerrar el programa.		

Tabla 66. RSNF-06

Requisitos de comunicación

RSNF-07			
Título	Idioma		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El programa debe estar en castellano.		

Tabla 67. RSNF-07

RSNF-08			
Título	Mostrar información		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema debe mostrar claramente la información necesaria al usuario.		

Tabla 68. RSNF-08

Requisitos de recursos

RSNF-09			
Título	Accesibilidad de ficheros		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	Los ficheros .txt necesarios para realizar las operaciones deben estar disponibles para que el programa pueda acceder a ellos y leerlos.		

Tabla 69. RSNF-09

Requisitos de usabilidad

RSNF-10			
Título	Instrucciones		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema ofrecerá instrucciones en cada sección con el objetivo de orientar al usuario.		

Tabla 70. RSNF-10

Requisitos de comprobación

RSNF-11			
Título	Detección NT y T no incluidos		
Fuente	Analista		
Prioridad	Alta	**Necesidad**	Esencial
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema detectará en la gramática si las reglas se componen de los *Terminales* y *No terminales* especificados.		

Tabla 71. RSNF-11

RSNF-12			
Título	Detección comentarios		
Fuente	Analista		
Prioridad	Media	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema detectará comentarios en los ficheros de gramática y de las cadenas siempre que sigan el formato */*contenido del comentario*/*.		

Tabla 72. RSNF-12

RSNF-13			
Título	Estructura ficheros		
Fuente	Analista		
Prioridad	Media	**Necesidad**	Deseable
Verificabilidad	Alta	**Claridad**	Alta
Estabilidad	Durante toda la vida del sistema		
Descripción	El sistema comprobará que los ficheros están escritos siguiendo la estructura deseada por el programa.		

Tabla 73. RSNF-13

6.4. Casos de uso

En este apartado se especificaran los casos de uso. Los casos de uso son una descripción de los pasos o las actividades que deberán realizar los usuarios al llevar a cabo el uso de la aplicación.

Para exponer cada uno de los casos de uso, se seguirá este esquema.

CU-XX	
Título	
Actores	
Descripción	
Precondiciones	
Post condiciones	
Escenario	
Condiciones de fallo	

Tabla 74. Plantilla casos de uso

- **Identificador.** Caso de uso– Dígito#1 Dígito#2, esta nomenclatura permitirá identificar cada caso de uso de manera unívoca.
- **Título.** Nombre que se le da al caso de uso.
- **Actores:** los agentes externos que pueden intervenir en el caso de uso.
- **Descripción.** Breve explicación del proceso llevado a cabo por el agente en el caso de uso.
- **Pre-condiciones.** Condiciones que deben ser previamente ciertos para poder llevar a cabo el caso de uso.

- **Post-condiciones:** Condiciones que la correcta ejecución del caso de uso los hacen posibles.
- **Escenario.** Descripción esquemática de las fases que componen el caso de uso.
- **Condiciones de fallo.** Describe posibles errores y las respuestas del sistema ante los mismos.

Los casos de uso que se dan en la aplicación son los mostrados en la imagen *Ilustración 7. Casos de uso*.

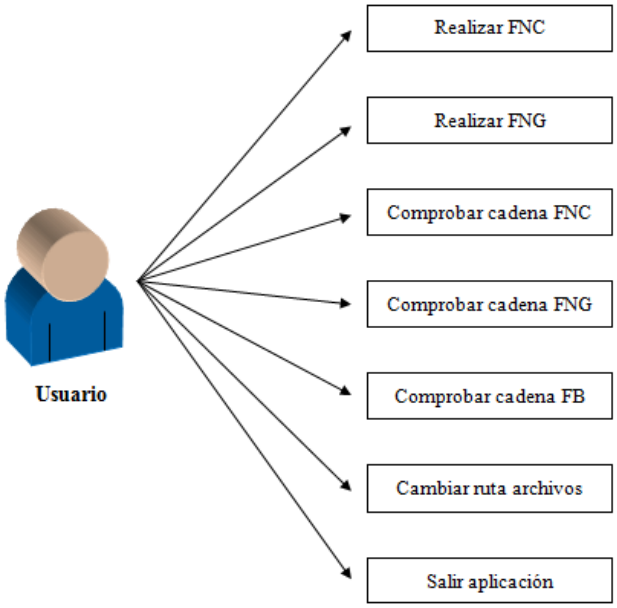


Ilustración 7. Casos de uso

La descripción de los casos de uso se realiza en las tablas mostradas posteriormente.

CU-01	
Título	Realizar FNC
Actores	Usuario
Descripción	El usuario, con la aplicación ejecutándose, escoge la opción que le permite transformar una gramática pasada por parámetro a otra en Forma Normal de Chomsky.
Precondiciones	• Tiene que estar la gramática en el fichero .txt con el formato adecuado. • No tiene que contener la gramática una regla que permita al axioma principal generar la palabra vacía.
Post condiciones	• La gramática está transformada. • Se han generado el fichero que guarda esas transformaciones.
Escenario	• El usuario se encuentra en el menú principal. • Pulsa la opción "Realizar FNC".
Condiciones de fallo	• La gramática contiene una regla que permita al axioma principal generar la palabra vacía. • La gramática no tenga el formato adecuado dentro del fichero. • No se introduzca bien la opción que marque la opción deseada.

Tabla 75. CU-01

CU-02	
Título	Realizar FNG
Actores	Usuario
Descripción	El usuario, con la aplicación ejecutándose, escoge la opción que le permite transformar una gramática pasada por parámetro a otra en Forma Normal de Greibach.
Precondiciones	• Tiene que estar la gramática en el fichero .txt con el formato adecuado para ese fichero. • No tiene que contener la gramática una regla que permita al axioma principal generar la palabra vacía.
Post condiciones	• La gramática está transformada. • Se han generado el fichero que guarda esas transformaciones.
Escenario	• El usuario se encuentra en el menú principal. • Pulsa la opción "Realizar FNG".
Condiciones de fallo	• La gramática contiene una regla que permita al axioma principal generar la palabra vacía. • La gramática no tenga el formato adecuado dentro del fichero. • No se introduzca bien la opción que marque la opción deseada.

Tabla 76. CU-02

CU-03	
Título	Comprobar cadena FNC
Actores	Usuario
Descripción	Al ejecutar la aplicación, el usuario selecciona la opción de comprobar una o más cadenas de entrada con una gramática en Forma Normal de Chomsky.
Precondiciones	• Tiene que estar la gramática en el fichero .txt con el formato adecuado para ese fichero. • La definición de la cadena o cadenas en el fichero .txt correspondiente se tiene que ajustar al formato especificado para ellas. • No tiene que contener la gramática una regla que permita al axioma principal generar la palabra vacía.
Post condiciones	• La gramática está transformada. • Se han generado el fichero que guarda esas transformaciones. • Se ha creado el fichero que contiene la pertenencia o no de esas palabras. • Se ha originado el fichero que contempla los árboles de decisión de las palabras aceptadas. • Se ha generado el fichero que contiene las tablas del algoritmo CYK que determina la pertenencia.
Escenario	• El usuario se encuentra en el menú principal. • Pulsa la opción “Comprobar cadena FNC”. • Determina si el fichero de las cadenas contiene o no cada palabra separada sus *Terminales* por un espacio.
Condiciones de fallo	• La gramática contiene una regla que permita al axioma principal generar la palabra vacía. • La gramática no tenga el formato adecuado dentro del fichero. • Las cadenas no tengan el formato adecuado dentro de su fichero. • No se introduzca bien la opción que marque la opción deseada.

Tabla 77. CU-03

CU-04	
Título	Comprobar cadena FNG
Actores	Usuario
Descripción	Al ejecutar la aplicación, el usuario selecciona la opción de comprobar una o más cadenas de entrada con una gramática en Forma Normal de Greibach.
Precondiciones	• Tiene que estar la gramática en el fichero .txt con el formato adecuado para ese fichero. • La definición de la cadena o cadenas en el fichero .txt correspondiente se tiene que ajustar al formato especificado para ellas. • No tiene que contener la gramática una regla que permita al axioma principal generar la palabra vacía.
Post condiciones	• La gramática está transformada. • Se han generado el fichero que guarda esas transformaciones. • Se ha creado el fichero que contiene la pertenencia o no de esas palabras. • Se ha originado el fichero que contempla los árboles de decisión de las palabras aceptadas.
Escenario	• El usuario se encuentra en el menú principal. • Pulsa la opción “Comprobar cadena FNG”. • Determina si el fichero de las cadenas contiene o no cada palabra separada sus *Terminales* por un espacio.
Condiciones de fallo	• La gramática contiene una regla que permita al axioma principal generar la palabra vacía. • La gramática no tenga el formato adecuado dentro del fichero. • Las cadenas no tengan el formato adecuado dentro de su fichero. • No se introduzca bien la opción que marque la opción deseada.

Tabla 78. CU-04

CU-05	
Título	Comprobar cadena FB
Actores	Usuario
Descripción	Al ejecutar la aplicación, el usuario selecciona la opción de comprobar una o más cadenas de entrada con la gramática sin modificar su formato.
Precondiciones	• Tiene que estar la gramática en el fichero .txt con el formato adecuado para ese fichero. • La definición de la cadena o cadenas en el fichero .txt correspondiente se tiene que ajustar al formato especificado para ellas. • No tiene que contener la gramática una regla que permita al axioma principal generar la palabra vacía.
Post condiciones	• Se han generado el fichero que guarda esas transformaciones. • Se ha creado el fichero que contiene la pertenencia o no de esas palabras. • Se ha originado el fichero que contempla los árboles de decisión de las palabras aceptadas.
Escenario	• El usuario se encuentra en el menú principal. • Pulsa la opción “Comprobar cadena FB”. • Determina si el fichero de las cadenas contiene o no cada palabra separada sus *Terminales* por un espacio.
Condiciones de fallo	• La gramática contiene una regla que permita al axioma principal generar la palabra vacía. • La gramática no tenga el formato adecuado dentro del fichero. • Las cadenas no tengan el formato adecuado dentro de su fichero. • No se introduzca bien la opción que marque la opción deseada.

Tabla 79. CU-05

CU-06	
Título	Cambiar ruta archivos
Actores	Usuario
Descripción	El usuario desea cambiar de fichero de análisis, y por tanto modificar la ruta de los ficheros que analiza el programa.
Precondiciones	Ninguna.
Post condiciones	• El programa utiliza nuevas rutas.

Escenario	• El usuario se encuentra en el menú principal.
	• Pulsa la opción "Cambiar ruta".
	• Escoge la opción que desea realizar:
	  ○ Cambiar ruta del fichero de la gramática.
	  ○ Cambiar ruta del fichero de las cadenas.
	  ○ Cambiar ruta del fichero de la gramática y del fichero de las cadenas.
	• Introduce la nueva ruta según le especifique el programa.

Condiciones de fallo	• No se introduzca bien la opción que marque el cambio de ruta. • El fichero especificado en la ruta no exista.

Tabla 80. CU-06

CU-07	
Título	Salir aplicación
Actores	Usuario
Descripción	El usuario ha terminado de utilizar la aplicación y desea cerrarla.
Precondiciones	Ninguna.
Post condiciones	• La aplicación se ha cerrado.
Escenario	• El usuario se encuentra en el menú principal. • Pulsa la opción "Salir de la aplicación".
Condiciones de fallo	• No se introduzca bien la opción que marque la terminación del programa.

Tabla 81. CU-07

6.5. Matriz de trazabilidad

Requisitos de usuario	Casos de uso						
Requisitos de capacidad	**CU-01**	**CU-02**	**CU-03**	**CU-04**	**CU-05**	**CU-06**	**CU-07**
RUC-01	X	X	X	X	X		
RUC-02			X	X	X		
RUC-03						X	
RUC-04						X	
RUC-05	X	X	X	X	X		
RUC-06			X	X	X		
RUC-07	X						
RUC-08			X				
RUC-09	X		X				
RUC-10			X				
RUC-11			X				
RUC-12			X				
RUC-13				X			
RUC-14		X		X			
RUC-15				X			
RUC-16				X			
RUC-17					X		
RUC-18					X		
RUC-19					X		
RUC-20							X
Requisitos de Restricción	**CU-01**	**CU-02**	**CU-03**	**CU-04**	**CU-05**	**CU-06**	**CU-07**
RUR-01	X	X	X	X	X		
RUR-02	X	X	X	X	X	X	X
RUR-03	X	X	X	X	X	X	X
RUR-04	X	X	X	X	X	X	X
RUR-05	X	X	X	X	X	X	X

Tabla 82. Requisitos usuario/Casos de uso

Requisitos de usuario	Requisitos de Software funcionales												
Requisitos de capacidad	**RSF -01**	**RSF -02**	**RSF -03**	**RSF -04**	**RSF -05**	**RSF -06**	**RSF -07**	**RSF -08**	**RSF -09**	**RSF -10**	**RSF -11**	**RSF -12**	**RSF -13**
RUC-01													
RUC-02													
RUC-03													
RUC-04													
RUC-05													
RUC-06													
RUC-07	X												
RUC-08			X										
RUC-09		X											
RUC-10					X								
RUC-11				X									
RUC-12													
RUC-13													
RUC-14						X	X						
RUC-15									X				
RUC-16								X					
RUC-17										X			
RUC-18													X
RUC-19												X	
RUC-20													
Requisitos de Restricción	**RSF -01**	**RSF -02**	**RSF -03**	**RSF -04**	**RSF -05**	**RSF -06**	**RSF -07**	**RSF -08**	**RSF -09**	**RSF -10**	**RSF -11**	**RSF -12**	**RSF -13**
RUR-01	X	X	X	X	X	X	X	X	X	X	X	X	X
RUR-02	X	X	X	X	X	X	X	X	X	X	X	X	X
RUR-03	X	X	X	X	X	X	X	X	X	X	X	X	X
RUR-04	X	X	X	X	X	X	X	X	X	X	X	X	X
RUR-05	X	X	X	X	X	X	X	X	X	X	X	X	X

Tabla 83. Requisitos usuario/ Requisitos Soft.Fun

Requisitos de usuario	Requisitos de Software no funcionales												
Requisitos de capacidad	**RSNF-01**	**RSNF-02**	**RSNF-03**	**RSNF-04**	**RSNF-05**	**RSNF-06**	**RSNF-07**	**RSNF-08**	**RSNF-09**	**RSNF-10**	**RSNF-11**	**RSNF-12**	**RSNF-13**
RUC-01	X		X				X	X		X			
RUC-02	X		X				X	X		X			
RUC-03	X		X			X	X	X	X	X			
RUC-04	X		X			X	X	X	X	X			
RUC-05	X	X	X		X		X	X		X			
RUC-06	X	X	X		X		X	X		X			
RUC-07	X	X					X	X	X		X	X	X
RUC-08	X	X		X			X	X	X			X	X
RUC-09	X	X					X	X	X		X	X	X
RUC-10	X	X		X			X	X	X		X	X	X
RUC-11	X	X		X			X	X	X		X	X	X
RUC-12	X	X		X			X	X	X			X	X
RUC-13	X	X		X			X	X	X			X	X
RUC-14	X	X					X	X	X		X	X	X
RUC-15	X	X		X			X	X	X		X	X	X
RUC-16	X	X		X			X	X	X		X	X	X
RUC-17	X	X		X			X	X	X			X	X
RUC-18	X	X		X			X	X	X		X	X	X
RUC-19	X	X		X			X	X	X		X	X	X
RUC-20	X	X					X	X					
Requisitos de Restricción	**RSNF-01**	**RSNF-02**	**RSNF-03**	**RSNF-04**	**RSNF-05**	**RSNF-06**	**RSNF-07**	**RSNF-08**	**RSNF-09**	**RSNF-10**	**RSNF-11**	**RSNF-12**	**RSNF-13**
RUR-01		X	X	X	X	X		X	X	X	X	X	X
RUR-02							X						
RUR-03	X												
RUR-04													

RUR-05													

Tabla 84. Requisitos usuario/ Requisitos Soft.NoFun

6.6. Identificación de los Usuarios Participantes y Finales

A continuación se especifican los participantes de este proyecto:

- **Jefe de proyecto**: es la persona que tiene la responsabilidad total del planeamiento y la ejecución de cualquier proyecto.
- **Diseñador:** responsable de especificar las características de la arquitectura del sistema y que servirá de base para el trabajo de los programadores.
- **Responsable de Pruebas:** es el encargado de realizar las pruebas del software necesarias para determinar si el sistema cumple satisfactoriamente con todos los requisitos.
- **Analista:** encargado de recopilar y especificar los requisitos tanto de usuario como del software a desarrollar.
- **Programador:** persona que escribe, depura y mantiene el código fuente de un programa informático.
- **Gestor documentación:** encargado de recopilar información y realizar la redacción del estado del arte.

La Universidad Carlos III de Madrid es la interesada en este proyecto, también denominada *stakeholder*, cuyos representantes D. Germán Gutiérrez Sánchez y D. Juan Manuel Alonso Weber establecerán los requisitos necesarios de la aplicación a desarrollar y a los que se les entrega del producto una vez finalizado.

Por último, los usuarios finales de la aplicación son los alumnos y profesores de *Teoría de autómatas y Lenguajes formales* y *Procesadores del lenguaje*.

7. Diseño del Sistema

En este apartado se recoge toda la información a cerca de la implementación del proyecto denominado *Naarpe*.

7.1. Definición de la Arquitectura del Sistema

La arquitectura que se ha seguido en este proyecto se designa *Arquitectura centralizada*. Denominada de esta manera puesto que se centra en un solo lugar físico, por ejemplo una sola CPU, y los usuarios al trabajar con ella reciben los resultados realizados en dicha interacción. Los sistemas en este modelo no se comunican con otros, tan solo existe la acción de dependencia del usuario con la máquina y viceversa.

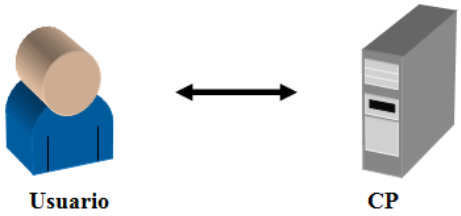


Ilustración 8. Arquitectura del sistema

Diseñamos este sistema por considerarlo el más adecuado para cumplir los requisitos fijados en el proyecto. Las razones que han llevado a tomar esta decisión se explican a continuación.

Otras opciones de diseño permiten el uso de un servidor. Este guarda toda la información relativa a los datos, tanto de usuario como lo relativo al problema a tratar.

La inicialización del programa implementado se realiza mediante un ejecutable. A través de este es posible acceder a todos los directorios no restringidos de la máquina que lo ejecuta. La opción alternativa de utilizar un servidor solo hubiese producido más lentitud a la hora de analizar los datos.

Un servidor proporciona un mecanismo de guardado de datos de forma segura y confidencial. Ninguno de los objetivos del proyecto tiene relación con los datos confidenciales de los usuarios. Esta utilidad del servidor no es necesaria.

No se ha diseñado un programa que necesite comunicación con otros usuarios. La interacción solo requiere un usuario y una CPU en la cual ejecutar el programa. Si necesitase comunicarse un sistema con otro, entonces es obligatorio el uso de servidor. Nuestro programa, al no requerir este punto, no obtiene beneficio del uso de servidor.

Por lo tanto, para la realización de *Naarpe* se optó por una arquitectura centralizada.

7.2. Definición del sistema

7.2.1. Programa

En este apartado explicaremos qué función realiza el fichero *Programa.java*. Dentro de este archivo se encuentra el *main* del programa junto con dos métodos necesarios para el funcionamiento de este.

leerTeclado

Método que analizar que lo introducido por teclado se haya leído correctamente. El parámetro en la llamada es el texto metido por el usuario y en caso de no haberse producido un error lo mostrará por pantalla.

comprobarCasosFuerzaBruta

Este método analiza si una gramática contiene la palabra vacía como parte del lenguaje generado por esta. Determinar este punto es necesario para posteriormente tener en cuenta qué reglas se podrían eliminar y cuáles no.

Inicialmente, el sistema recibe el *No terminal* introducido por parámetro (el axioma principal), lo guarda en un *ArrayList* y comprueba si genera alguna regla que tenga en su parte derecha únicamente un *No terminal*. De darse este caso, guarda el *No terminal* de la parte derecha en un *ArrayList* para analizarlo posteriormente.

Terminadas de analizar todas las producciones del axioma principal, elimina la posición 0 del *ArrayList*, en la cual está el *No terminal* principal, y comprueba si no se ha quedado vacío. De no ser así, toma el *No terminal* que se encuentre en la posición 0 (el que anteriormente estaba en la 1) y realiza el mismo procedimiento que anteriormente usó con el axioma principal.

Este proceso es iterativo, y termina si se da una de estas dos posibilidades: cuando al menos uno de los *No terminales* analizados genera la palabra vacía en una de sus reglas ó cuando ninguno de ellos la genere.

Encontrada una producción que genera lambda, el método imprimiría por pantalla "*No se puede eliminar de la gramatica la palabra vacia porque pertenece al lenguaje generado por esta. Esto no es admisible por este programa."* y devolvería true. En caso contrario, devolvería false (pudiéndose eliminar posteriormente, en caso de haber encontrado alguna, las producciones lambda).

Main

Este es el método principal de todo el programa. Desde él se llaman a las funciones principales de ejecución.

Cuando ejecutamos la aplicación, el *main* realiza una comprobación inicial mirando si se ha introducido la ruta de los ficheros en la llamada. Si el usuario ha comenzado el programa sin argumentos, muestra un mensaje pidiéndole que introduzca la dirección del fichero que contiene la gramática y del que engloba las cadenas a analizar.

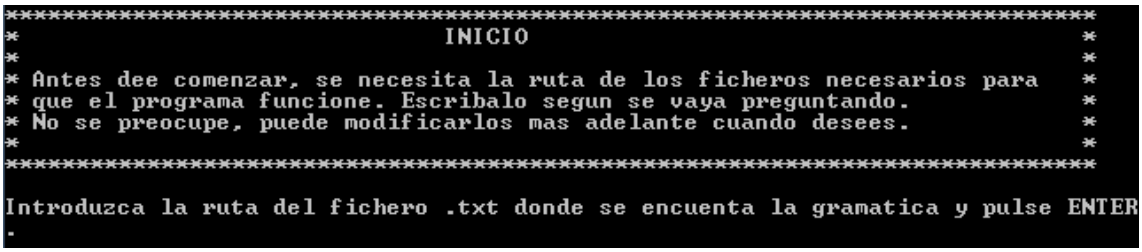


Ilustración 9. Ruta gramática

Ilustración 10. Ruta cadenas

El *main* comprueba, tanto introducidas por parámetro como por consola, que las rutas de los archivos realmente conduzcan a un fichero .txt. En caso de no existir alguna dirección de las fijadas, el sistema preguntará de nuevo al usuario hasta que la escriba correctamente. Al realizar la comprobación y obtener un resultado correcto, el sistema da paso al menú principal. Este consta de las opciones mostradas en la imagen *Ilustración 11*. *Menú de opciones*.

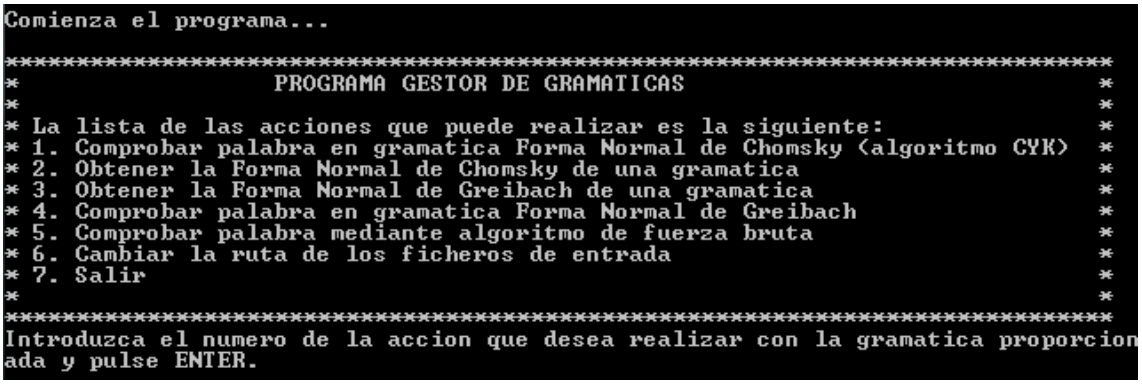


Ilustración 11. Menú de opciones

Todas las opciones, menos *Salir*, pueden ejecutarse tantas veces como se desee gracias a un *while* interno del programa.

Cuando se muestre el menú por pantalla, el usuario deberá introducir la opción deseada. Siempre que inserte una, el programa comprobará si lo escrito pertenece a una de las opciones disponibles. En caso de dar un valor inválido, el sistema imprimirá *OPCION INVALIDA*, mostrará el menú y volverá a pedir la introducción de la opción.

Para tratar los casos de introducción de un valor inválido, el sistema modifica el error por el valor 8 y ejecuta *default* interno. De esta manera el *while* de repetición sigue con valor true.

Cada una de las opciones posibles, que son las mostradas en el menú, realiza lo siguiente:

Acción 1

El usuario, al marcar esta opción, desea realizar la transformación de una gramática a formato de la Forma Normal de Chomsky y comprobar si las palabras guardadas, en el fichero de cadenas, pertenecen o no a dicha gramática transformada. Internamente, al introducir el usuario el valor 1 se ejecuta el *case 1* del *switch* principal.

Inicialmente, en esta acción el sistema pregunta al usuario si las palabras a analizar tienen los *Terminales* separados por un espacio. Las opciones de respuesta son sí o no, cada una de ellas mostrada con una opción numérica. Si la elección introducida es invalida, el programa pedirá de nuevo usuario que responda la cuestión planteada. Hasta que no se escriba una de las alternativas posibles, el sistema seguiría preguntando.

Respondida a la pregunta, el método llama a la función principal del fichero *Lectura_Fichero.java*, explicada en el apartado *7.2.10. Lectura gramática*. Cuando termine de ejecutarse, obtenemos la gramática que se ha escrito en el fichero correspondiente. Después, el sistema sacará de la gramática el axioma principal, el número de reglas, la longitud máxima de la parte derecha de las reglas y la definición para la palabra vacía.

Luego se ejecuta el método *comprobarCasosFuerzaBruta*. Si devuelve false, se lee el fichero que contiene las palabras según la opción señalada (si las cadenas están separadas o no por espacios) y con ello ya tendríamos todos los valores necesarios para transformar a Forma Normal de Chomsky.

Consecutivamente, el sistema llama al método *ejecucionFNC* del fichero *forma_normal_de_chomsky.java*. Con él se obtiene la gramática ya transformada en la Forma Normal de Chomsky. El apartado *7.2.2.Forma normal de Chomsky* explica el proceso que lleva a cabo para devolverla con el formato de la Forma Normal de Chomsky. Como el número de reglas y la longitud máxima de la parte derecha puede que sea diferente por transformación, volvemos a tomar los valores de estos. Además, imprimimos por pantalla cómo quedado la gramática resultante.

Posteriormente, el sistema inicializa los ficheros que guardan los resultados generados. A continuación, mediante un bucle *for*, separa cada una de las cadenas que están guardadas es un *Arraylist*. Esta separación se hace teniendo en cuenta si las palabras están separadas o no por

espacios. Con cada una de ellas, llama al método *ejecucionCYK* del fichero *Algoritmo_CYK.java*. La explicación de este método se realiza en el apartado *7.2.4. Algoritmo CYK*.

Por último, deja con valor nulo todas las variables inicializadas en el *case 1* y que puedan ser utilizadas también otros *case*. Así evita conflictos de valores en las sucesivas ejecuciones del menú principal.

Hemos explicado qué ocurre cuando el método *comprobarCasosFuerzaBruta* devuelve false. Si devolviese true, el sistema primero pregunta al usuario si desea que modifique automáticamente la gramática, de tal forma que elimine las no admisiones para poder continuar. Además, el programa avisará que estas modificaciones conllevan a generar una gramática que no contemple todas las palabras que sí lo hacía la original. El usuario puede contestar sí o no, cada una de ellas asignada a un valor numérico de respuesta.

Escogiendo sí, se realizarían los mismo pasos que cuando *comprobarCasosFuerzaBruta* devuelve false. Eligiendo no, el programa no puede continuar. Imprime *"El programa no puede continuar con esta opción. Modifique la gramática si desea realizar las operaciones de esta opcion."* y muestra el menú principal. En caso de marcar una opción inválida, el sistema muestra por pantalla *OPCION INVALIDA* y terminará la acción 1, dando paso a continuación al menú principal.

Acción 2

El usuario, al marcar esta opción, desea realizar la transformación de una gramática a formato de la Forma Normal de Chomsky. Internamente, al introducir el usuario el valor 2 se ejecuta el *case 2* del *switch* principal.

Inicialmente, el método llama a la función principal del fichero *Lectura_Fichero.java*, explicada en el apartado *7.2.10. Lectura gramática*. Cuando termine de ejecutarse, obtenemos la gramática que se ha escrito en el fichero correspondiente. Después, el sistema sacará de la gramática el axioma principal, el número de reglas, la longitud máxima de la parte derecha de las reglas y la definición para la palabra vacía.

Luego se ejecuta el método *comprobarCasosFuerzaBruta*. Si devuelve false, se lee el fichero que contiene las palabras según la opción señalada (si las cadenas están separadas o no por espacios) y con ello ya tendríamos todos los valores necesarios para transformar a Forma Normal de Chomsky.

Consecutivamente, el sistema llama al método *ejecucionFNC* del fichero *forma_normal_de_chomsky.java*. Con él se obtiene la gramática ya transformada en la Forma Normal de Chomsky. El apartado *7.2.2.Forma normal de Chomsky* explica el proceso que lleva

a cabo para devolverla con el formato de la Forma Normal de Chomsky. Como el número de reglas y la longitud máxima de la parte derecha puede que sea diferente por transformación, volvemos a tomar los valores de estos. Además, imprimimos por pantalla cómo quedado la gramática resultante.

Hemos explicado qué ocurre cuando el método *comprobarCasosFuerzaBruta* devuelve false. Si devolviese true, el sistema primero pregunta al usuario si desea que modifique automáticamente la gramática, de tal forma que elimine las no admisiones para poder continuar. Además, el programa avisará que estas modificaciones conllevan a generar una gramática que no contemple todas las palabras que sí lo hacía la original. El usuario puede contestar sí o no, cada una de ellas asignada a un valor numérico de respuesta.

Escogiendo sí, se realizarían los mismo pasos que cuando *comprobarCasosFuerzaBruta* devuelve false. Eligiendo no, el programa no puede continuar. Imprime *"El programa no puede continuar con esta opción. Modifique la gramática si desea realizar las operaciones de esta opcion."* y muestra el menú principal. En caso de marcar una opción inválida, el sistema muestra por pantalla *OPCION INVALIDA* y terminará la acción 2, dando paso a continuación al menú principal.

Por último, deja con valor nulo todas las variables inicializadas en el *case 2* y así evitar conflictos de valores en las sucesivas ejecuciones del menú principal.

Acción 3

El usuario, al marcar esta opción, desea realizar la transformación de una gramática a formato de la Forma Normal de Greibach. Internamente, al introducir el usuario el valor 3 se ejecuta el *case 3* del *switch* principal.

Inicialmente, el método llama a la función principal del fichero *Lectura_Fichero.java*, explicada en el apartado *7.2.10. Lectura gramática*. Cuando termine de ejecutarse, obtenemos la gramática que se ha escrito en el fichero correspondiente. Después, el sistema sacará de la gramática el axioma principal, el número de reglas, la longitud máxima de la parte derecha de las reglas y la definición para la palabra vacía.

Luego se ejecuta el método *comprobarCasosFuerzaBruta*. Si devuelve false, se lee el fichero que contiene las palabras según la opción señalada (si las cadenas están separadas o no por espacios) y con ello ya tendríamos todos los valores necesarios para transformar a Forma Normal de Greibach.

Consecutivamente, el sistema llama al método *ejecucionFNG* del fichero *forma_normal_de_greibach.java*. Con él se obtiene la gramática ya transformada en la Forma Normal de Greibach. El apartado *7.2.3.Forma normal de Greibach* explica el proceso que lleva

a cabo para devolverla con el formato de la Forma Normal de Greibach. Como el número de reglas y la longitud máxima de la parte derecha puede que sea diferente por transformación, volvemos a tomar los valores de estos. Además, imprimimos por pantalla cómo quedado la gramática resultante.

Hemos explicado qué ocurre cuando el método *comprobarCasosFuerzaBruta* devuelve false. Si devolviese true, el sistema primero pregunta al usuario si desea que modifique automáticamente la gramática, de tal forma que elimine las no admisiones para poder continuar. Además, el programa avisará que estas modificaciones conllevan a generar una gramática que no contemple todas las palabras que sí lo hacía la original. El usuario puede contestar sí o no, cada una de ellas asignada a un valor numérico de respuesta.

Escogiendo sí, se realizarían los mismo pasos que cuando *comprobarCasosFuerzaBruta* devuelve false. Eligiendo no, el programa no puede continuar. Imprime *"El programa no puede continuar con esta opción. Modifique la gramática si desea realizar las operaciones de esta opcion."* y muestra el menú principal. En caso de marcar una opción inválida, el sistema muestra por pantalla *OPCION INVALIDA* y terminará la acción 3, dando paso a continuación al menú principal.

Por último, deja con valor nulo todas las variables inicializadas en el *case 3* y que puedan ser utilizadas también otros *case*. Así evita conflictos de valores en las sucesivas ejecuciones del menú principal.

Acción 4

El usuario, al marcar esta opción, desea realizar la transformación de una gramática a formato de la Forma Normal de Greibach. Internamente, al introducir el usuario el valor 4 se ejecuta el *case 4* del *switch* principal.

Inicialmente, en esta acción el sistema pregunta al usuario si las palabras a analizar tienen los *Terminales* separados por un espacio. Las opciones de respuesta son sí o no, cada una de ellas mostrada con una opción numérica. Si la elección introducida es invalida, el programa pedirá de nuevo usuario que responda la cuestión planteada. Hasta que no se escriba una de las alternativas posibles, el sistema seguiría preguntando.

Respondida a la pregunta, el método llama a la función principal del fichero *Lectura_Fichero.java*, explicada en el apartado *7.2.10. Lectura gramática*. Cuando termine de ejecutarse, obtenemos la gramática que se ha escrito en el fichero correspondiente. Después, el sistema sacará de la gramática el axioma principal, el número de reglas, la longitud máxima de la parte derecha de las reglas y la definición para la palabra vacía.

Luego se ejecuta el método *comprobarCasosFuerzaBruta*. Si devuelve false, se lee el fichero que contiene las palabras según la opción señalada (si las cadenas están separadas o no por espacios) y con ello ya tendríamos todos los valores necesarios para transformar a Forma Normal de Greibach.

Consecutivamente, el sistema llama al método *ejecucionFNG* del fichero *forma_normal_de_greibach.java*. Con él se obtiene la gramática ya transformada en la Forma Normal de Greibach. El apartado *7.2.3.Forma normal de Greibach* explica el proceso que lleva a cabo para devolverla con el formato de la Forma Normal de Greibach. Como el número de reglas y la longitud máxima de la parte derecha puede que sea diferente por transformación, volvemos a tomar los valores de estos. Además, imprimimos por pantalla cómo quedado la gramática resultante.

Posteriormente, el sistema inicializa los ficheros que guardan los resultados generados. A continuación, mediante un bucle *for*, separa cada una de las cadenas que están guardadas es un *Arraylist*. Esta separación se hace teniendo en cuenta si las palabras están separadas o no por espacios. Con cada una de ellas, llama al método *mirarCadena_FNG* del fichero *Algoritmo_FNG.java*. La explicación de este método se realiza en el apartado *7.2.5.Algoritmo FNG*.

Hemos explicado qué ocurre cuando el método *comprobarCasosFuerzaBruta* devuelve false. Si devolviese true, el sistema primero pregunta al usuario si desea que modifique automáticamente la gramática, de tal forma que elimine las no admisiones para poder continuar. Además, el programa avisará que estas modificaciones conllevan a generar una gramática que no contemple todas las palabras que sí lo hacía la original. El usuario puede contestar sí o no, cada una de ellas asignada a un valor numérico de respuesta.

Escogiendo sí, se realizarían los mismo pasos que cuando *comprobarCasosFuerzaBruta* devuelve false. Eligiendo no, el programa no puede continuar. Imprime *"El programa no puede continuar con esta opción. Modifique la gramática si desea realizar las operaciones de esta opcion."* y muestra el menú principal. En caso de marcar una opción inválida, el sistema muestra por pantalla *OPCION INVALIDA* y terminará la acción 4, dando paso a continuación al menú principal.

Por último, deja con valor nulo todas las variables inicializadas en el *case 4* y así evitar conflictos de valores en las sucesivas ejecuciones del menú principal.

Acción 5

Al marcar esta opción, el usuario desea analizar las cadenas, introducidas en el fichero especificado para tal, con la gramática introducida sin modificar (exceptuando que sus reglas

con contemplen una forma deseada tal y como se expone en el apartado *Limpieza de gramáticas*). Al realizar esta acción, el sistema ejecutará el *case 5* del *switch* principal.

Inicialmente, en esta acción el sistema pregunta al usuario si las palabras a analizar tienen los *Terminales* separados por un espacio. Las opciones de respuesta son sí o no, cada una de ellas mostrada con una opción numérica. Si la elección introducida es invalida, el programa pedirá de nuevo usuario que responda la cuestión planteada. Hasta que no se escriba una de las alternativas posibles, el sistema seguiría preguntando.

Respondida a la pregunta, el método llama a la función principal del fichero *Lectura_Fichero.java*, explicada en el apartado *7.2.10. Lectura gramática*. Cuando termine de ejecutarse, obtenemos la gramática que se ha escrito en el fichero correspondiente. Después, el sistema sacará de la gramática el axioma principal, el número de reglas, la longitud máxima de la parte derecha de las reglas y la definición para la palabra vacía.

Luego se ejecuta el método *comprobarCasosFuerzaBruta*. Si devuelve false, se lee el fichero que contiene las palabras según la opción señalada (si las cadenas están separadas o no por espacios) y con ello ya tendríamos todos los valores necesarios para realizar el análisis.

Posteriormente, inicializamos los ficheros que guardan los resultados generados. Además, el sistema comprueba si hay reglas que cuyo formato es indeseado, las cuales hay que modificar para que los algoritmos funcionen. Cuando detecte al menos una, avisa al usuario mediante un mensaje que indica que eso no es admisible por el programa.

Terminado el análisis, el usuario decide si el programa elimina o no las inadmisiones para poder seguir ejecutando la opción escogida. Para ello se muestran dos opciones numéricas referentes a una de las decisiones a tomar. En caso de responder no, el programa mostrará *"El programa no puede continuar con esta opcion. Modifique la gramatica si desea realizar las operaciones de esta opcion."* y terminará la acción 5, dando paso a continuación a la muestra del menú principal. Respondiendo cualquier otro valor, el programa mostrará "*OPCION INVALIDA"* y terminará la acción 5, mostrando el menú principal.

La respuesta sí permitirá al programa proseguir con la ejecución. Llamará a la función *limpiezaGramatica* del fichero *Algoritmo_Fuerza_Bruta.java* (explicada en el apartado *7.2.6. Algoritmo Fuerza bruta*). Esta devolverá la gramática en condiciones óptimas para el análisis. Como el número de reglas y la longitud máxima de la parte derecha puede que sea diferente por transformación, volvemos a tomar los valores de estos.

A continuación, mediante un bucle, separamos cada una de las cadenas que hemos leído anteriormente (dependiendo de si están separadas o no por espacios se realizará de una forma

diferente) y con cada una se llama al método *mirarCadena_Fuerza_Bruta* explicado en el apartado *7.2.6. Algoritmo Fuerza bruta*.

Por último, deja con valor nulo todas las variables del *case 5* y que se puedan utilizar en otros *case*. Así evitamos conflictos de valores en las sucesivas ejecuciones del menú principal.

Puede que la gramática no contenga ningún mal formato, por lo que se realizaría lo mismo que hemos explicado, exceptuando el llamamiento a la función *limpiezaGramatica.*

Acción 6

El usuario, al marcar esta opción, desea cambiar la ruta de algún archivo. Puede modificar la dirección del fichero que contiene la gramática, el de las cadenas a analizar, los dos ó no modificar ninguno. Cuando determine realizar esta acción, internamente el sistema ejecutará el *case 6* del *switch* principal.

El sistema muestra por pantalla unas preguntas que reflejan las opciones anteriormente mencionadas, de las cuales el usuario debe escoger una. La elección introducida se comprueba, no teniendo que dar ni respuesta en blanco ni otro valor fuera de las opciones mostradas. El menú de esta acción se muestra por pantalla según se refleja la siguiente imagen:

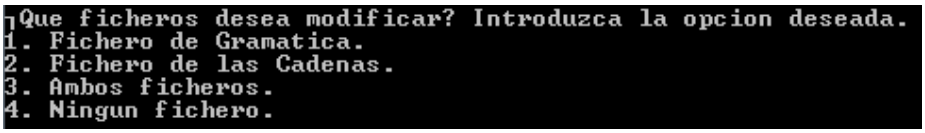


Ilustración 12. Menú cambio ruta

Si elige una opción inválida, el sistema mostrará *OPCION INVALIDA* y terminará la acción 6 dando paso a continuación a la muestra del menú principal.

En cambio, seleccionado una opción correcta daríamos paso a la ejecución de cada una de las acciones. Según la opción introducida, el usuario deberá introducir la nueva ruta del fichero de gramáticas, del fichero de las cadenas o ambas rutas. El sistema analizará el texto introducido por el usuario, asegurando que está bien escrita la dirección de un fichero .txt. De no ser así, se volverá a pedir que introduzca la ruta deseada tantas veces como sea necesario.

Acción 7

Si el usuario marca esta posibilidad, el ejecutable terminaría. El sistema imprimiría "... *Fin del programa*", pondría la variable de repetición del bucle del menú a false (*repeticion=false;*) y terminaría la ejecución completa del programa.

7.2.2. Forma normal de Chomsky

El fichero *forma_normal_de_chomsky.java* se detalla en este apartado. Realiza la transformación de la gramática a Forma Normal de Chomsky. Las funciones que componen este archivo se muestran en los subsiguientes apartados.

reglasInnecesarias

Este método elimina las reglas del tipo A → A, es decir, las reglas en las que un *No terminal* se genera a sí mismo. Estas producciones producen la ejecución de bucles infinitos cuando se analizan. En la llamada, se le introduce como parámetro la gramática a analizar.

Recorriendo cada una de las reglas, comprueba si el *No terminal* a la izquierda y el primero a la derecha son el mismo. Siendo cierta esta comprobación, comprueba la máxima longitud de las reglas. Cuando la longitud es mayor de dos, eso quiere decir que en la parte derecha hay más de un valor y comprueba si los siguientes valores de esta regla están vacíos. Esta confirmación es necesaria porque solo queremos reglas del tipo A → A. Si la respuesta es que no hay nada después, entonces se pone a *null* dicha regla y se aumenta la variable *eliminadas* para contemplar que una regla ha sido excluida.

Si la máxima longitud de las reglas es 2, eso quiere decir que ya no va a haber más valores después de la regla. Todas las reglas están compuestas por un solo valor (*terminal* o *No terminal*) en la parte derecha. En este caso, se pone a *null* dicha regla y se aumenta la variable *eliminadas* para contemplar que una regla ha sido excluida.

Después de analizar todas las reglas de la gramática, si se han realizado eliminaciones de reglas, se cambia la variable *modificado* a *true* y se llama a la función *arregloGramatica*.

Por último, devolvemos la gramática con las reglas innecesarias eliminadas.

arregloGramatica

Este método arregla la gramática de tal forma que las reglas eliminadas no queden en la gramática final. Para ello, se le pasa por parámetro la gramática y el número de reglas borradas.

Inicialmente, el sistema crea un nuevo *array* para guardar la gramática con las modificaciones. Los valores de filas y columnas son respectivamente el número de reglas menos las eliminadas y la máxima longitud de las reglas. El sistema copia todas las producciones de la gramática al nuevo *array*, exceptuando las que estén todas a *null* que no se transcribirían. Terminadas las copias, el método modifica la variable global *númeroReglas* poniendo como nuevo valor el anterior número menos las reglas eliminadas.

En ese momento de la ejecución la gramática sin las reglas eliminadas está guardada en el nuevo *array*. Para volverla a tener en el *array* que teníamos inicialmente, y que es el que manejan todas las funciones, hay que pasar el contenido de uno a otro.

Los parámetros del *array gramatica* han sido modificados, por tanto lo redenominamos. Una vez realizado este paso, el programa copia todo el contenido del nuevo *array* en la gramática inicial. Por último, devolvemos la gramática con todas las reglas en orden.

arregloGramaticaFinalReglas

El método modifica la gramática eliminando el valor *null*, como última posición, cuando lo tienen todas las reglas. El parámetro que recibe es la gramática a analizar.

Inicialmente se tiene un bucle *while*, el cual ejecuta el código de búsqueda tantas veces como sea necesario. Puede que todas las reglas tengan más de un *null* al final de las producciones, y con dicho *while* se puede realizar el análisis iterativamente.

Recorriendo todas las reglas, mira cuántas de ellas tienen el valor *null* al final. Si el número coincide con el número de reglas de la gramática, hay que eliminarlo de todas. En caso contrario, el bucle *while* terminaría y el método devolvería la gramática sin modificar.

Para eliminar este *núll* el programa crea un nuevo *array*. Este toma los valores de fila, como el número de reglas, y de columnas, como el valor de la máxima longitud de las reglas menos uno. Mediante un *for*, copiamos todos los valores de la gramática en el nuevo *array* menos el *null* final.

Terminada de hacer las copias, modifica la variable global *amplitudGramatica*, quitando uno al valor anterior, y se modifica el tamaño de la gramática inicial. Tenemos el nuevo *array* que ha guardado todas las producciones que deben mantenerse, por lo que copiamos todo el contenido del nuevo *array* en la gramática inicial (ya modificada los parámetros).

simbolosInaccesibles

Este método analiza si los *No terminales* que aparecen en las reglas de la gramática son accesibles desde el axioma principal ó si hay reglas que nunca serán llamadas. En caso de darse alguna producción inaccesible, quiere decir que no hay regla que genere algo a partir de él.

Para contemplar que todas las reglas han sido analizadas, generamos un *array* de dos posiciones como valor de columnas y tantas filas como número de reglas tenga la gramática. La primera columna representará si esa regla ha sido accedida mediante el análisis de otra, y la segunda simbolizará que se ha examinado toda la parte derecha de la producción a la que representa. Inicializaremos cada una de sus posiciones con el valor *No*.

Primero se analiza las reglas generadas por el axioma principal. Cuando se detecte una producción de este, ponemos, en el *array* que guarda si son accesibles, con valor *Si* la fila con valor del número de la regla y la columna 0.

Posteriormente, analiza si las reglas producidas por el axioma principal contienen algún *No terminal* en la parte derecha. En caso de darse, buscará las reglas generadas por dicho *No terminal.* Una vez encontradas, pondrá valor *Si* a fila con valor del número de la regla y la columna 0 del array, mostrando de este modo que sí que son accesibles.

Terminada de analizar la parte derecha de todas las reglas del axioma principal, pondrá valor *Si* a fila con valor del número de la regla y la columna 1 del *array*, mostrando de este modo que las reglas han sido analizadas completamente.

Después, se analizarán las reglas las cuales han sido accedidas pero no analizadas, es decir, aquellas que tengan, en el *array* que guarda si son accesibles, la posición 0 el valor *Si* y en la 1 el valor *No*. Sigue el mismo proceso de análisis que con las reglas producidas por el axioma principal. Analiza la parte derecha de cada una de las reglas y mirará si las reglas contienen algún *No terminal* en la parte derecha. Buscará las reglas que tengan en la parte izquierda ese *No terminal* y pondrá valor *Si* a fila con valor del número de la regla y la columna 0 del *array*. Terminada de analizar la parte derecha, pondrá valor *Si* a fila con valor del número de la regla y la columna 1 del *array*.

Cuando se termine el primer análisis de todas las reglas, puede que se hayan producido nuevos accesos a reglas que aun no estén analizadas, por lo que se volverá al principio y se seguirá analizando las reglas con la posición 0 el valor *Si* y en la 1 el valor *No.*

El análisis terminará cuando no se encuentren más reglas con esas condiciones. A continuación, el método busca si hay alguna que tenga el valor *No* en ambas posiciones; esto quiere decir que es inaccesible. Un bucle pondrá a *null* todos los valores de la regla o reglas inaccesibles y aumentará *noConectadas* tantas veces como producciones haya encontrado.

Por último, cambiará modificado=true, llamará a las funciones *arregloGramatica*, *eliminacionReglasIguales* y *arregloGramaticaFinalReglas* para dejar la gramática en perfectas condiciones (sin producciones solo con *null*, todas con *null* al final, etc.) y devolverá la gramática arreglada y libre de reglas inaccesibles.

reglasNoGenerativas

Método que elimina las reglas del tipo A -> Lambda. El parámetro que usa es a gramática a analizar.

Inicialmente se busca las reglas que generen la palabra vacía. Cuando encuentra una, mira si el *No terminal* que la genera tiene más reglas generadoras. La forma de comprobarlo es llamando a la función *mirarSiExisteReglaIgualTerminal*. De generar más reglas, miraríamos qué otras reglas tienen dicho *No terminal* en su parte derecha. Guardaríamos una copia de todas

estas pero eliminando dicho *No terminal* de la parte derecha. Las reglas copiadas se guardarían en un *arraylist*, separadas cada una de ellas por el símbolo ->.

En cambio, si no genera más reglas, miraríamos que otras reglas tienen dicho *No terminal* en su parte derecha y lo quitaríamos directamente. La regla quedaría de la misma manera, solo que el *No terminal* que genera la palabra vacía no aparecería en la producción.

Se tendrá en cuenta para este proceso anterior, si las reglas están compuestas en su parte derecha por uno o más componentes. En el caso de ser únicamente un componente, si el *No terminal* que genera la palabra vacía no produce más reglas, la regla que llame a este *No terminal* también se eliminaría entera.

Luego, tanto si se eliminan reglas como si se generan nuevas, se pondría *modificado=true* y llamará a *arregloGramatica*, *introduccionNuevasReglas*, *eliminacionReglasIguales* y *arregloGramaticaFinalReglas*. Dejarán la gramática sin las producciones lambda y entonces el método devolverá la gramática limpia.

mirarSiExisteReglaIgualTerminal

Este método determina si existe al menos una regla generada por el *No terminal* introducido por parámetro que sea distinta a la producción de la palabra vacía. El método recibe por parámetro la gramática a analizar y la posición, en la gramática, de la producción generadora de la palabra vacía.

Inicialmente recorre todas las reglas de la gramática para analizarlas, exceptuando la que se encuentre en posición pasada por parámetro. Para cada una de ellas, analiza el *No terminal* de la parte izquierda. Si este es igual que el que se ha pasado por parámetro, cambia *enocontrado* a valor true.

Finalmente, devuelve el valor que contenga *enocontrado*.

reglasDeRedenominacion

Esta función elimina las reglas del tipo *No terminal* para dar otro *No terminal*. Los valores introducidos por parámetro son la gramática a analizar y la lista de los *No terminales*.

Inicialmente, el método recorre todas las reglas de la gramática para obtener el primer valor de la parte derecha de la regla. Comprueba si ese valor es un *No terminal* y si la gramática tiene como longitud máxima de las reglas un valor mayor que dos. Siendo cierto, comprueba si después de ese *No terminal* no hay ningún otro valor en la producción.

Cumpliéndose que solo genere un *No terminal*, imprime en el fichero de salida la regla que va a ser eliminada. Después pone a *null* todos sus valores de la producción descartada y aumenta en uno el valor de la variable *eliminadas* para contemplar cuantas ha eliminado.

Posteriormente recorre toda las demás reglas buscando qué producciones genera el *No terminal* de la regla eliminada. Cuando encuentre al menos una, genera una nueva regla con la parte derecha de la producción encontrada pero en la parte izquierda incluye el *No terminal* de la regla eliminada. Estas nuevas producciones se guardarán en un *arraylists* separadas por el valor ->. Después, la variable *cuantasNuevasReglas* se aumenta cada vez que añadamos una nueva regla al *arraylists*.

Luego, si *eliminadas>0* y/o *cuantasNuevasReglas>0*, cambia *modificado* a *true*. Por último, llama a las funciones *arregloGramatica*, *introduccionNuevasReglas*, *eliminacionReglasIguales* y *arregloGramaticaFinalReglas* para posteriormente devolver la gramática arreglada y libre de reglas de redenominación.

introduccionNuevasReglas

Función que introduce en la gramática nuevas reglas que se han formado en otros métodos. Los parámetros pasados en la llamada son la gramática a analizar, el *arraylist* con las nuevas reglas y cuantas reglas contiene.

Para introducir las nuevas reglas, creamos un nuevo *array*. En este caso el número de columnas será la misma, ya que las producciones tienen la misma longitud, pero el número de filas será el valor obtenido al sumar el número de reglas más el número de nuevas reglas a introducir.

Posteriormente se copiarán todas las reglas de la gramática original en este nuevo *array*. Debajo de estas, añade las nuevas. Para incluirlas, lee cada uno de los componentes del *arraylist* de la siguiente manera: el primer valor que lee es el *No terminal* de la parte izquierda. Todo lo que lea hasta encontrar el valor -> será lo que se incluya en la parte derecha de la nueva regla. Cuando lea ->, quiere decir que una producción ha terminado lo cual se aumentará la fila del *array* nuevo para añadir la siguiente regla con el mismo proceso. Sigue haciendo este proceso iterativamente hasta que el *arraylist* quede completamente leído.

Por último, cambia el contenido *numeroReglas* por la suma del valor anterior más el número de nuevas reglas añadidas. Después, copia las producciones del nuevo *array* en la gramática original y la devuelve con las nuevas reglas ya introducidas.

redenominacion

Función que transforma la gramática en lo que se denomina Forma Normal de Chomsky. Los parámetros que tiene son la gramática a analizar, la lista de *terminales* y la de *No terminales*.

Inicialmente, recorre todas las partes derechas de las reglas comprobando qué componentes las forman. Hay diferencia en el análisis si a máxima longitud de una regla es tres o un valor más alto.

Si la máxima longitud es tres, quiere decir que como máximo las producciones tienen dos elementos en la parte derecha. Comprobando estos componentes, si posee dos valores en la parte derecha, estos deberán ser *No terminales*. En cambio, si algún factor, o los dos, es *Terminal* el sistema generará una nueva regla por cada uno de ellos.

Lo primero que realiza, en caso de detectar un *Terminal*, es generar un *No terminal* y que tenga como producción el *Terminal*. Una vez formada esta regla, la guarda para posteriormente introducirla en la gramática. A continuación cambia el *Terminal* de la regla que se estaba analizando por el *No terminal* generado. De esta manera la gramática tendrá la parte derecha de las reglas compuesta por dos *No terminales* o un solo *Terminal*.

Para la longitud de una regla mayor que tres el proceso es ligeramente diferente. En el caso de encontrarse con el análisis de reglas que tengan solo dos componentes en su parte derecha, el proceso es el mismo que el anterior.

Para reglas con más elementos se realizan instrucciones diferentes. El último elemento es examinado como el caso anterior del *Terminal*, es decir, si es obtiene un *Terminal* se creará una nueva regla que lo genere y será sustituido en la regla el *No terminal* generado.

El siguiente valor de la regla, si todavía no se ha llegado al primer elemento de la parte derecha, seguirá el mismo proceso. Tanto si se modifica, como si no, quedará contemplado con un *No terminal*. La Forma Normal de Chomsky tiene como una de las posibles producciones la generación de dos *Terminales*, por este motivo este *No terminal* lo juntaremos con el anterior y los generaremos en una nueva regla. Una vez guardada, el nuevo *No terminal* sustituirá a los otros dos en la regla analizada.

Este proceso seguirá hasta que solo quede analizar el primer valor de la parte derecha. Cuando llegue el caso, este será analizado de la misma manera que el último de la producción (el cual fue analizado el primero en el método).

Con todas las reglas analizadas, se incluyen en la gramática las nuevas producciones llamando al método *introduccionNuevasReglas* y además llama a *eliminacionReglasIguales* y *arregloGramaticaFinalReglas* para posteriormente devolver la gramática con la Forma Normal de Chomsky.

comprobarTerminales

Método que comprueba si el valor pasado por parámetro entra dentro de la denominación de los *Terminales*. Para determinar la pertenencia, compara el valor introducido

por parámetro con todos los que forman la lista de los *Terminales*. En el momento que encuentre una coincidencia, devuelve *encontrado=true*. Si no, devuelve *encontrado=false*.

imprimirPantalla

Método que imprime la gramática por pantalla. Para ello, la gramática es pasada por parámetro.

Por cada una de las filas que representan las reglas, se lee la columna 0 y se muestra. Después imprime ->, para reproducir la separación entre la parte derecha e izquierda de la producción. Luego, lee los siguientes valores de las columnas y los muestra, siempre que no contengan el valor *null*, separados por un espacio.

Cuando termine de leer una fila del *array*, continuará con las siguientes hasta recorrerlo entero.

imprimirFichero

Método que imprime la gramática en el fichero. Para ello, la gramática es pasada por parámetro.

Por cada una de las filas que representan las reglas, se lee la columna 0 y se guarda en el fichero. Después escribe ->, para reproducir la separación entre la parte derecha e izquierda de la producción. Luego, lee los siguientes valores de las columnas y los muestra, siempre que no contengan el valor *null*, separados por un espacio.

Cuando termine de leer una fila del *array*, continuará con las siguientes hasta recorrerlo entero.

eliminacionReglasIguales

Método que elimina dos reglas que se han generado y que son iguales. El parámetro que utiliza es la gramática a analizar.

Recorre todo el *array* de la gramática comparando fila por fila si todos los valores de las columnas son iguales a los de otra fila. En el momento en el que encuentre una, esta se elimina poniendo todos los valores a *null* y se aumenta *noConectadas* para contemplar todas las reglas que se excluyen.

Cuando estén todas analizadas, llama a *arregloGramatica* para dejar la gramática libre de las reglas eliminadas y a continuación devuelve la gramática.

ejecucionFNC

Método que ejecuta todos los métodos de esta clase, logrando transformar la gramática inicial a la Forma Normal de Chomsky.

Primero guarda el valor del axioma inicial, el número de reglas, el número de columnas de la matriz de la gramática, la definición de la palabra vacía e inicializa el fichero en el que se va a guardar los pasos realizados para la Forma Normal de Chomsky.

Luego llama a los métodos *reglasInnecesarias*, *reglasNoGenerativas*, *simbolosInaccesibles* y *reglasDeRedenominacion*. Si se ha realizado modificaciones de la gramática, contemplados en el *while(modificado==true)*, en estos métodos los volverá a llamar tantas veces como sean necesarios hasta que ya no se produzcan cambios. Terminados estos métodos, llama al método *redenominacion* para completar la transformación.

Por último, deja con valor nulo todas las variables usadas únicamente para este método, así evita conflictos de valores en las sucesivas ejecuciones, y devuelve la gramática completamente transformada.

7.2.3. Forma normal de Greibach

El fichero *forma_normal_de_greibach.java* se detalla en este apartado. Realiza la transformación de la gramática a Forma Normal de Greibach. Las funciones que componen este archivo se muestran en los subsiguientes apartados.

reglasInnecesarias

Este método ejecuta la misma acción que el explicado en el apartado *7.2.2. Forma normal de Chomsky* denominado *reglasInnecesarias*.

arregloGramatica

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el apartado *7.2.2. Forma normal de Chomsky* denominado *arregloGramatica*.

arregloGramaticaFinalReglas

En este apartado, el método a explicar es el mismo que el descrito en el apartado *7.2.2. Forma normal de Chomsky* denominado *arregloGramaticaFinalReglas*.

simbolosInaccesibles

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el apartado *7.2.2. Forma normal de Chomsky* denominado *simbolosInaccesibles*.

reglasNoGenerativas

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el apartado *7.2.2. Forma normal de Chomsky* denominado *reglasNoGenerativas*.

mirarSiExisteReglaIgualTerminal

Este método ejecuta las mismas operaciones que las explicadas en el apartado *7.2.2. Forma normal de Chomsky* denominado *mirarSiExisteReglaIgualTerminal*.

eliminacionReglasIguales

Este método ejecuta las mismas operaciones que las explicadas en el apartado *7.2.2. Forma normal de Chomsky* denominado *mirarSiExisteReglaIgualTerminal*.

reglasDeRedenominacion

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el apartado *7.2.2. Forma normal de Chomsky* denominado *reglasDeRedenominacion.*

introduccionNuevasReglas

En este apartado, el método a explicar es el mismo que el descrito en el apartado *7.2.2. Forma normal de Chomsky* denominado *introduccionNuevasReglas*.

imprimirPantalla

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el apartado *7.2.2. Forma normal de Chomsky* denominado *imprimirPantalla*.

imprimirFichero

Este método ejecuta las mismas operaciones que las explicadas en el apartado *7.2.2. Forma normal de Chomsky* denominado *imprimirFichero*.

eliminacionRecursividadIzquierdas

Este método elimina la recursividad a izquierdas de las reglas de la gramática. Los parámetros que posee son la lista de los *No terminales* y la gramática a analizar.

La recursividad a izquierdas se da cuando una regla contiene el mismo *No terminal* tanto en la parte izquierda como en el primer símbolo de la parte derecha. El proceso de análisis se realiza hasta que la gramática no contenga ninguna recursividad a izquierdas.

Inicialmente, el programa empieza el análisis por el final de la gramática y luego irá subiendo hasta la primera regla. Por cada una de las reglas, comprueba si tiene el mismo valor en la primera posición de la regla que el *No terminal* a la izquierda. Cuando se dé este caso, el sistema genera un nuevo *No terminal* para generar una regla que producirá toda la parte derecha de la regla analizada menos el *No terminal* que produce la recursividad. Esta nueva regla se guarda en un *arraylist* que contendrá las nuevas reglas generadas en este método.

Comprobamos qué otras reglas genera este *No terminal* recursivo. Por cada una que encuentre, realizará una copia de la regla, pero al final de cada una introducirá el nuevo *No terminal* generado.

Después, llamará a *arregloGramatica*, *introduccionNuevasReglasDeFNG* y *arregloGramaticaFinalReglas* para que introduzcan las reglas generadas y eliminen las reglas recursivas.

Puede que ese *No terminal* genere más reglas recursivas, y haya que realizar el mismo proceso descrito anteriormente.

Como hemos modificado las producciones de un determinado *No terminal*, hay que transformar las demás producciones que lo contengan. Cuando se encuentre una en la cual aparezca, el programa crea una copia de esta y donde aparece el *No terminal* recursivo introduce todas las posibles partes derechas que genere este. De este modo se tendrá modificada la regla tantas veces como reglas genere el *No terminal*. Al final, se elimina la producción original que contiene al *No terminal* cambiando todos los valores a *null*.

Llamaremos a *arregloGramatica*, *introduccionNuevasReglasDeFNG* y *arregloGramaticaFinalReglas* para que introduzcan las reglas generadas y eliminen las reglas recursivas analizadas de la gramática.

Por último, devolveros la gramática sin recursividad a izquierdas.

introduccionNuevasReglasDeFNG

Método que introduce en la gramática nuevas reglas que se han formado en los métodos que se usan para dejar la gramática con el formato FNG. Los parámetros pasados en la llamada son la gramática a analizar, el *arraylist* con las nuevas reglas y cuantas reglas contiene.

Para introducir las nuevas reglas, creamos un nuevo *array*. En este caso el número de columnas será la misma, ya que las producciones tienen la misma longitud, pero el número de filas será el valor obtenido al sumar el número de reglas más el número de nuevas reglas a introducir.

Posteriormente se copiarán todas las reglas de la gramática original en este nuevo *array*. Debajo de estas, añade las nuevas. Para incluirlas, lee cada uno de los componentes del *arraylist* de la siguiente manera: el primer valor que lee es el *No terminal* de la parte izquierda. Todo lo que lea hasta encontrar el valor -> será lo que se incluya en la parte derecha de la nueva regla. Cuando lea ->, quiere decir que una producción ha terminado lo cual se aumentará la fila del *array* nuevo para añadir la siguiente regla con el mismo proceso. Sigue haciendo este proceso iterativamente hasta que el *arraylist* quede completamente leído.

Por último, cambia el contenido *numeroReglas* por la suma del valor anterior más el número de nuevas reglas añadidas. Después, copia las producciones del nuevo *array* en la gramática original y la devuelve con las nuevas reglas ya introducidas.

conversionAFNG

Método que transforma la gramática dejando las reglas con un *Terminal* en la primera posición de la parte derecha y, en caso de contener más valores, una sucesión de *No terminales*.

Inicialmente comprobamos todas las reglas que contengan un *No terminal* en la primera posición. Cogemos ese *No terminal* y miramos cuáles son las reglas que produce. Por cada una de estas, realizaremos una copia de la regla inicial solo que ese *No terminal* será sustituido por la producción que genere. Una vez terminado las copias, la regla inicial se terminará.

Luego se llama a *arregloGramatica*, *introduccionNuevasReglasDeFNG* y *arregloGramaticaFinalReglas* para dejar la gramática con las nuevas reglas y sin las eliminadas. Todo este proceso se realizará hasta que los primeros valores de todas las reglas sean *Terminales*.

Después, analizamos toda la parte derecha de las reglas menos el primer símbolo (que ha quedado modificado a un valor de *Terminal*). Si la regla tiene longitud mayor a uno en la parte derecha, esta debe producir una sucesión de *No terminales* después del primer *Terminal*. En el momento en el que se encuentre un *Terminal*, se genera un nuevo *No terminal* que lo sustituirá. Este, mediante la creación de una nueva regla, producirá el *Terminal* modificado. Posteriormente se llama a *arregloGramatica*, *introduccionNuevasReglasDeFNG* y *arregloGramaticaFinalReglas* para dejar la gramática con las nuevas reglas y sin las eliminadas.

Por último, devuelve la gramática completamente transformada.

comprobarTerminales

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el apartado *7.2.2. Forma normal de Chomsky* denominado *comprobarTerminales.*

ordenacionDeReglasParaFNG

Este método ordena las reglas de tal forma que puedan ser posteriormente analizadas y transformadas a FNG.

Al inicio de la gramática deben permanecer las reglas que pertenezcan al axioma principal, por lo que en un *arraylist*, que contenga los *No terminales* analizados, incluimos el axioma principal. Después, se analizan todas las partes derechas de las reglas que genere el axioma. En cuanto se encuentre un *No terminal*, este se añadirá al *arraylist* comprobando que anteriormente no se haya introducido ese valor.

Cuando terminemos de examinar el axioma, sacaremos el siguiente valor en el arraylist de los *No terminales*. Comprobaremos la parte derecha de sus reglas y si hay un *No terminal* que no esté en el listado, lo introducimos *arraylist*. Realizaríamos el mismo proceso hasta terminar de analizar todos los componentes del *arraylist*.

Terminado la comparación, generamos un *array* con el mismo tamaño que la gramática. En él iremos introduciendo las reglas según el orden de los *No terminales* que estén en el

arraylist. Inicialmente estarán todas las del axioma principal. Después, saca el siguiente *No terminal* y guarda todas las producciones que este genere en el nuevo *array*. A continuación, realiza el mismo proceso para todos los *No terminales* hasta completar la búsqueda por el *arraylist.*

Por último copiaremos el nuevo *array*, que contiene las reglas ordenadas, al de la gramática y la devolveremos.

ejecucionFNG

Método que ejecuta todos los métodos de esta clase para poder realizar las operaciones necesarias y transformar la gramática inicial a una que tenga el formato de la Forma Normal de Greibach.

Primero guarda el valor del axioma inicial, el número de reglas, el número de columnas de la matriz de la gramática, la definición de la palabra vacía e inicializa el fichero en el que se va a guardar los pasos realizados para la Forma Normal de Greibach.

Luego llama a los métodos *reglasInnecesarias*, *reglasNoGenerativas*, *simbolosInaccesibles* y *reglasDeRedenominacion*. Si se ha realizado modificaciones de la gramática, contemplados en el *while(modificado==true)*, volverá a llamar a las funciones tantas veces como sean necesarios hasta que ya no se produzcan cambios.

Cuando *modificado=false*, llama a los métodos *ordenacionDeReglasParaFNG*, *eliminacionRecursividadIzquierdas*, *eliminacionReglasIguales*, *conversionAFNG* y *eliminacionReglasIguales* para completar la transformación.

Por último, deja con valor nulo todas las variables usadas únicamente para este método, para evitar conflictos de valores en las sucesivas ejecuciones, y devuelve la gramática completamente transformada.

7.2.4. Algoritmo CYK

El archivo *Algoritmo_CYK.java* se detalla en este apartado. Las funciones que componen este archivo se muestran en los subsiguientes apartados.

RellenarTablaCYK

Método que llama a las funciones que introducen contenido en la tabla del CYK. Los parámetros que necesita son la cadena a analizar y la gramática.

Primero, llama a *rellenarPrimeraColumna* para que introduzca los valores iniciales. Terminado este método, comprueba cuánto es la longitud de la cadena.

Si es mayor o igual que 2, entonces hay al menos una columna más que rellenar. Para hacerlo llama al método *rellenarSegundaColumna*. En cambio, si solo hay un *Terminal* en la cadena, el método devolvería la tabla tal y como la ha completado *rellenarPrimeraColumna*.

rellenarSegundaColumna devuelve la tabla del CYK con la segunda columna completa. Después, el sistema comprueba si la longitud de la cadena es mayor o igual a 3. Esto quiere decir que queda al menos una columna más por rellenar, por lo que se llamaría al método *rellenarSiguientesColumnas*. En cambio, si solo contiene dos *Terminales*, el método devolvería la tabla tal y como la ha completado *rellenarSegundaColumna*.

Finamente, el método devuelve la tabla según la hayan completado los distintos métodos.

rellenarPrimeraColumna

Método que completa la primera columna de la tabla. Los parámetros que recibe son la cadena a analizar y la gramática.

El método obtiene el primer *Terminal* de la cadena a analizar. Después, comprueba la primera posición de todas las partes derechas de las reglas a ver si coinciden con ese *Terminal*. En el caso de dar cierto, mira a ver si en la segunda posición tiene el valor *null*. Esto querrá decir que la regla genera tan solo un *Terminal*. A continuación, guarda el *No terminal* de la parte izquierda de la regla en el *String* que guarda todas las coincidencias. Este mismo proceso es realizado para todas las reglas de la gramática.

Luego, todas las coincidencias obtenidas se guardan en la posición correspondiente en la tabla. Terminado de rellenar el valor en la tabla, saca el siguiente *Terminal* de la cadena y realiza las comprobaciones en la gramática. Este proceso dura hasta analizar el último valor de la cadena y completar la posición correspondiente en la tabla.

Finalmente, el método devolverá la tabla con la primera columna completa.

rellenarSegundaColumna

La función obtiene todas las combinaciones posibles cada una de las posiciones de la segunda columna de la tabla y al final deja sólo las que son válidas según la gramática.

Inicialmente se coloca en la primera posición de la columna. Toma los valores de una columna menos pero la misma posición de fila y los la columna menos y una fila más, es decir, si está en (i=0, j=1) tomaría los de la (i=0, j=0) y (i=1, j=0). Guardaría en la posición de la tabla todas las combinaciones posibles, dos a dos, de los valores que hay en las coordenadas analizadas. Cada una de las combinaciones se separará por "->". Esta metodología se realizaría para todas las filas de la segunda columna.

Terminadas las combinaciones, si se ha dado más de una llama al método *buscarLaReglaDelContenido*, el cual deja las posiciones de la segunda columna de la tabla con los *No terminales* de la parte izquierda de las reglas que pueden generar las combinaciones encontradas.

Por último se llama a *liempiezaTabla*, la cual deja las posiciones de la tabla sin repetición de *No terminales*, y devuelve la tabla con la segunda columna completamente analizada.

rellenarSiguientesColumnas

Este método obtiene todos los valores para cada una de las posiciones de la tercera columna en adelante.

Inicialmente se completa la fila tres. Toma las posiciones de X_{ik} y $X_{i+k,j-k}$. Para (i=1, j=3) hay que hacer el mismo análisis con (i=1, j=1) y (i=2, j=2) y después con (i=1, j=2) y (i=3, j=1). Se irán aumentando o disminuyendo los valores de ***j*** e ***i*** tal y como se explicó en el apartado 2.3. *Algoritmo Cocke-Younger-Kasami*. Guardaría en la posición de la tabla todas las combinaciones posibles, dos a dos, de los valores que hay en las coordenadas analizadas. Cada una de las combinaciones se separará por "->". Esta metodología se realizaría para todas las filas de la tercera columna.

Terminadas las combinaciones, si se ha dado más de una llama al método *buscarLaReglaDelContenido*, el cual deja las posiciones de la segunda columna de la tabla con los *No terminales* de la parte izquierda de las reglas que pueden generar las combinaciones encontradas. Después, llama a *liempiezaTabla*, la cual deja las posiciones de la tabla sin repetición de *No terminales*.

Terminada de analizar la tercera columna, miraría si la tabla contiene más y las analizaría cada una de ellas de la misma forma.

Por último, el método devuelve la tabla con las columnas finales completamente analizadas.

buscarLaReglaDelContenido

Método que lee el contenido de una columna de la tabla y cambia todas las combinaciones que se encuentren en cada posición por el *No terminal* de la gramática que las genere.

Inicialmente toma el primer valor de la columna de la tabla a analizar. Lee palabra por palabra del valor que haya obtenido. Cuando lea un -> para y toma todo lo anterior como una posible combinación.

A continuación, mira si se da esa combinación en alguna de las partes derechas de las reglas de la gramática. Si la encuentra, cambia la combinación por el *No terminal* que genera la regla. Cuando no la encuentre, entonces elimina la combinación de esa posición de la tabla.

Este proceso lo realiza para todas las posiciones de la columna. Cuando termine, el método devuelve la tabla con la columna completa de los *No terminales* que es posible de obtener en cada caso.

liempiezaTabla

Método que elimina los *No terminales* repetidos de cada una de las posiciones de la tabla.

Inicialmente toma el primer valor de la columna uno de la tabla a analizar. Lee palabra por palabra del valor que haya obtenido. Cuando obtiene un valor, mira si está guardado en el *arrayList guardadoInicioReglas*. Si está contenido en este no haría nada. En caso contrario guardaría el valor adquirido.

Cuando termine de analizar esa posición, elimina todo el contenido y guarda lo que contenga el *arrayList guardadoInicioReglas*. A continuación borra toda la información de *guardadoInicioReglas*

Terminada esa posición, analiza las siguientes con el mismo proceso. Cuando termine con esa columna, también analizará las demás como hasta ahora.

Cuando termine, el método devuelve la tabla con la columna completa de los *No terminales* que es posible de obtener en cada caso.

generarCadenaArbolCYK

Método que genera una cadena que contiene todas las reglas que generan la cadena a analizar, separadas por *[* y *]* para determinar los niveles de estas producciones dentro del árbol de decisión.

Inicialmente comprueba si la cadena sólo tiene un *Terminal* que la componga. Si es así, guarda dentro del *String cadenaArbol* el valor *[*, seguido de la primera regla que genere ese *Terminal* más el carácter *]*.

Para cadenas con más de un Terminal, realizamos la búsqueda de los valores de *String cadenaArbol* de la misma manera que se rellena la tabla. Conseguimos el valor de dos posiciones de la tabla según la *j* y la *i* introducidas por parámetro. A continuación, verificamos, mediante *mirarEnGramatica*, si hay alguna regla que genere la combinación de dos *No terminales*, siendo cada uno de ellos tomado de una de las dos coordenadas obtenidas. Si no se

puede dar, se comprueba más combinaciones de *No terminales* que contengan las dos posiciones de la tabla.

Si no se ha encontrado regla alguna al terminar todas las combinaciones posibles, llama recursivamente al método pero modificando los valores de la *j* y la *i*. Por el contrario, si se ha encontrado regla que lo genere entonces llama a *numeroDeLaReglaEncontrada* y obtiene la posición de esta, la guarda en *String cadenaArbol* con valor *[*, seguido del número de la regla más el carácter *]*. Después llama recursivamente al método.

De esta manera, va obteniendo todos los valores de las posibles combinaciones hasta que genere todo el proceso que se refleje en el árbol de decisión.

mirarEnGramatica

Método que devuelve la posición de la regla generada por un *No terminal*, pasado por parámetro, que en su parte derecha contenga el valor de dos *No terminales* juntos, también pasados por parámetro. A parte de estos dos, además se introduce por parámetro la gramática a analizar.

El método recorre toda la gramática mirando si la parte derecha coincide con el *No terminal*, denominado nodo, pasado por parámetro. En el momento en el que encuentre una, comprueba si su parte derecha está compuesta por los *No terminales hoja1* y *hoja2* pasados por parámetro. Si llega a encontrar la regla que cumpla con lo establecido, el método devuelve *true*. En caso contrario, devuelve *false*.

numeroDeLaReglaEncontrada

Método que devuelve la posición de la regla generada por un *No terminal*, pasado por parámetro, que en su parte derecha contenga el valor de dos *No terminales* juntos, también pasados por parámetro. A parte de estos dos, además se introduce por parámetro la gramática a analizar.

El método recorre toda la gramática mirando si la parte derecha coincide con el *No terminal*, denominado nodo, pasado por parámetro. En el momento en el que encuentre una, comprueba si su parte derecha está compuesta por los *No terminales hoja1* y *hoja2* pasados por parámetro. Si llega a encontrar la regla que cumpla con lo establecido, el método devuelve la posición en la que se encuentre dentro de la gramática.

numeroDeLaReglaTerminal

Método que devuelve la posición de la regla generada por un *No terminal*, pasado por parámetro, que en su parte derecha contenga solo un valor. A parte este *No terminal*, además se introduce por parámetro la gramática a analizar.

El método recorre toda la gramática mirando si la parte derecha coincide con el *No terminal*, denominado nodo, pasado por parámetro. En el momento en el que encuentre una, el método devuelve la posición.

sePuedeGenerarPalabra

Método que determina si la cadena puede ser generada por la gramática. Tan solo tiene por parámetro en la llamada la tabla completa obtenida en el análisis de la cadena.

Únicamente la función analiza la primera posición de la última columna de la tabla. Obtiene el valor de esa colocación y comprueba si este contiene el *No terminal* del axioma principal. De ser cierto, entonces el método devuelve *true*. De lo contrario, devuelve *false*.

ejecucionCYK

Método que ejecuta todos los métodos de esta clase para poder realizar las operaciones necesarias y analizar la gramática inicial con las cadenas introducidas.

Primero guarda el valor del axioma inicial, el número de reglas y el número de columnas de la matriz de la gramática.

Luego llama al método rellenarTablaCYK para obtener la tabla con los valores que genera el algoritmo CYK cada vez que comprueba una cadena. Después llama a *sePuedeGenerarPalabra* y según el valor que devuelva sabremos si la cadena pertenece o no a la gramática. En caso de pertenecer, se guardará en el fichero que contiene los resultados "*ACEPTADA cadena:* " seguido de la cadena analizada. A continuación, en el fichero que guarda los datos del árbol de decisión se escribe "*Arbol del CYK:\n----------\nDe la cadena*", seguido de la cadena analizada, y se llama al método *ejecucionArbolYK*.

Cuando sea una cadena que no es aceptada, se guardará en el fichero que contiene los resultados "*ACEPTADA cadena:* " seguido de la cadena analizada. A continuación, en el fichero que guarda los datos del árbol de decisión se escribe "*Arbol del CYK:\n----------\nDe la cadena*", seguido de la cadena analizada, y debajo se guarda "*No hay árbol posible*".

Por último, deja con valor nulo todas las variables usadas únicamente para este método, para evitar conflictos de valores en las sucesivas ejecuciones.

7.2.5. Algoritmo FNG

El fichero *Algoritmo_FNG.java* se detalla en este apartado. Este método determina la pertenencia de una palabra en una gramática en FNG. Las funciones que componen este archivo se muestran en los subsiguientes apartados.

metodoFuerzaBruta

Método que analiza si la cadena que se pasa por parámetro se puede dar con la gramática proporcionada. Los parámetros a introducir en su llamada son: variable que indica el nivel de profundidad en el que estamos buscando, la gramática a analizar, la lista de los *No terminales*, parte de la cadena que se está analizando en cada llamamiento al método y la cadena a comparar.

Inicialmente comprobamos que la parte de la cadena que se está analizando no sea mayor que la cadena a comparar. Si no es mayor, llama al método *buscaPrimerNT*, explicado en el apartado siguiente, que busca el primer *No terminal* de la parte de la cadena a comparar.

Si no hemos encontrado ningún *No terminal*, ejecutamos el método *iguales*, explicado en el apartado *Iguales*, y comprobamos si la parte de la cadena que se está analizando es igual a parte de la cadena a comparar. Cuando es cierto, igualamos encontrado a *true*, y cuando no, a *false*.

Cuando sí que se encuentre un *No terminal*, ejecuta el método *matchParcial*, explicado en el apartado *matchParcial*, para comprobar si los *Terminales* que se han leído hasta el momento coinciden con la cadena. Cuando coincidan, igualamos encontrado a *true*, y cuando no, a *false*.

Después, sacamos el primer *No terminal* por la izquierda que exista en la parte de la cadena a comparar. Comprobamos si hay alguna regla que genere ese *No terminal* y que nos devuelva true el llamamiento recursivo al método. Esto querrá decir que hemos encontrado el camino correcto para generar la cadena. De no ser así, devolvería el método *metodoFuerzaBruta encontrado=false*.

buscaPrimerNT

Función que busca el primer *No terminal* dentro de la parte de la cadena a analizar. Los parámetros que tiene son la lista de los *No terminales* y la parte de la cadena que se está analizando en cada llamamiento al método.

El método consiste en un bucle que analiza cada posición del *arrayList* introducido por parámetro, comprobando para cada una de ellas si se trata de un *Terminal* o un *No terminal*. Cuando encuentre un *No terminal*, el bucle para y devuelve la posición de la cadena en la que se ha encontrado. En caso de no existir, devolvería el valor -1.

Iguales

Función que analiza si la parte de la cadena a analizar es igual que la cadena a comparar. Los parámetros que posee son la parte de la cadena que se está analizando en cada llamamiento al método y la cadena a comparar.

Inicialmente comprueba que ambas sean del mismo tamaño. Cuando no sea así es que no son iguales y devuelve false. Cuando sí lo sean, compara posición a posición que sean idénticas. Devuelve *true* cuando lo son y *false* cuando no.

matchParcial

Método que analiza si la parte de la cadena a analizar, hasta el primer *No terminal* encontrado, es igual que la cadena hasta ese mismo punto.

Si el *No terminal* tiene una posición mayor que la longitud de la cadena, el resultado es falso, es decir, no son iguales hasta ese *No terminal*. Cuando la posición esté dentro del rango de valores, cuyo tope es el tamaño de la cadena, compara posición a posición que sean idénticas. Devuelve *true* cuando lo son y *false* cuando no.

aplicaRegla

Función que sustituye el *No terminal* indicado, por la parte derecha de una regla que genere. Los parámetros que necesita para la ejecución son la gramática a analizar, la cadena a comparar, la posición en la que se encuentra el *No terminal* a sustituir y el número de la regla que se va a necesitar su parte derecha.

Lo primero que hay que realizar es una copia de la parte de la cadena que se está analizando hasta la posición donde se encuentre el *No terminal* a sustituir. Una vez la tengamos, introducimos la parte derecha de la regla donde antes se encontraba el *No terminal*.

Después dejaremos la parte restante de la cadena tal y como estaba. Para ello copiamos todo lo que iba después del *No terminal*. Por último, devolveremos la parte de la cadena a analizar con el *No terminal* modificado.

mirarCadena_Fuerza_Bruta

Método que ejecuta todos los métodos de esta clase para poder realizar las operaciones necesarias y analizar la gramática inicial con las cadenas introducidas.

Primero guarda el valor del axioma inicial, el número de reglas, el número de columnas de la matriz de la gramática y la definición de la palabra vacía.

Luego llama al método *metodoFuerzaBruta* y según el valor que devuelva sabremos si la cadena pertenece o no a la gramática. En caso de pertenecer, se guardará en el fichero que contiene los resultados "*ACEPTADA cadena:* " seguido de la cadena analizada. A continuación, en el fichero que guarda los datos del árbol de decisión se escribe "*Arbol Fuerza Bruta:\n----------\nDe la cadena*", seguido de la cadena analizada, y se llama al método *arbolFNG* explicado en el apartado *7.2.8. Árbol FNG*.

Cuando sea una cadena que no es aceptada, se guardará en el fichero que contiene los resultados "*ACEPTADA cadena:* " seguido de la cadena analizada. A continuación, en el fichero que guarda los datos del árbol de decisión se escribe "*Arbol Fuerza Bruta:\n----------\nDe la cadena*", seguido de la cadena analizada, y debajo se guarda "*No hay árbol posible*".

Por último, deja con valor nulo todas las variables usadas únicamente para este método, para evitar conflictos de valores en las sucesivas ejecuciones, y devuelve la gramática completamente transformada.

7.2.6. Algoritmo Fuerza bruta

El fichero *Algoritmo_Fuerza_Bruta.java* se detalla en este apartado. Las funciones que componen este archivo se muestran en los subsiguientes apartados.

reglasInnecesarias

Este método ejecuta la misma acción que el explicado en el apartado *7.2.2. Forma normal de Chomsky* denominado *reglasInnecesarias*.

simbolosInaccesibles

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el capítulo *7.2.2. Forma normal de Chomsky* denominado *simbolosInaccesibles*.

reglasNoGenerativas

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el apartado *7.2.2. Forma normal de Chomsky* denominado *reglasNoGenerativas*.

arregloGramatica

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el capítulo *7.2.2. Forma normal de Chomsky* denominado *arregloGramatica*.

introduccionNuevasReglas

En este apartado, el método a explicar es el mismo que el descrito en el apartado *7.2.2. Forma normal de Chomsky* denominado *introduccionNuevasReglas*.

arregloGramaticaFinalReglas

En este apartado, el método a explicar es el mismo que el descrito en el capítulo *7.2.2. Forma normal de Chomsky* denominado *arregloGramaticaFinalReglas*.

eliminacionReglasIguales

Este método ejecuta la misma acción que el explicado en el apartado *7.2.2. Forma normal de Chomsky* denominado *eliminacionReglasIguales.*

mirarSiExisteReglaIgualTerminal

Este método ejecuta las mismas operaciones que las explicadas en el capítulo *7.2.2. Forma normal de Chomsky* denominado *mirarSiExisteReglaIgualTerminal*.

eliminacionRecursividadIzquierdas

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el apartado *7.2.3. Forma normal de Greibach* denominado *eliminacionRecursividadIzquierdas*.

introduccionNuevasReglas2

Método que introduce en la gramática nuevas reglas que se han formado en otros métodos. Los parámetros pasados en la llamada son la gramática a analizar, el *arraylist* con las nuevas reglas y cuantas reglas contiene.

Para introducir las nuevas reglas, creamos un nuevo *array*. En este caso el número de columnas será la misma, ya que las producciones tienen la misma longitud, pero el número de filas será el valor obtenido al sumar el número de reglas más el número de nuevas reglas a introducir.

Posteriormente se copiarán todas las reglas de la gramática original en este nuevo *array*. Debajo de estas, añade las nuevas. Para incluirlas, lee cada uno de los componentes del *arraylist* de la siguiente manera: el primer valor que lee es el *No terminal* de la parte izquierda. Todo lo que lea hasta encontrar el valor -> será lo que se incluya en la parte derecha de la nueva regla. Cuando lea ->, quiere decir que una producción ha terminado lo cual se aumentará la fila del *array* nuevo para añadir la siguiente regla con el mismo proceso. Sigue haciendo este proceso iterativamente hasta que el *arraylist* quede completamente leído.

Por último, cambia el contenido *numeroReglas* por la suma del valor anterior más el número de nuevas reglas añadidas. Después, copia las producciones del nuevo *array* en la gramática original y la devuelve con las nuevas reglas ya introducidas.

reglasDeRedenominacion

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el apartado *7.2.2. Forma normal de Chomsky* denominado *reglasDeRedenominacion.*

metodoFuerzaBruta

Este método ejecuta la misma acción que el explicado en el apartado *7.2.2 Algoritmo FNG* denominado *metodoFuerzaBruta*.

buscaPrimerNT

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el capítulo *7.2.2 Algoritmo FNG* denominado *buscaPrimerNT*.

Iguales

Este método ejecuta las mismas operaciones que las explicadas en el capítulo *7.2.2 Algoritmo FNG* denominado *Iguales*.

matchParcial

En este apartado, el método a explicar es el mismo que el descrito en el apartado *7.2.2 Algoritmo FNG* denominado *matchParcial*.

aplicaRegla

Este método ejecuta la misma acción que el explicado en el apartado *7.2.2 Algoritmo FNG* denominado *aplicaRegla*.

ordenacionDeReglasParaFNG

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el capítulo *7.2.3. Forma normal de Greibach* denominado *ordenacionDeReglasParaFNG*.

limpiezaGramatica

Método que ejecuta métodos de esta clase para poder realizar las operaciones necesarias y transformar la gramática inicial a un formato que pueda analizar posteriormente el algoritmo.

Primero guarda el valor del axioma inicial, el número de reglas, el número de columnas y la definición de la palabra vacía. Luego llama a los métodos *reglasInnecesarias*, *reglasNoGenerativas*, *simbolosInaccesibles* y *reglasDeRedenominacion*. Si se ha realizado modificaciones de la gramática, contemplados en el *while(modificado==true)*, volverá a llamar a las funciones tantas veces como sean necesarios hasta que ya no se produzcan cambios.

Cuando *modificado=false*, llama a los métodos *ordenacionDeReglasParaFNG*, *eliminacionRecursividadIzquierdas* y *eliminacionReglasIguales* para completar la transformación.

Por último, imprime en el fichero la gramática obtenida después de las modificaciones.

mirarCadena_Fuerza_Bruta

Método que ejecuta todos los métodos de esta clase para poder realizar las operaciones necesarias y analizar la gramática inicial con las cadenas introducidas.

Primero guarda el valor del axioma inicial, el número de reglas, el número de columnas de la matriz de la gramática y la definición de la palabra vacía.

Luego llama al método *metodoFuerzaBruta* y según el valor que devuelva sabremos si la cadena pertenece o no a la gramática. En caso de pertenecer, se guardará en el fichero que contiene los resultados "*ACEPTADA cadena:* " seguido de la cadena analizada. A continuación, en el fichero que guarda los datos del árbol de decisión se escribe "*Arbol Fuerza Bruta:\n*--------

----------\n*De la cadena*", seguido de la cadena analizada, y se llama al método *arbolFuerzaBruta* explicado en el apartado *7.2.9. Árbol Fuerza bruta*.

Cuando sea una cadena que no es aceptada, se guardará en el fichero que contiene los resultados "*ACEPTADA cadena:* " seguido de la cadena analizada. A continuación, en el fichero que guarda los datos del árbol de decisión se escribe "*Arbol Fuerza Bruta:\n----------\nDe la cadena*", seguido de la cadena analizada, y debajo se guarda "*No hay árbol posible*".

Por último, deja con valor nulo todas las variables usadas únicamente para este método, para evitar conflictos de valores en las sucesivas ejecuciones, y devuelve la gramática completamente transformada.

7.2.7. Árbol CYK

El fichero *Arbol_CYK.java* se detalla en este apartado. Genera el árbol de derivación del algoritmo CYK plasmado con distintos niveles.

Para ejecutar el único método, *ejecucionArbolYK*, en la llamada se incluyen los parámetros que contienen la cadena que realiza el CYK para determinar el árbol de derivación, la gramática analizada, la longitud de la regla más larga y el nombre del fichero que mostrará los resultados.

La cadena que determina el árbol de derivación, compuesta por números de reglas y los carácteres *[* y *]*, es analizada palabra por palabra. Este estudio se realiza obteniendo el valor de la posición de la cadena a analizar (empieza en la 0 y llega hasta el último valor que la componga) y compararlo con *[,]* ó ver si es un valor numérico.

Tratándose de *[*, quiere decir que comienza un nuevo nivel en el árbol. El caso contrario es *]*, que determina que el nivel que se estaba analizando ha terminado. Cuando se lea un *[*, aumentaremos el número de | y tabuladores a imprimir antes de que se dé una regla, y cuando lea *[*, se disminuirá. Estas | y tabuladores sirven para ver visualmente en el fichero qué nivel del árbol es en el que se encuentra la regla que venga después.

Dando un valor numérico, eso quiere decir que la regla de la gramática situada en esa posición es la que genera parte de la cadena en el nivel analizado. Con el número de regla nos situamos en esa posición en la gramática. A continuación, imprimimos el número de | y tabuladores necesarios hasta llegar al último valor de tabulación, que será cuando imprima en el |--- seguido del *Terminal* de la parte izquierda de la regla, mas -> y la parte derecha de esta. Como estará en Forma Normal de Chomsky, el número máximo de valores en la parte derecha serán dos.

Terminado este proceso con todos los valores que formen la cadena, conseguiremos un árbol de derivación del estilo al que se muestra en la siguiente imagen.

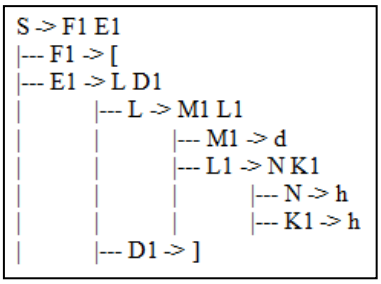


Ilustración 13. Árbol FNC

7.2.8. Árbol FNG

En este apartado se detalla el fichero *Arbol_FNG.java*, explicando las distintas funciones que lo componen. Estas son las que se muestran en los subsiguientes apartados.

sacarVectorConParentesis

Genera la cadena, compuesta por valores numéricos y los caracteres *[* y *]*, que forma el árbol de derivación. Los argumentos a incluir en la llamada son la gramática a analiza, el listado de los *No terminales* y el vector de número de reglas que forman la cadena (obtenido de la realización del método del algoritmo de fuerza bruta para la Forma Normal de Greibach).

El vector de número de reglas es analizado palabra por palabra. Este análisis se realiza obteniendo el valor de la posición de la cadena a analizar (empieza en la 0 y llega hasta el último valor que la componga). Con el número adquirido, guardamos en un *String* el carácter *[* y nos situamos en la regla de la gramática que esté situada en esa posición.

Al estar las reglas en FNG, el primer valor obtenido será un *Terminal*. A continuación, se analiza la regla comprobando si después de este *Terminal* hay o no una sucesión *de No terminales* (llamando a la función *comprobarTerminales* explicada en el siguiente subapartado). De ser así, llamaríamos de nuevo a la función para analizar los *Terminales* de dicha regla.

Cuando terminemos de analizarlos todos los *No terminales* que componen la regla, guardaríamos en el *String* el carácter *]*, indicando que esa regla ya ha terminado de analizarse y el nivel de profundidad que representa también.

Al finalizar el método se obtendrá un *String* que representa los distintos niveles del árbol con sus respectivas reglas.

comprobarTerminales

Método que comprueba si el valor pasado por parámetro pertenece a los *No terminales*. Para determinar la pertenencia, compara el valor con todos los que forman la lista de los *No terminales*. En el momento que encuentre uno devuelve que se ha encontrado (*encontrado=true*). Si no, devuelve *encontrado=false*.

ejecucionArbolFNG

El String que determina el árbol de derivación, compuesto por números de reglas y los carácteres *[* y *]*, es analizado palabra por palabra. Este análisis se realiza obteniendo el valor de la posición del *String* a analizar (empieza en la 0 y llega hasta el último valor que la componga) y compararlo con *[* o *]* ó ver si es un valor numérico.

Tratándose de *[*, quiere decir que comienza un nuevo nivel en el árbol. El caso contrario es *]*, que determina que el nivel que se estaba analizando ha terminado. Cuando se lea un *[*, aumentaremos el número de | y tabuladores a imprimir antes de que se dé una regla, y cuando lea *[*, se disminuirá. Estas | y tabuladores sirven para ver visualmente en el fichero qué nivel del árbol es en el que se encuentra la regla que venga después.

Dando un valor numérico, eso quiere decir que la regla de la gramática situada en esa posición es la que genera parte de la cadena. Con el número de regla nos situamos en esa posición en la gramática. A continuación, imprimimos el número de | y tabuladores necesarios hasta llegar al último valor de tabulación, que será cuando imprima en el |--- seguido del *Terminal* de la parte izquierda de la regla, mas -> y la parte derecha de esta.

Terminado este proceso con todos los valores que formen la cadena, conseguiremos un árbol de derivación del estilo al que se muestra en la siguiente imagen.

```
P-> [ L BA1
|--- L -> d N R E
|          |--- N -> f
|          |--- R -> v
|          |--- E -> h
|--- BA1 -> ]
```

Ilustración 14. Árbol FNG

arbolFNG

Se trata del método principal de este fichero. En él se llaman a las funciones *sacarVectorConParentesis* y *ejecucionArbolFNG* para sacar el árbol de derivación de una gramática en Forma Normal de Greibach.

Por último, deja con valor nulo todas las variables utilizadas para evitar conflictos de valores en las sucesivas ejecuciones del menú principal.

7.2.9. Árbol Fuerza bruta

En este apartado se detalla el fichero *Arbol_Fuerza_Bruta.java*, explicando las distintas funciones que lo componen. Estas son las que se muestran en los subsiguientes apartados.

sacarVectorConParentesis

Genera la cadena, compuesta por valores numéricos y los caracteres *[* y *]*, que forma el árbol de derivación. Los argumentos a incluir en la llamada son la gramática a analiza, el listado de los *No terminales* y el vector de número de reglas que forman la cadena (obtenido de la realización del método del algoritmo de Fuerza Bruta).

El vector de número de reglas es analizado palabra por palabra. Este análisis se realiza obteniendo el valor de la posición de la cadena a analizar (empieza en la 0 y llega hasta el último valor que la componga). Con el número adquirido, guardamos en un *String* el carácter *[* y nos situamos en la regla de la gramática que esté situada en esa posición.

A continuación, se analiza la regla comprobando si el valor es un *Terminal* o *No terminal* (llamando a la función *comprobarTerminales* explicada en el siguiente subapartado). De ser así, llamaríamos de nuevo a la función para analizar los *Terminales* de dicha regla.

Cuando terminemos de analizarlos todos los *No terminales* que componen la regla, guardaríamos en el *String* el carácter *]*, indicando que esa regla ya ha terminado de analizarse y el nivel de profundidad que representa también.

Al finalizar el método se obtendrá un *String* que representa los distintos niveles del árbol con sus respectivas reglas.

comprobarTerminales

El método contemplado en este apartado, ejecuta la misma acción que el explicado en el apartado *7.2.9. Árbol FNG* denominado *comprobarTerminales*.

ejecucionArbolFB

El *String* que determina el árbol de derivación, compuesto por números de reglas y los caracteres *[* y *]*, es analizado palabra por palabra. Este análisis se realiza obteniendo el valor de la posición del *String* a analizar (empieza en la 0 y llega hasta el último valor que la componga) y compararlo con *[* o *]* ó ver si es un valor numérico.

Tratándose de *[*, quiere decir que comienza un nuevo nivel en el árbol. El caso contrario es *]*, que determina que el nivel que se estaba analizando ha terminado. Cuando se lea un *[*, aumentaremos el número de | y tabuladores a imprimir antes de que se dé una regla, y cuando lea *[*, se disminuirá. Estas | y tabuladores sirven para ver visualmente en el fichero qué nivel del árbol es en el que se encuentra la regla que venga después.

Dando un valor numérico, eso quiere decir que la regla de la gramática situada en esa posición es la que genera parte de la cadena. Con el número de regla nos situamos en esa posición en la gramática. A continuación, imprimimos el número de | y tabuladores necesarios

hasta llegar al último valor de tabulación, que será cuando imprima en el |--- seguido del *Terminal* de la parte izquierda de la regla, mas -> y la parte derecha de esta.

Terminado este proceso con todos los valores que formen la cadena, conseguiremos un árbol de derivación del estilo al que se muestra en la siguiente imagen.

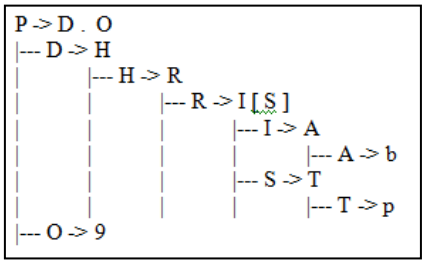


Ilustración 15. Árbol FB

arbolFuerzaBruta

Se trata del método principal de este fichero. En él se llaman a las funciones *sacarVectorConParentesis* y *ejecucionArbolFB* para sacar el árbol de derivación en una gramática analizada con Fuerza Bruta.

Por último, deja con valor nulo todas las variables utilizadas para evitar conflictos de valores en las sucesivas ejecuciones del menú principal.

7.2.10. Lectura gramática

El fichero *Lectura_Fichero.java* se detalla en este apartado. Realiza la lectura del fichero que contiene la gramática y comprueba que se haya seguido la estructura establecida para este. Los métodos que lo componen se muestran en los subsiguientes apartados.

mirarApertura

El fichero únicamente contempla un método. La función de este consiste en abrir el fichero, cuya ruta se le pasa por parámetro, y comprobar que se puede abrir sin errores. En el caso de darse un error, avisa al usuario mostrando por pantalla *"Error: Fichero no encontrado."*. De lo contrario, devolverá true al fichero que realizó su llamada.

aperturaFichero

Este método se encarga de abrir un fichero, cuya ruta se le pasa por parámetro, y comprobar que al hacerlo no surja ningún error. En el caso de darse uno, avisa al usuario mostrando por pantalla *"Error: Fichero no encontrado."*. De lo contrario, devolverá *true* al método que realizó su llamada.

guardadoNumeroDatos

Comenzamos leyendo la primera línea del fichero. Esta tiene que empezar por "LISTA". En caso de no ser así, se muestra un error y se cierra el programa. A continuación

debe de estar escrito “TERMINALES” ó “NO”. Si se encuentra otro valor también muestra un error y termina el programa.

Si la palabra leída ha sido “TERMINALES:”, todo lo que contenga después, y que se considere *Terminal*, se guardará en el *arrayList* de *Terminales*. El proceso de guardado se realiza leyendo todas las palabras contempladas en la línea, comprobando que no sean un comentario o igual a la palabra “LISTA” (en cuyo caso ya habrán terminado la definición de los *Terminales*). Cuando termine la línea y no se haya encontrado “LISTA”, se lee la siguiente y se analiza de la misma forma. Solo parará cuando lea “LISTA”. La comprobación de si es un comentario se realizará llamando al método *sacarComentario*.

Si la palabra leída ha sido “NO”, después tiene que estar “TERMINALES:”. Al no cumplir esta sucesión, se muestra un error y se cierra el programa. El proceso de análisis realiza leyendo todas las palabras contempladas en la línea, comprobando que no sean un comentario o igual a la palabra “LISTA” (en cuyo caso ya habrán terminado la definición de los *No terminales*). Cuando termine la línea y no se haya encontrado “LISTA”, se lee la siguiente y se analiza de la misma forma. Solo parará cuando lea “LISTA”.

El programa para de leer *Terminaes* o *No terminales* cuando lea “LISTA”. El porqué de este detenimiento es que a continuación leerá los que no haya analizado, es decir, si estaba guardando los *Terminales* ahora leerá los *No terminales* o viceversa. El proceso después de ese segundo “LISTA” es el mismo que para el primero. La única excepción es que después de los *No terminales* tienen que leerse los *Terminales* o viceversa. En caso de darse dos iguales, muestra un error y cierra el programa.

Posteriormente, el sistema debe leer en el fichero la sucesión de las palabras “PALABRA” y “VACIA:”. A continuación, siempre que no sea un comentario, leerá la definición de la palabra vacía y la guardará en su correspondiente variable. Dicha definición llegará hasta que en la siguiente línea contemple el primer valor como “GRAMATICA:”.

Por último vendrá la definición de la gramática. Este método leerá cuantas reglas posee la gramática y la longitud de la regla más larga. Ambos valores los guardará en sus correspondientes variables.

sacarComentario

Analiza si la apalabra pasada por parámetro es el inicio de un comentario, es el final de este o es un *Terminal* de una cadena.

Lo primero que analiza es si el primer carácter del *String* pasado por parámetro es /. Si lo es, pero después no se encuentra el carácter *, cambia *retornar=0* que indica que se trata de

un *Terminal* de la palabra. En cambio, si el siguiente es un *, el método modifica la variable *retornar=1* para indicar que es un comentario.

Al lee en primer lugar es el carácter * y a continuación una /, el método cambia *retornar=2* para indicar la finalización de un comentario.

El método, al terminar la ejecución, devolverá el valor de *retornar*.

guardadoGramatica

Método que guarda las reglas que hay en el fichero dentro del *array* asignado a la gramática.

Inicialmente se leen todas las líneas del fichero hasta dar con la que empiece por "GRAMATICA:". A continuación, siempre que no se detecte que se está leyendo un comentario (comprobación que se realizará llamando al método *sacarComentario*), se guardará cada una de las reglas que lea en una posición del *array* de la gramática.

Por cada una de las líneas, todo lo que lea hasta el valor -> irá guardado en la posición 0 del *array* de la gramática y será el *No terminal* que genere la regla. Todo el contenido después de -> será la parte derecha. Para guardar esta parte, cada uno de los valores separados por un espacio será un *Terminal* o *No terminal* de la gramática. Cuando lee un valor, llama al método *comprobarPalabras* para comprobar si está bien definido según la lista de *Terminales*, *No terminal*es o la palabra vacía, y no lo guarda si la respuesta de pertenencia es correcta. Si hubiese algo mal definido, muestra un error y cierra el programa.

Finalmente, devuelve el *array* de la gramática con todas las reglas dentro.

comprobarPalabras

Función que comprueba si los *Terminales* y *No terminales* definidos en la gramática están especificados como tal en los *arrayList* de *Terminales* y *No terminales* ó es igual a la definición de la palabra vacía.

El valor que se introduzca por parámetro será comparado por los valores de la lista de *Terminales* y *No terminales*. Si no pertenece a ninguna de estas, comprueba si se trata de la palabra vacía. En el caso de encontrarse en alguna de ellas, el método devolverá true, y por el contrario devolverá false.

erroresDeComentarios

Método que elimina las filas completas con valores a *null* del *array* de la gramática. Estas son producidas cuando se leen líneas enteras de comentarios y no se guarda nada en la gramática. El único parámetro de la función es la gramática a analizar.

Inicialmente se crea un nuevo *array* con los mismos parámetros que el de la gramática. En él se guardan las fialas del *array* de la gramática que no comiencen por *null*. Cada vez que se encuentre una con valor, se aumenta el valor de la variable *iGramModifi*.

Después se modifican los parámetros de la gramática, poniendo tantas filas como valor tenga *iGramModifi*. De esta manera lograremos tener la gramática con el número de filas igual al número de reglas.

Por último se copian los valores del nuevo *array* al de la gramática y se devuelve este último por parámetro.

ejecucionLecturaFichero

Método que ejecuta todos los métodos de esta clase para poder realizar las operaciones necesarias de lectura del fichero de la gramática.

Primero guarda el valor del axioma inicial, el número de reglas, el número de columnas de la matriz de la gramática, el número de *Terminales* y *No terminales* y la definición de la palabra vacía.

Luego llama a los métodos *aperturaFichero*, *guardadoNumeroDatos*, *guardadoGramatica* y *erroresDeComentarios*. Al final de la ejecución de todos ellos obtendremos la gramática introducida en el fichero con todos sus parámetros.

7.2.11. Lectura cadena

Este apartado refleja el contenido del fichero *Lectura_Cadena.java* y la explicación de los métodos que contiene. Estos son las que se muestran en los subsiguientes apartados.

mirarApertura

El fichero únicamente contempla un método. La función de este consiste en abrir el fichero, cuya ruta se le pasa por parámetro, y comprobar que se puede abrir sin errores. En el caso de darse un error, avisa al usuario mostrando por pantalla *"Error: Fichero no encontrado."*. De lo contrario, devolverá true al fichero que realizó su llamada.

7.2.12. Lectura varias cadenas

En este apartado se detalla el fichero *Lectura_Varias_Cadenas.java*, realizando una explicación de las distintas funciones que lo componen. *Lectura_Varias_Cadenas* leer el fichero en el que se encuentran las cadenas separadas todos sus *Terminales* por un espacio. Los métodos que se muestran en los subsiguientes apartados son los que componen el archivo.

aperturaFichero

Este método se encarga de abrir un fichero, cuya ruta se le pasa por parámetro, y comprobar que al hacerlo no surja ningún error. En el caso de darse uno, avisa al usuario

mostrando por pantalla *"Error: Fichero no encontrado."*. De lo contrario, devolverá *true* al método que realizó su llamada.

lecturaCadena

Este método irá leyendo palabra por palabra del fichero que contenga las cadenas. Para poder analizarlas, primero lee una línea del archivo, y a continuación la desglosa en palabras.

Después, llama al método *sacarComentario* (explicado en el apartado siguiente) y obtiene un valor. Este resultado indica si se está leyendo un comentario, si es el final de este ó si se está examinando una palabra a analizar.

En el caso de empezar a analizar un comentario, el método modificará la variable *activar* poniendo el valor 1 para ella, y seguiría leyendo las palabras del fichero. Aunque se siga leyendo palabras solo se comprobará si definen la finalización de un comentario. Cuando determine el final, el sistema pondrá la variable *activar=0*.

En el caso de ser *activar=0*, la palabra leída la compara con el valor de la palabra vacía. Si ambos son iguales, guarda la expresión leída como una cadena, es decir, no realiza la acción de separarla en caracteres. Cuando sean distintos valores, cada palabra leída de la línea se guardará por separado en un *arraylist*. Terminada de analizar, guardará en el *arraylist* el valor -> para indicar que lo guardado anteriormente es una cadena completa (pese a que esté desglosada en diferentes palabras).

El proceso de análisis descrito arriba se seguirá para todas las líneas del archivo.

sacarComentario

Analiza si la apalabra pasada por parámetro es el inicio de un comentario, es el final de este o es un *Terminal* de una cadena.

Lo primero que analiza es si el primer carácter del *String* pasado por parámetro es /. Si lo es, pero después no se encuentra el carácter *, cambia *escadena=0* que indica que se trata de un *terminal* de la palabra. En cambio, si el siguiente es un *, el método modifica la variable *escadena=1* y *activar=1* para indicar que es un comentario.

Al lee en primer lugar es el carácter * y a continuación una /, el método cambia *escadena=1* y *activar=0* para indicar la finalización de un comentario.

En caso de ser cualquier otro valor y no estar el método modifica las variables *escadena* y *activar* de tal manera que queden *escadena=0* y *activar=0* y así indicar que se trata de un *Terminal* de la palabra.

ejecucionLecturaCadena

Se trata del método principal de este fichero. En él se llaman a las funciones *aperturaFichero* y *lecturaCadena* para sacar las cadenas separadas todos sus *Terminales* por un espacio.

7.2.13. Lectura carácter a carácter

En este apartado se detalla el fichero *Lectura_carácter_a_carácter.java*, explicando las distintas funciones que lo componen. *Lectura_carácter_a_carácter* realiza la acción de leer el fichero en el que se encuentran las cadenas compuestas por *Terminales* de un solo carácter. Los métodos que se muestran en los subsiguientes apartados son los que componen el fichero.

aperturaFichero

Este método realiza el mismo proceso de análisis que el descrito en el apartado *7.2.12. Lectura varias cadenas* con el mismo nombre, es decir, que *aperturaFichero*.

lecturaCadena

Este método irá leyendo palabra por palabra del fichero que contenga las cadenas a analizar. Para realizar este objetivo, primero lee una línea del archivo y a continuación la desglosa en palabras.

Después, llama al método *sacarComentario* (explicado en el apartado siguiente) y obtiene un valor. Este resultado indica si se está leyendo un comentario, si es el final de este ó si se está examinando una palabra a analizar.

En el caso de empezar a analizar un comentario, el método cambiará la variable *activar* a valor 1 y seguiría leyendo las palabras del fichero, pero no realizaría ninguna acción con ellas. Cuando determine el final de un comentario, el sistema pondrá la variable *activar=0*.

En el caso de ser *activar=0*, la palabra leída la compara con el valor de la palabra vacía. Si ambos son iguales, guarda la expresión leída como una cadena, es decir, no realiza la acción de separarla en caracteres. De ser distinto, cada carácter que componga la palabra se guardará por separado en un *arraylist*. De esta forma, el programa posteriormente podrá determinar cuál es cada uno de los *Terminales* que la componen.

Terminada de analizar una línea del fichero, se guardará en el *arraylist* el valor -> para indicar que lo guardado anteriormente es una palabra completa (pese a que esté desglosada en cada uno de sus *Terminales*).

El proceso de análisis descrito arriba se seguirá para todas las líneas del archivo.

sacarComentario

Este método realiza el mismo proceso de análisis que el descrito en el apartado *7.2.12. Lectura varias cadenas* con el mismo nombre, es decir, que *sacarComentario*.

ejecucionLecturaCadena

Se trata del método principal de este fichero. En él se llaman a las funciones *aperturaFichero* y *lecturaCadena* para abrir el fichero que contiene las cadenas a analizar y separarlas por *Terminales*. Antes de realizar estas llamadas, se guardará el valor de la palabra vacía para poder distinguirla dentro del fichero.

7.3. Decisiones de codificación en el Sistema

En este apartado se explica el porqué de escoger un algoritmo recursivo con backtracking frente a uno iterativo para realizar la codificación del derivador por Fuerza Bruta.

El problema principal consiste en encontrar un camino que, derivando por las distintas reglas que forman la gramática, obtenga la cadena a analizar. Para ello es necesario probar todas las combinaciones posibles hasta encontrar una que dé con la solución. Puede darse el caso en el que no se encuentre ninguna solución.

Un algoritmo iterativo se utiliza cuando es necesario realizar tareas repetitivas. Además, la codificación de un algoritmo recursivo generalmente es menos eficiente que uno iterativo dado que necesita mayor tiempo de ejecución y conlleva más costes de memoria. Esto nos llevaría a tomar la decisión de escoger el iterativo dado que es una búsqueda iterativa por reglas de la gramática y nos ahorraríamos en tiempo y memoria durante la ejecución.

Pero hay un punto que el algoritmo iterativo no cumple y es crucial para la resolución del problema. Un algoritmo recursivo con backtracking realizar en tareas parciales, las cuales se comprueban y a su vez se descompondrán en subtareas. Además, y esta es la parte principal, es capaz de volver atrás si se comprueba que la rama escogida de análisis no lleva a una solución.

Como hemos mencionado, el algoritmo debe explorar la gramática hasta encontrar una posible solución o después de analizar todas las reglas puede que no se llegue a dar la cadena analizada. Este proceso conlleva tomar una rama del árbol y ver si llega a dar solución o no, y en caso de no ser la adecuada volver atrás y analizar desde otro punto posible.

Ya que se trata de una búsqueda combinatoria, el algoritmo más adecuado para el derivador por Fuerza Bruta es el algoritmo recursivo con backtracking.

8. Evaluación

8.1. Plan de pruebas

En este apartado se describirá el plan de pruebas a seguir para determinar que los requisitos obtenidos estén realizados adecuadamente, y de esta forma evaluar que el producto haya sido realizado satisfactoriamente.

8.1.1. Especificación del Entorno de Pruebas

El entorno de pruebas estará formado por distintos ordenadores con sistema operativo Windows. Las características de cada uno de ellos son las siguientes.

- Equipos Windows.
  - **Procesador**: Intel® Core™2 Duo.
  - **Memoria RAM**: 4GB.
  - **Velocidad**: 2.00 GHz.
  - **Sistema operativo**: 32 bits.

8.1.2. Especificación formato de Pruebas

Se incluirá una tabla por cada una de las pruebas, cuyos campos y formato se definen a continuación:

- **Identificador**. Servirá como identificador de cada prueba. Seguirá la nomenclatura P-XX– Dígito#1 Dígito#2, que permitirá identificar cada prueba de manera unívoca.
- **Objetivo.** Define qué se quiere comprobar al realizar la prueba.
- **Entradas.** Identifica los distintos valores posibles de entrada al sistema para la realización de las distintas pruebas.
- **Salida.** Indica los valores de salida generados por el sistema al realizar las distintas pruebas.
- **Precondiciones.** Condiciones que deben cumplirse para que se pueda realizar la prueba.
- **Secuencia.** Pasos a seguir por el usuario para la realización de la prueba.

P-XX	
Objetivo	
Entradas	
Salidas	
Precondiciones	
Secuencia	

Tabla 85. Plantilla pruebas

8.2. Casos De Prueba

El listado de los casos de uso es la siguiente.

- **P-01.** Comprobación que se realiza correctamente la Forma Normal de Chomsky.
- **P-02.** Comprobación que se realiza correctamente la Forma Normal de Greibach.
- **P-03.** Confirmación de pertenencia o no de la cadena en gramática Forma Normal de Greibach.
- **P-04.** Certificación pertenencia o no de cadena en gramática Forma Normal de Chomsky.
- **P-05.** Certificación pertenencia o no de cadena en Fuerza Bruta.
- **P-06.** Salir de la aplicación.
- **P-07.** Cambiar rutas de los ficheros.

P-01	
Objetivo	Comprobación que se realiza correctamente la Forma Normal de Chomsky.
Entradas	Introducimos un fichero .txt con la gramática en Forma Normal de Chomsky.
	Introducimos un fichero .txt con la gramática sin Forma Normal de Chomsky.
Salidas	No se realizaría ningún cambio en la gramática y se mostraría igual en el fichero .txt generado para la salida de la transformación.
	Mostraría en un fichero .txt generado para la salida de la transformación todos los pasos seguidos para transformar a Forma Normal de Chomsky.
Precondiciones	1. El usuario debe haber escogido la opción de transformar la gramática a Forma Normal de Chomsky.
Secuencia	1. El usuario entra en la aplicación (si no está ya ejecutándose). 2. Inserta la ruta del fichero de gramática y el de la cadena. 3. Pulsa la opción transformar la gramática a Forma Normal de Chomsky. 4. El sistema muestra el resultado final de la transformación y genera un fichero .txt con los pasos seguidos para dicha transformación.

Tabla 86. P-01

P-02	
Objetivo	Comprobación que se realiza correctamente la Forma Normal de Greibach
Entradas	Introducimos un fichero .txt con la gramática en Forma Normal de Greibach.
	Introducimos un fichero .txt con la gramática sin Forma Normal de Greibach.
Salidas	No se realizaría ningún cambio en la gramática y se mostraría igual en el fichero .txt generado para la salida de la transformación.
	Mostraría en un fichero .txt generado para la salida de la transformación todos los pasos seguidos para transformar a Forma Normal de Greibach.
Precondiciones	1. El usuario debe haber escogido la opción de transformar la gramática a Forma Normal de Greibach.
Secuencia	1. El usuario entra en la aplicación (si no está ya ejecutándose). 2. Inserta la ruta del fichero de gramática 3. Pulsa la opción transformar la gramática a Forma Normal de Greibach. 4. El sistema muestra el resultado final de la transformación y genera un fichero .txt con los pasos seguidos para dicha transformación.

Tabla 87. P-02

P-03	
Objetivo	Certificación pertenencia o no de cadena en gramática Forma Normal de Greibach.
Entradas	Introducimos un fichero .txt con la gramática en Forma Normal de Greibach. Introducimos un fichero .txt con una cadena que acepte.
	Introducimos un fichero .txt con la gramática en Forma Normal de Greibach. Introducimos un fichero .txt con una cadena que no acepte.
	Introducimos un fichero .txt con la gramática sin Forma Normal de Greibach. Introducimos un fichero .txt con una cadena que acepte.
	Introducimos un fichero .txt con la gramática sin Forma Normal de Greibach. Introducimos un fichero .txt con una cadena que no acepte.
	Introducimos un fichero .txt con la gramática sin Forma Normal de Greibach. Introducimos un fichero .txt con cadenas que no acepte y que acepte.
Salidas	No se realizaría ningún cambio en la gramática y se mostraría igual en el fichero .txt generado para la salida de la transformación. Aceptaría la palabra introducida. Además generaría los ficheros correspondientes al árbol de generación y la pertenencia de la palabra como aceptada.
	No se realizaría ningún cambio en la gramática y se mostraría igual en el fichero .txt generado para la salida de la transformación. No aceptaría la palabra introducida. Además generaría los ficheros

	correspondientes al árbol de generación (vacío) y la pertenencia de la palabra como aceptada.
	Mostraría en un fichero .txt generado para la salida de la transformación todos los pasos seguidos para transformar a Forma Normal de Greibach. Aceptaría la palabra introducida. Además generaría los ficheros correspondientes: al árbol de generación y la pertenencia de la palabra como aceptada.
	Mostraría en un fichero .txt generado para la salida de la transformación todos los pasos seguidos para transformar a Forma Normal de Greibach. No aceptaría la palabra introducida. Además generaría los ficheros correspondientes: al árbol de generación (vacío) y la pertenencia de la palabra como aceptada.
	Mostraría en un fichero .txt generado para la salida de la transformación todos los pasos seguidos para transformar a Forma Normal de Greibach. No aceptaría la palabra que no perteneciesen y sí las que sí perteneciesen. Además generaría los ficheros correspondientes: los árboles de generación (vacío en caso de no pertenencia y mostrado en caso de pertenecer) y la pertenencia de la palabra como aceptada o no aceptada.
Precondiciones	1. El usuario debe haber escogido la opción de comprobar palabra en gramática Forma Normal de Greibach.
Secuencia	1. El usuario entra en la aplicación (si no está ya ejecutándose). 2. Inserta la ruta del fichero de gramática y el de la cadena. 3. Pulsa la opción comprobar palabra en gramática Forma Normal de Greibach. 4. El sistema muestra el resultado final de la transformación y genera un fichero .txt con los pasos seguidos para dicha transformación. 5. Generaría los ficheros .txt correspondientes al árbol de generación y la pertenencia de la palabra como aceptada o no.

Tabla 88. P-03

P-04	
Objetivo	Certificación pertenencia o no de cadena en gramática Forma Normal de Chomsky.
Entradas	Introducimos un fichero .txt con la gramática en Forma Normal de Chomsky. Introducimos un fichero .txt con una cadena que acepte.
	Introducimos un fichero .txt con la gramática en Forma Normal de Chomsky. Introducimos un fichero .txt con una cadena que no acepte.
	Introducimos un fichero .txt con la gramática sin Forma Normal de Chomsky. Introducimos un fichero .txt con una cadena que acepte.
	Introducimos un fichero .txt con la gramática sin Forma Normal de Chomsky.

	Introducimos un fichero .txt con una cadena que no acepte.
	Introducimos un fichero .txt con la gramática sin Forma Normal de Chomsky. Introducimos un fichero .txt con cadenas que no acepte y que acepte.
Salidas	No se realizaría ningún cambio en la gramática y se mostraría igual en el fichero .txt generado para la salida de la transformación. Aceptaría la palabra introducida. Además generaría los ficheros correspondientes a la tabla del CYK, el árbol de generación y la pertenencia de la palabra como aceptada.
	No se realizaría ningún cambio en la gramática y se mostraría igual en el fichero .txt generado para la salida de la transformación. No aceptaría la palabra introducida. Además generaría los ficheros correspondientes a la tabla del CYK (vacía), el árbol de generación (vacío) y la pertenencia de la palabra como aceptada.
	Mostraría en un fichero .txt generado para la salida de la transformación todos los pasos seguidos para transformar a Forma Normal de Greibach. Aceptaría la palabra introducida. Además generaría los ficheros correspondientes: la tabla del CYK, el árbol de generación y la pertenencia de la palabra como aceptada.
	Mostraría en un fichero .txt generado para la salida de la transformación todos los pasos seguidos para transformar a Forma Normal de Greibach. No aceptaría la palabra introducida. Además generaría los ficheros correspondientes: a la tabla del CYK , el árbol de generación (vacío) y la pertenencia de la palabra como aceptada.
	Mostraría en un fichero .txt generado para la salida de la transformación todos los pasos seguidos para transformar a Forma Normal de Greibach. No aceptaría la palabra que no perteneciesen y sí las que sí perteneciesen. Además generaría los ficheros correspondientes: las tablas del CYK, los árboles de generación (vacío en caso de no pertenencia y mostrado en caso de pertenecer) y la pertenencia de la palabra como aceptada o no aceptada.
Precondiciones	1. El usuario debe haber escogido la opción de comprobar palabra en gramática Forma Normal de Chomsky (algoritmo CYK).
Secuencia	1. El usuario entra en la aplicación (si no está ya ejecutándose). 2. Inserta la ruta del fichero de gramática y el de la cadena. 3. Pulsa la opción comprobar palabra en gramática Forma Normal de Chomsky. 4. El sistema muestra el resultado final de la transformación y genera un fichero .txt con los pasos seguidos para dicha transformación. 5. Generaría los ficheros .txt correspondientes a la tabla del CYK, el árbol de generación y la pertenencia de la palabra como aceptada o no.

Tabla 89. P-04

P-05	
Objetivo	Certificación pertenencia o no de cadena en Fuerza Bruta.
Entradas	Introducimos un fichero .txt con la gramática sin formato. Introducimos un fichero .txt con una cadena que acepte.
	Introducimos un fichero .txt con la gramática sin formato. Introducimos un fichero .txt con una cadena que no acepte.
	Introducimos un fichero .txt con la gramática sin formato. Introducimos un fichero .txt con cadenas que no acepte y que acepte.
Salidas	Aceptaría la palabra introducida. Además generaría los ficheros correspondientes al árbol de generación y la pertenencia de la palabra como aceptada.
	No aceptaría la palabra introducida. Además generaría los ficheros correspondientes al árbol de generación (vacío) y la pertenencia de la palabra como aceptada.
	No aceptaría la palabra que no perteneciesen y sí las que sí perteneciesen. Además generaría los ficheros correspondientes a los árboles de generación (vacío en caso de no pertenencia y mostrado en caso de pertenecer) y la pertenencia de la palabra como aceptada o no aceptada.
Precondiciones	1. El usuario debe haber escogido la opción de comprobar palabra en gramática Fuerza Bruta.
Secuencia	1. El usuario entra en la aplicación (si no está ya ejecutándose). 2. Inserta la ruta del fichero de gramática y el de la cadena. 3. Pulsa la opción comprobar palabra en gramática Fuerza Bruta. 4. Generaría los ficheros .txt correspondientes al árbol de generación y la pertenencia de la palabra como aceptada.

Tabla 90. P-05

P-06	
Objetivo	Salir de la aplicación.
Entradas	No hay entrada.
Salidas	La aplicación ha sido cerrada.
Precondiciones	1. El usuario debe tener el programa abierto.
Secuencia	1. El usuario entra en la aplicación (si no está ya ejecutándose) . 2. Inserta la ruta del fichero de gramática y el de la cadena. 3. Pulsa la opción de salir del programa. 4. El usuario ve la aplicación cerrada.

Tabla 91. P-06

P-07	
Objetivo	Cambiar rutas de los ficheros.
Entradas	No hay entrada.
Salidas	La aplicación ha sido cerrada.
Precondiciones	1. El usuario debe tener el programa abierto.
Secuencia	1. El usuario entra en la aplicación (si no está ya ejecutándose). 2. Inserta la ruta del fichero de gramática y el de la cadena (aunque no los vaya a usar). 3. Pulsa la opción de cambiar las rutas. 4. Inserta la nueva ruta del fichero de gramática y el de la cadena.

Tabla 92. P-07

8.3. Pruebas con usuarios

En este apartado explicaremos cómo hemos recopilado información de los usuarios con respecto a la satisfacción, necesidades o elementos innecesarios de la aplicación *Naarpe*.

La técnica escogida es el cuestionario. Con este método intentamos obtener información general sobre la valoración de los usuarios respecto a la aplicación. Esta opción fue elegida por la facilidad de recopilar información más general y descartar opciones de manera sencilla.

Los participantes constan de alumnos graduados en Ingeniería Informática por la Universidad Carlos III de Madrid y alumnos cursando actualmente el Grado en Ingeniería Informática en la Universidad Carlos III de Madrid.

El cuestionario a completar es el siguiente.

CUESTIONARIO

1. **¿Ves útil el programa?**

   □ Si. □ No.

2. **¿Las preguntas que realiza el sistema están claramente expresadas?**

   □ Si. □ No.

3. **Los ficheros que genera el sistema, ¿los ves de utilidad?**

   □ Si. □ No.

4. **¿La información que se genera en los ficheros está claramente expresada?**

   □ Si. □ No.

5. **¿Las respuestas que realiza el sistema están claramente expresadas?**

   □ Si. □ No.

6. **¿El manejo del programa es intuitivo?**

   □ Si. □ No.

7. **¿Qué aspectos mejorarías?**

Para poder completarlo, los usuarios leyeron el manual de la aplicación y realizaron pruebas con todas las opciones que permite el programa. La valoración recopilada ha sido muy satisfactoria. Todas las preguntas, exceptuando la 4, obtuvieron un 100% de la calificación *Si*. La respuesta de la pregunta 4 fue un 75% *Si*. La nota negativa se produjo por el siguiente comentario que el usuario escribió en la pregunta 8: está bien poner en los ficheros las reglas eliminadas pero también deberían de estar las nuevas. Y si pudiese ser, explicar cómo se generan, las transformaciones que hace el programa.

Una recopilación de los comentarios que rellenaron los usuarios en la cuestión número 8 es la siguiente:

- Al introducir un error en la gramática, cuando lo detecta el programa, que no se detenga. Poniendo una respuesta incorrecta salta un mensaje y vuelve a aparecer el menú. Con

los errores en la gramática se debería de hacer lo mismo. Resulta tedioso ejecutar el programa desde cero.

- Sería recomendable que el programa mostrase un mensaje diciendo que ha terminado de analizar las palabras, y si quiere continuar, pulsando "aceptar", entonces vuelva al menú.
- Mejoraría si tuviese interfaz grafica con la que trabajar. Para distinguir mejor el menú de las respuestas del programa.

Las propuestas realizadas se consideran útiles para incluirlas en posteriores mejoras del sistema.

9. Estudio de la complejidad

En este apartado se realizará un análisis y contraste de la complejidad de los algoritmos Cocke-Younger-Kasami (CYK) y derivación por Fuerza Bruta (FB), implementados en *Java*, para determinar si una palabra pertenece al lenguaje generado por una gramática. También se pretende verificar que la herramienta *Naarpe* es comparable a otra herramienta como *JFLAP*, y es capaz de abordar problemas de tamaño razonable en un tiempo aceptable.

A los algoritmos mencionados se les pasará como parámetro una gramática y una palabra para determinar si pertenece o no al lenguaje generado por esta. Este proceso de decisión es denominado como una prueba. Para cada una de las pruebas, a ambos algoritmos se les pasará la misma gramática y la misma palabra a analizar.

Con lo explicado en el párrafo anterior, se observa que realizaremos un estudio empírico de los dos algoritmos. Para dicho análisis se tomará como referencia el tiempo que tarda en tomar la decisión sobre si la palabra pertenece o no a la gramática. Desde el punto de vista de *usabilidad* consideraremos que los tiempos de respuesta superiores a un minuto entran en el rango de poco aceptables, especialmente a la hora de realizar prácticas por parte del alumnado.

Además, se realizará una comparación de los dos algoritmos implementados con la herramienta *JFLAP*. Tal y como se comentó en el apartado *1.1. Motivación*, la herramienta *JFLAP* posee y proporciona unos recursos limitados. Tan solo pueden utilizarse en la definición de gramáticas 26 *No terminales*, aunque en el caso de utilizar algoritmos que realicen transformaciones a FNC, el número de *No terminales* desciende a 10 o 14.

También se realiza un estudio sobre una gramática con menos de 10 producciones, la cual a *JFLAP* le suponen un gran esfuerzo computacional comprobar la pertenencia de las cadenas. Esta gramática se explica en el subapartado *9.3.Gramática de expresiones de suma de* números binarios.

El análisis de estos puntos se abordará desde un equipo con las características mencionadas en el apartado *8.1.1. Especificación del Entorno de Pruebas*. Las siguientes pruebas de los subapartados darán respuesta a las partes planteadas en este estudio.

9.1. Caso mínimo

En este apartado se tomará el caso mínimo, es decir, la gramática con menor tamaño y la palabra con menor tamaño. El realizar esta prueba se debe a establecer el tiempo que tarda los algoritmos CYK y el derivador por Fuerza Bruta de *Naarpe* con el Fuerza Bruta de *JFLAP*, en determinar la palabra más sencilla que se pueda generar.

El caso mínimo trata la regla $S \rightarrow a$ en la gramática y la comprobación de la cadena *a*, para verificar su pertenencia, y la palabra *b*, para determinar que no puede ser generada.

Realizado el análisis, los resultados obtenidos son los que se muestra en la siguiente tabla.

Algoritmo		**Tiempo**	
		Palabra aceptada	**Palabra no aceptada**
Naarpe	Cocke-Younger-Kasami	0 milisegundos	0 milisegundos
	Fuerza Bruta	0 milisegundos	0 milisegundos
JFLAP		0 milisegundos	0 milisegundos

Tabla 93. Tiempo de análisis

El tiempo de análisis para cada uno de estos algoritmos, según ha demostrado esta prueba, es de 0 milisegundos (tanto para palabras aceptadas como las que no).

La palabra sólo está compuesta por un *terminal* y la gramática tiene una única regla, esto hace que el análisis sea sencillo y se requiera de escaso cómputo. Los resultados obtenidos son los esperados.

9.2. Gramática de expresiones de suma de números binarios

Esta prueba contiene una gramática sencilla, la cual es capaz de derivar expresiones de suma de números binarios. La definición de esta es:

$$S \rightarrow N \mid S + S$$

$$N \rightarrow 0 \mid 1 \mid 0\ N \mid 1\ N$$

Realizamos las pruebas, con palabras pertenecientes a la gramática, usando el derivador de Fuerza Bruta que contiene *JFLAP*. Los resultados obtenidos son los siguientes:

Cadena aceptada	**Tiempo análisis***
0	**
0+1	**
0+1+0	220
0+1+0+1	1777
0+1+0+1+0	2552
0+1+0+1+0+1	42960

Tabla 94. Tiempos JFLAP

*Los tiempos están expresados en milisegundos.

** La respuesta de la ejecución se realiza de forma instantánea, lo cual impide tomar el tiempo transcurrido.

La prueba muestra que al derivar palabras con *JFLAP* con el algoritmo de Fuerza Bruta, el tiempo de obtención de respuesta puede fácilmente sobrepasar el tiempo de 1 minuto para palabras con un número de *Terminales* muy pequeño. Este hecho resulta inaceptable dado que la gramática es muy elemental.

La doble recursividad y la ambigüedad, dada con la regla la regla $S \rightarrow S + S$, pueden influir en el mal resultado de los tiempos, pero además, el derivador de *JFLAP* por Fuerza Bruta parece basarse en una implementación poco eficiente

Por este motivo, el uso del derivador por Fuerza Bruta de *JFLAP* es inviable en la práctica.

9.3. Gramática con distintos tamaños de palabras

En este apartado se especifican las distintas pruebas realizadas sobre una gramática y distintos tamaños de palabra, tanto que sean posibles de generar como no. Con ello evaluaremos la influencia del tamaño de la palabra en el tiempo de decisión de pertenencia.

El proceso de pruebas del programa se ha realizado con numerosas gramáticas, pero en este estudio de la complejidad solo se mostrará un caso particular. Hemos escogido la siguiente gramática.

S → Expresion

Expresion → Número | Expresion + Expresion | Expresion – Expresion | Expresion * Expresion | Expresion / Expresion

Número → Digito

Número → Digito Número

Digito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

La gramática empleada, aunque aparenta ser sencilla, presenta cierta complejidad, debido a doble recursividad, ambigüedad y ramificación entre 4 y 10.

Aparentemente se trata de una gramática sin complicación alguna, pero posee características que la clasifican como un buen ejemplo a analizar. Estas características son:

- Posee doble recursividad, es decir, contiene recursividad a izquierdas y a derechas. Puede observarse este hecho en la regla *Expresion→ Expresion + Expresion*.
- Por las reglas *Expresion → Expresion + Expresion | Expresion – Expresion | Expresion * Expresion | Expresion / Expresion* la gramática contiene ambigüedad. Puede generarse

la misma cadena aplicando derivaciones distintas de los *No terminales* de la parte derecha.

Según el *No Terminal* que se quiera derivar, la ramificación de las posibilidades es diferente. Al analizar un dígito con la regla *Número → Digito*, las opciones viables están entre diez *Terminales*. Dependiendo de cuantos de cada uno de ellos posea la cadena a analizar, el proceso de comprobación será más o menos lento.

Con esta gramática, inicialmente realizamos un análisis con los dos algoritmos implementados en *Naarpe* (CYK y Fuerza Bruta) de palabras generadas por la gramática o no.

Posteriormente se compara el algoritmo CYK implementado en *Naarpe* con el correspondiente de *JFLAP*. La ejecución de cada una de las pruebas se realizará con su homónimo.

9.3.1. Prueba comparando los algoritmos CYK y FB implementados

Prueba no aceptadas

Esta prueba trata de determinar cuánto tiempo tarda cada uno de los algoritmos en determinar una palabra no perteneciente al lenguaje generado por la gramática. Para ello hay dos casos, las peores instancias y las mejores. Las mejores son aquellas en las que al primer carácter de la palabra se sabe que no es perteneciente. Por el contrario, las peores instancias se determinan por el último carácter. Los resultados se muestran a continuación:

NO ACEPTADAS MEJOR INSTANCIA*				
Cadena	**Análisis CYK**		**Análisis FB**	
	Con escritura de fichero	***Sin escritura en fichero***	***Con escritura de fichero***	***Sin escritura en fichero***
/123456789	32	31	0	0
/123456789/123456789	62	47	1	1
/123456789/123456789/123456789	172	172	0	0
/123456789/123456789/123456789/ 123456789	390	390	0	0
/123456789/123456789/123456789/ 123456789/123456789	795	764	0	0
/123456789/123456789/123456789/ 123456789/123456789/123456789	1421	1374	0	0
/123456789/123456789/123456789/ 123456789/123456789/123456789/ 123456789	2310	2294	0	0

Tabla 95. No aceptadas mejor instancia

NO ACEPTADAS PEOR INSTANCIA*				
Cadena	**Análisis CYK**		**Análisis FB**	
	Con escritura de fichero	***Sin escritura en fichero***	***Con escritura de fichero***	***Sin escritura en fichero***
123456789/	63	32	48	32
123456789/123456789/	78	93	842	843
123456789/123456789/123456789/	234	203	10811	10795
123456789/123456789/123456789/ 123456789/	405	390	110995	108546
123456789/123456789/123456789/ 123456789/123456789/	1029	796	**	**
123456789/123456789/123456789/ 123456789/123456789/123456789/	1484	1483	**	**
123456789/123456789/123456789/ 123456789/123456789/123456789/ 123456789/	3106	2341	**	**

Tabla 96. No aceptadas peor instancia

*Los tiempos están expresados en milisegundos.

** Representa que no se ha podido ejecutar en un tiempo razonable.

Prueba aceptadas

Este apartado muestra las pruebas realizadas para tratar de determinar cuánto tiempo tarda cada uno de los algoritmos en determinar una palabra perteneciente al lenguaje. Aquí no hay mejor o peor instancia, el tiempo de cómputo depende únicamente de la longitud de la palabra. Los resultados se muestran a continuación:

INSTANCIAS ACEPTADAS*				
Cadena	**Análisis CYK**		**Análisis FB**	
	Con escritura de fichero	***Sin escritura en fichero***	***Con escritura de fichero***	***Sin escritura en fichero***
12345+6789	31	27	5	0
123456789+123456789	78	63	16	1
123456789+123456789-123456789	187	156	47	46
123456789+123456789-123456789*123456789	359	350	110	94
123456789+123456789-123456789*123456789/123456789	765	733	124	122
123456789+123456789-123456789*123456789/123456789+123456789	1358	1327	172	156
123456789+123456789-123456789*123456789/123456789+123456789-123456789	2217	2201	265	250

Tabla 97. Instancias aceptadas

*Los tiempos están expresados en milisegundos.

Las diferentes pruebas contienen dos vertientes, una con escritura en fichero y la otra sin él. El programa *Naarpe* genera ficheros durante las pruebas, por lo que se ha decidido tomar los tiempos con escritura y sin ella. La escritura influye de forma poco consistente en las mediciones, lo cual hace que los resultados fiables sean los de la versión sin grabación de ficheros.

La conclusión obtenida es que el algoritmo de Fuerza Bruta posee gran dependencia sobre la cadena a analizar no aceptada, variando considerablemente el tiempo dependiendo dónde se encuentre el punto en el que ya no pueda ser generada. En cambio, el algoritmo CYK depende del tamaño de palabra, pero no tiene la dependencia de la peor instancia.

La gramática resultante en FNC generada por el programa implementado para realizar las pruebas con el algoritmo CYK se muestra en el apartado *Anexo II*.

9.3.2. Prueba comparando CYK implementado con JFLAP

Prueba aceptadas

Esta prueba trata de determinar cuánto tiempo tarda cada uno de los algoritmos CYK (tanto de *Naarpe* como de *JFLAP*) en determinar una palabra perteneciente al lenguaje generado por la gramática. Los resultados obtenidos se muestran a continuación:

INSTANCIA ACEPTADA*		
Cadena	**Análisis CYK**	
	Naarpe	**JFLAP**
123456789/123456789	63	137
123456789/123456789/123456789	218	400
123456789/123456789/123456789/123456789	516	984
123456789/123456789/123456789/123456789/123456789	969	1777
123456789/123456789/123456789/123456789/123456789/ 123456789	1704	3015
123456789/123456789/123456789/123456789/123456789/ 123456789/123456789	2870	4502
123456789/123456789/123456789/123456789/123456789/ 123456789/123456789/123456789	4250	6835

Tabla 98. Instancias aceptadas JFLP y CYK

*Los tiempos están expresados en milisegundos.

Comparando el algoritmo CYK de *Naarpe* con el incorporado en la herramienta *JFLAP*, observamos que el de *Naarpe* es capaz de abordar la decisión de palabras razonablemente grandes en tiempos aceptables y además hemos obtenido una notable mejoría en el tiempo de respuesta con respecto a *JFLAP*.

9.4. Gramática de miniProlog

Prolog es un lenguaje de programación el cual compila un código fuente en código de byte. Hemos tomado una gramática orientada a una práctica de la asignatura *Teoría de autómatas y Lenguajes formales*, que genera un subconjunto reducido de este lenguaje y la hemos probado con algoritmo CYK de *Naarpe*. La gramática junto con las palabras de prueba se especifican en el apartado *Anexo I*. Los resultados obtenidos se muestran en la tabla siguiente.

Análisis Sintáctico de Instancias aceptadas*	
Cadena	**Algoritmo CYK**
padre[luis,maria].	46
padre[luis,maria];padre[jose,luis].	63
abuelo[x,y]:padre[x,z],padre[z,y].	62
padre[luis,maria];padre[jose,luis];abuelo[x,y]: padre[x,z],padre[z,y].	250
factorial[0,1].	31
factorial[0,1];factorial[x,y].	62
factorial[0,1];factorial[x,y]:gt[x,0].	78
factorial[0,1];factorial[x,y]:gt[x,0],is[z,min[x,1]].	140
factorial[0,1];factorial[x,y]:gt[x,0],is[z,sub[x,1]], factorial[z,w].	297
factorial[0,1];factorial[x,y]:gt[x,0],is[z,sub[x,1]], factorial[z,w],is[y,mul[x,w]].	438

Tabla 99. Prolog resultados con CYK

*Los tiempos están expresados en milisegundos.

Para una gramática con 61 reglas, el algoritmo implementado proporciona los resultados de los análisis de las cadenas en un tiempo razonable. Esta gramática sería imposible analizarla con el derivador de Fuerza Bruta de *JFLAP* y muy complicada de plantear con su algoritmo de CYK debido al número limitado de *No Terminales* disponibles. Los resultados son satisfactorios.

9.5. Conclusiones globales de todas las pruebas

Después de todas las pruebas realizadas, en conjunto obtenemos una gran mejoría con *Naarpe* respecto a una de las mejores herramientas que hay actualmente en el mercado.

El algoritmo CYK de *Naarpe* tiene un tiempo de ejecución estable, tanto para las palabras no aceptadas como para las que sí, y es capaz de abordar la decisión en tiempos aceptables (inferior al minuto para palabras con tamaños normales). Estos resultados son satisfactorios.

El algoritmo de Fuerza Bruta de la herramienta *Naarpe* posee gran dependencia sobre la cadena a analizar perteneciente a las no aceptadas por la gramática. Poniéndonos en el peor de los casos, el cual es el que realmente hay que tener en cuenta para la complejidad, los resultados

no son tan buenos como los del CYK. En la herramienta este derivador es clasificado como peor que el CYK, pero siempre es factible su uso para derivar palabras de las que se sabe que son generadas por la gramática.

Las pruebas con el algoritmo de Fuerza Bruta de *JFLAP*, siendo triviales, superan el límite de tiempo aceptable. La recursividad, la ambigüedad y la ramificación hacen el estudio más complejo pese a poseer tan solo 11 *Terminales*.

Si comparamos el algoritmo CYK implementado en *Naarpe* con el que posee *JFLAP*, ambos funcionan muy bien para gramáticas en FNC, aunque el CYK de *Naarpe* es más rápido y no tiene el problema de la limitación de los *No terminales*.

10. Especificación del Entorno Tecnológico

Para satisfacer los requisitos establecidos, los objetivos propuestos y lograr un resultado óptimo, se requieren de los recursos necesarios para que éste pueda ser llevado a cabo. Por tanto, en este apartado se tratará el entorno tecnológico del sistema.

10.1. Alternativas para el diseño

La primera decisión a la que nos enfrentamos es si la aplicación se va a desarrollar para escritorio o para web.

Si se desarrolla una aplicación para web, no es preciso tener en cuenta qué sistema operativo tiene el usuario. En cambio, sí requiere disponer siempre de acceso a internet, realizar el código de tal manera que se pueda ver desde distintos buscadores y mantener un servidor en funcionamiento.

Por el contrario, si desarrollamos una aplicación de escritorio, podemos realizar un programa capaz de manejar datos más rápidamente, ya que al acceder directamente a la memoria local del ordenador ganamos en tiempo de ejecución. También se consigue jugar con una mayor creatividad en cuanto a la interfaz. En cambio, al igual que pasa con las distintas versiones para los buscadores de una aplicación web, hay que tener en cuenta el sistema operativo de usuario y realizar un código que pueda ejecutarse en cualquier PC.

Evaluando lo anteriormente mencionado, nos decantamos por una aplicación de escritorio. Necesitaremos manejar los datos referentes a las gramáticas y palabras a analizar de la manera más cómoda y rápida posible, por lo tanto acceder a la memoria local resulta más ventajoso. Tiene que poderse manejar toda la información de las reglas de la gramática pasada por parámetro, añadir las producciones necesarias para dejarla en el formato adecuado, las cadenas a comparar, la generación de las tablas del CYK, etc. Además, teniendo en cuenta los costes que llevaría el mantener un servidor, no estaría compensado por el uso que harán los usuarios de este.

10.2. Tecnologías

Para llevar a cabo el proyecto, una vez que se decida si la aplicación se desarrolla para escritorio o para web, debemos determinar qué tecnología es la más adecuada.

En la universidad aprendemos que cualquier lenguaje puede resolver un determinado problema, pero que hay unos más adecuados que otros dependiendo del planteamiento. Estableciendo el objetivo de este proyecto, los lenguajes que mejor se adaptan a la resolución de los requisitos serían *Java* y *C*. A continuación se plasmará una reseña de cada uno y una comparativa de ambos.

10.2.1. Lenguaje Java

Java es un lenguaje de programación orientado a objetos y una plataforma informática creada por James Gosling y comercializada por primera vez en 1995 por *Sun Microsystems*. Posteriormente, dicha plataforma fue adquirida y comercializada por la compañía *Oracle*.

Gran parte de su sintaxis procede de los lenguajes *C* y *C++*, pero posee una mayor simplicidad que estas dos al eliminar conceptos como el manejo de punteros o la gestión de la memoria. La memoria realiza una función automática, denominada el recolector de basura, que reserva, libera y compacta espacios de memoria sin necesidad de llamar a una función específica en el código.

Java es un lenguaje compilado e interpretado. El código que se genera al compilar contiene bytecodes, que son interpretados por una máquina virtual. Con esto, se consigue la independencia del código respecto de la máquina, ya que se ejecutará en máquinas virtuales que sí son dependientes de la plataforma [32].

La primera implementación de *Java* data de 1995 y pronto los navegadores web incorporaron soporte *Java* para la ejecución de pequeñas aplicaciones interactivas (Applets) [33]. Actualmente este lenguaje se usa en portátiles, centros de datos, consolas para juegos, súper computadoras, teléfonos móviles, Internet, etc [34].

10.2.2. Lenguaje C

C es un lenguaje creado entre los años 1972 y 1973 por un equipo de programadores de los ***Laboratorios Bell de AT&T*** [33].

Dennis Ritchie diseñó e implementó el primer compilador de lenguaje *C*. Este lenguaje se basó en dos lenguajes ***BCPL***, creado por Martin Richards, y ***B***, creado por Ken Thompson.

Originalmente, el estándar de *C* fue el ***K&R C*** [35]. Ese nombre proviene de los nombres de los autores del libro *The C Programming Language*, Brian Kernigham y Dennis Ritchie, el cuál fue durante muchos años la "referencia oficial" del lenguaje [33].

10.3. Comparación de tecnologías

El lenguaje *C* cuenta con la ventaja de contener una estructura importante, los punteros. Estos marcan la principal diferencia con otros lenguajes. Pese a que en la época en la que se creó *C* los punteros fuesen de gran ayuda al programar, en la actualidad, esa ventaja puede hacer más complicado el proceso de programación cuando se trata de proyectos de grandes dimensiones. *Java* no posee punteros externos, es decir, punteros que pueden codificar los usuarios (ya que en la codificación interna sí que posee punteros), pero si tiene es la configuración necesaria para poder realizar una programación orientada a objetos. Comparando

estas dos características y enfocándolas en este trabajo, *Java* gana al facilitar la tarea de programar a gran escala gracias a la programación orientada a objetos.

Otra diferencia es que *C* no necesita intérpretes para ser ejecutado y *Java* sí. Un intérprete necesita menos memoria que un compilador, con lo que los dispositivos que tengan poca memoria, como por ejemplo un Smartphone, les beneficia el uso de un intérprete. En cambio un intérprete es más lento que código compilado. *C* tiene ventaja al ser más rápido, y este proyecto tiene como objetivo la capacidad de realizar análisis con gran velocidad, por lo que en este punto gana *C*. Pero *Java* gana al tener un intérprete y requerir de menor cantidad de memoria. En esta cuestión, ambos lenguajes están muy igualados. Otra ventaja que inclina la decisión hacia *Java* es que los intérpretes permiten una mayor interactividad con el código en tiempo de desarrollo y ayudan al programador a ver los errores sin necesidad de realizar la compilación.

Un beneficio de *Java* es que posee el recolector de basuras. Esto es de gran ayuda al programar ya que el programador no tiene que invocar a una subrutina para liberar memoria, como sí que lo tiene que hacer en *C*.

Con *C* tenemos la ventaja de realizar instrucciones que tengan un control directo sobre el hardware, dado que trata de un lenguaje de bajo nivel con instrucciones de alto nivel. Java no posee esta característica, por lo que en este aspecto vence *C*.

Vistas las características y comparaciones de las tecnologías, realizaremos un breve análisis de estas para escoger la decisión más adecuada para el trabajo. Es fácil, usando el lenguaje *C*, cometer errores con el manejo de los punteros y el solucionar estos problemas, teniendo un límite de tiempo en la creación del proyecto, puede producir retrasos con respecto al tiempo estimado para cada tarea. Además, con *C* hay que llamar a una función para eliminar de memoria la información que no se vaya a volver a utilizar cosa que con *Java*, con el recolector de basuras, eso no ocurre.

Como ya se ha mencionado, la interactividad con el código mientras se desarrolla el programa da a Java una utilidad extra que se aprovecha como un gran beneficio en cuanto al tipo de depuración de errores.

C tiene un control directo sobre el hardware, lo que es beneficioso para un programa que maneje información almacenada en distintas partes. Si nos fijamos en los algoritmos implementados en el proyecto, el algoritmo de derivación por fuerza bruta realiza una generación de un árbol de derivación con crecimiento de su complejidad temporal de tipo exponencial. La cuestión de velocidad, hablando de exponenciales, aporta una mínima ventaja porque llegará un punto en el que sea intratable.

Tomando como referencia el objetivo de este proyecto y el análisis realizado, se escoge como mejor lenguaje de programación *Java*.

10.4. Entorno de desarrollo

Existen muchísimos entornos para cada lenguaje de programación, y cada uno de ellos ofrece al usuario unas determinadas funcionalidades. Al igual que para cada lenguaje, también existen diversos IDEs para *Java*. A continuación se exponen los dos más importantes y se hace una valoración para escoger el mejor de ellos, que será en el que se desarrollará la aplicación.

10.4.1. Eclipse

Eclipse es una plataforma de desarrollo de código abierto multiplataforma basada en *Java*. El software de código abierto es un software lanzado con una licencia que pretende asegurarse de que se les otorguen ciertos derechos a los usuarios. Fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para *Visual Age* aunque ahora se desarrolla por la *Fundación Eclipse*.

Tiene un conjunto de complementos, incluidas las Herramientas de Desarrollo de *Java* (JDT). *Eclipse* se usa como un IDE de *Java* aunque también incluye el Entorno de Desarrollo de Complementos (PDE), que es de interés para los desarrolladores que quieren extender *Eclipse*.

Aunque *Eclipse* está desarrollado en el lenguaje *Java*, incluye complementos para dar soporte a lenguajes de programación como *C/C++* y *COBOL*. El marco de trabajo de *Eclipse* puede también utilizarse como base para otros tipos de aplicaciones que no se relacionen con el desarrollo del software, como los sistemas de gestión de contenido [36].

10.4.2. NetBeans IDE

En junio de 2000, *Sun MicroSystems* creó el proyecto *NetBeans*. Al día de hoy hay disponibles los productos el *NetBeans IDE* y *NetBeans Platform*. Centrandonos en este proyecto hay que destacar *NetBeans IDE*. *NetBeans IDE* (siendo IDE las siglas de *entorno de desarrollo integrado* inglés) es un entorno de desarrollo para que los programadores puedan escribir, compilar, depurar y ejecutar programas. Está escrito en *Java* pero puede servir para desarrollar en *Java*, *JavaScript*, *PHP*, *Python*, *Ruby*, *Groovy*, *C/C++*, *Scala*, *Clojure*, etc Es un producto libre y gratuito sin restricciones de uso [37].

10.4.3. Elección

A continuación realizaremos una comparación entre los dos entornos anteriores. Limitaremos la comparación en aspectos menos profundos y todo ello centrándonos en la aplicación de este proyecto a realizar. Esta restricción es debido a que dichas herramientas son tan completas que se podría dedicar un documento entero a compararlas y no es el objetivo principal de este trabajo.

Un problema a tener en cuenta es la cuestión económica, es decir, evaluar qué precio compensaría pagar por una herramienta en cuanto al tiempo de desarrollo del proyecto. Con los dos entornos de desarrollo nos ahorramos ese problema. Ambos son gratuitos, por lo que económicamente ninguno de ellos tiene ventaja sobre el otro.

Otro factor a tener en cuenta es la curva de aprendizaje. Los componentes del equipo de este trabajo han trabajado con anterioridad con *Eclipse*. Esto supone un ahorro en tiempo de producción ya que se eliminaría parte del tiempo que conlleva adaptarse a una nueva herramienta. En cambio, nadie del equipo ha trabajado con *NetBeans* y si se escogiese, supondría añadir un tiempo extra al trabajo para la familiarización con el entorno de desarrollo. Este incremento en el tiempo a dedicar a conocer la herramienta, supondría un decremento en el tiempo establecido para la realización de pruebas, validación y mejora de la aplicación.

Tomando como referencia las funcionalidades, no hay gran diferencia entre *NetBeans* y *Eclipse*. Ambos poseen las mismas características, salvo que con *NetBeans* posee funcionalidades que se pueden ejecutar con menor número de acciones de desplazamiento en el menú de comandos. Otra facilidad en la funcionalidad es la interacción del usuario con el entorno, permitiéndole generar código predeterminado tan solo arrastrando el botón de la funcionalidad y soltándolo en la parte de la pantalla de programación. *NetBeans* posee esta facilidad de uso y velocidad a la hora de codificar para realizar aplicaciones con interfaz gráfica.

Esta ventaja se puede convertir en ocasiones en una desventaja. Al escoger una opción que genere de forma automática el código necesario, trae consigo la creación de código basura (habrá opciones que el usuario no desee producir). Esto hace que el código creado no sea el más óptimo.

Cuando se realizar la instalación de *NetBeans*, este incorpora gran número de plugins en comparación con los de *Eclipse*. A priori parece una ventaja, pero el objetivo de este proyecto es realizar una aplicación de escritorio. En la ejecución del trabajo no se van a usar todos los plugins del entorno *NetBeans*. Es cierto que se pueden eliminar, pero requiere conocer mejor cuáles son necesarios para el programa y un esfuerzo adicional de eliminar los que no.

Hay que destacar que *Eclipse* tiene desventaja en sistemas operativos *Linux*. Suele tener problemas en estos sistemas operativos al bloquearse y no responde adecuadamente a las acciones generadas por el usuario. En este caso tiene ventaja *NetBeans* al poder trabajar en el sistema operativo que se desee.

Evaluando las características y comparaciones anteriormente mencionadas, es *Eclipse* la mejor herramienta para este proyecto. Las razones que han llevado a cabo tomar esta decisión son: no hay que realizar ningún pago para su utilización, la herramienta no genera funcionalidades no deseadas (lo que implica conocer al 100% el código implementado y detectar mejor los errores cometidos) y antes de empezar el proyecto el equipo de desarrollo posee conocimientos sobre la aplicación. También nos decantamos por *Eclipse* ya que vamos a trabajar en el sistema operativo *Windows*. Los problemas con Linux no nos afectaran.

11. Aspectos legales

Los derechos de autor son los derechos que tienen los creadores de obras (personas físicas o jurídicas) sobre las mismas. No es lo mismo que Copyright, término anglosajón para designar solamente los derechos de explotación de una obra. Los derechos de autor no hacen referencia a derechos morales [38].

El software puede estar protegido por los derechos de autor asemejándose a una obra literaria. Estudiando en qué condiciones se podría adquirir encontramos lo siguiente puntos a tener en cuenta para que lo concedan:

a) Creación humana. El programa debe ser resultado de la capacidad intelectual del autor.
b) Expresada por cualquier medio o soporte. La obra debe estar contenida en un medio tangible o intangible. Por ejemplo un CD-ROM, etc.
c) Original. Esto quiere decir que la obra haya sido creada originalmente por el autor y no copiada de otras obras, y que debe contener la cantidad suficiente de creatividad para que no se la pueda considerar como algo trivial.

Hay que destacar que la protección se establece sobre la forma, continente y expresión de la idea, pero no sobre el contenido.

El software a crear, como hemos ido explicando en los apartados anteriores, se trata de la implementación de unos algoritmos para la resolución de pertenencia de palabras a cierto tipo de gramáticas. Los algoritmos, pese a estar implementados desde 0, ya tienen un registro sobre quién los descubrió y hay que investigar si el usarlos en un programa no autorizado por dichos creadores acarrea algún problema de derecho de autor. Investigando esta cuestión, se ha visto que no se patenta el algoritmo con el que se resuelve un programa informático, si no que se patenta el producto resultante. Con ello se aclara que no habría ningún problema con el derecho de autor sobre el uso de distintos algoritmos [39]. Así mismo, del proyecto no podría patentarse los algoritmos creados para la resolución del problema.

Centrándonos ahora en la plataforma elegida para la creación y el lenguaje escogido, como hemos mencionado en apartados anteriores, Eclipse es una plataforma de desarrollo de código abierto y a la vez un software de código abierto. La plataforma tampoco tiene ningún problema legal.

Si revisamos el lenguaje de implementación, la mayoría del código en Java tiene la licencia GNU (General Public License) que garantiza a los usuarios finales el poder usar, estudiar, compartir, copiar y modificar el código del software. La creación de código en Java es libre, por lo que tampoco hay impedimento legal en su uso.

La herramienta creada, está orientada a ayudar en la docencia a los profesores y a los alumnos. Su uso y distribución queda en manos de la Universidad Carlos III de Madrid. El prototipo *software*, al ser realizado dentro de un Trabajo de Fin de Grado la autoría la tiene la universidad, y por tanto, al ser un recurso propio de esta, está a libre disposición de los profesores y alumnos del centro.

En conclusión, este proyecto no se encuentra con ninguna restricción legal ni se le puede atribuir el derecho de autor al terminar el proyecto. Tampoco tenemos ningún problema legal en cuento a la creación del mismo ni a la distribución por parte de la Universidad Carlos III de Madrid.

12. Conclusiones y líneas futuras

En este apartado se realizará una síntesis sobre el desarrollo del proyecto y las conclusiones obtenidas a lo largo del mismo, añadiendo un planteamiento de líneas futuras de mejora.

12.1. Conclusiones

En la asignatura *Teoría de Autómatas y Lenguajes Formales* (2º curso del Grado en Ingeniería Informática de la Universidad Carlos III de Madrid) se plantean numerosos ejercicios de lenguajes y gramáticas, que inicialmente el alumno debe desarrollar "manualmente". No existen muchas herramientas software que ayuden al alumno en la comprensión de los conceptos y/o cómo resolver tales ejercicios que se plantean en la asignatura. Un programa utilizado es JFLAP, pero ofrece y posee unos recursos computacionales limitados. Un ejemplo de restricción es que tan sólo se puede dispones en una gramática entre 10 y 14 *No terminales* útiles si se va a realizar conversión a Forma Normal de Chomsky. Es cierto que está definido el máximo en 26 *No terminales*, comprendidos entre los valores de las letras A-Z en mayúsculas, pero la conversión a FNC puede consumir bastantes de ellos.

Abordar problemas medianos, como el de un análisis sintáctico con un lenguaje elemental, requieren el uso de múltiples *No terminales*, por lo que la herramienta JFLAP no es adecuada para realizar operaciones con ellos. En la posterior asignatura de *Procesadores del Lenguaje* se abordan aplicaciones de análisis sintáctico pero basadas en gramáticas más complejas, lo cual requiere de conocimientos muy avanzados que exceden los que se pueden incluir en el estudio de *Teoría de Autómatas y Lenguajes Formales*. Los alumnos disponen de muchas herramientas para la asignatura posterior, pero no se han podido adquirir los conocimientos adecuados en la asignatura que sienta las bases.

La falta de herramientas, o ciertas deficiencias de las mismas, impulsó a crear una aplicación que facilite y amplíe el proceso de aprendizaje (restringido a gramáticas de tipo 2). Esta herramienta también puede ser de especial utilidad en el ámbito profesional cuando es necesario el diseño una gramática y desarrollo de sistemas dependientes de dicha gramática. Así pues, el objetivo es crear una aplicación útil para el refuerzo a la hora de aplicar las reglas de derivación (i.e. producciones de la gramática) que dan lugar a las sentencias de un lenguaje.

Por ello, se planteó por parte de los tutores/directores estudiar y aplicar el algoritmo CYK, desarrollando de forma completa el código fuente de este algoritmo.

Concretamente, el software desarrollado, llamado *Naarpe*, determina la pertenencia o no de palabras dentro de gramáticas en Forma Normal de Chomsky (mediante el algoritmo CYK), en Forma Normal de Greibach o directamente sin transformación (mediante un algoritmo de derivación por Fuerza Bruta). Además, obtiene los árboles de derivación en caso

de que estas sí perteneciesen a la gramática. También realizará transformaciones de una gramática para obtener su Forma Normal de Chomsky o la Forma Normal de Greibach.

Se ha conseguido desarrollar todos los objetivos propuestos inicialmente por lo que el trabajo realizado ha sido satisfactorio. Además, se probó los algoritmos creados con la herramienta JFLAP para comprobar si se han conseguido mejoras con respecto a esta herramienta usada en las cases de *Teoría de Autómatas y Lenguajes Formales.*

La aplicación *Naarpe* ha mejorado ciertas limitaciones que tenían las herramientas actuales; por ejemplo la limitación cuantitativa a la hora de definir tanto *Terminales* como *No terminales*, o la falta de un programa que transformase a Forma Normal de Greibach.

Realizando estudios con los mismos algoritmos que posee la herramienta JFLAP y el programa *Naarpe*, se ha llegado a conclusiones muy favorables. Con el algoritmo CYK realizado, podemos analizar palabras en menor tiempo que *JFLAP* y además eso conlleva a poder determinar cadenas con longitudes mayores.

Comparando los distintos algoritmos de la herramienta *Naarpe*, determinamos que el algoritmo CYK tiene un tiempo de ejecución estable, tanto para las palabras no aceptadas como para las que sí, y es capaz de abordar la decisión en tiempos aceptables. En cambio, el algoritmo de Fuerza Bruta posee gran dependencia sobre las palabras no pertenecientes, analizándolas con una variación considerable de tiempo dependiendo de donde se encuentre el símbolo *No terminal* que provoque la no pertenencia al lenguaje.

Todos los algoritmos han sido creados desde cero. No se ha basado en un código ajeno el cual mejorar. Las únicas nociones son los documentos encontrados sobre el CYK, los cuales contienen un pseudocódigo, y lo estudiado en la asignatura de *Teoría de Autómatas y Lenguajes Formales* sobre los demás algoritmos. El grado de dificultad de comprensión de los diversos algoritmos fue:

- Algoritmo CYK.
- Algoritmo de transformación a Forma Normal de Greibach.
- Algoritmo de derivación por Fuerza Bruta.

Durante la implementación, surgió un problema de comprensión del análisis del algoritmo CYK. Pocos libros muestran ejemplos detallados del desarrollo, los cuales son necesarios para comprobar si los ejercicios realizados son coherentes y dan resultados correctos. Sin embargo, en cuanto a la implementación fue el algoritmo más sencillo y que menos problemas ha dado a lo largo de su desarrollo. Además, pese a ser más fáciles de entender, el algoritmo de derivación por Fuerza Bruta y el algoritmo de transformación a Forma Normal de

Greibach fueron los que más tardaron en implementarse debido a la necesidad de programar la eliminación de la recursividad a izquierdas.

Por último, por el objeto del Trabajo Fin de Grado, se han afianzado y ampliado los conocimientos acerca de los lenguajes y las gramáticas formales. También ha sido un reto personal. Enfrentarse a un proyecto que abarca tantas tareas diferentes (por ejemplo sacar requisitos, casos de uso, llevar a cabo una implementación, etc.) y completarlas todas desde cero, ha supuesto un desafío académico y profesional. Además, la mayoría de los proyectos académicos que he realizado han sido con otros compañeros y de menor envergadura. Llegar a terminar este trabajo tan diferente, y a la vez parecido, a los que he realizado hasta ahora y darme cuenta de que soy capaz de ello, ha sido muy gratificante.

Me he dado cuenta de todo el proceso y trabajo que conlleva realizar un proyecto así. Aunque hayan sido muchas horas dedicadas, mucho esfuerzo, es muy satisfactorio saber que el resultado final llega a un usuario, que no se queda en un archivo sin usar, que pueda ayudar a la docencia.

12.2. Líneas futuras

Aunque la herramienta *Naarpe* cumple con su cometido principal, es cierto que se podría mejorar con vistas a ofrecer más funcionalidades y perfeccionar determinados aspectos. Los trabajos futuros que se pueden considerar se describen a continuación.

La primera mejora consistiría en permitir el análisis de gramáticas que contengan la generación de la palabra vacía desde el axioma principal. Es una mejora importante. Actualmente *Naarpe* no permite el análisis de ninguna regla que genere lambda desde el axioma principal, problema que comparte con otras herramientas como *JFLAP*.

En relación a las gramáticas, otra mejora consistiría en permitir que el programa analice la gramática pasada por parámetro sin necesidad de realizar cambios en esta en la opción de Fuerza Bruta. Beneficiaría a la herramienta porque se podría analizar cualquier clase de gramática.

Los análisis realizados por el programa son sobre gramáticas con la Forma Normal de Chomsky y Forma Normal de Greibach. Sería interesante introducir opciones para gramáticas LL(1), LR(0), SLR, LR(1) y LALR.

También existe la posibilidad del desarrollo de una interfaz gráfica, que ayude tanto en el aprendizaje del alumno como en la labor docente del profesor.

Se ha realizado un estudio con usuarios. Las respuestas obtenidas en la parte de opiniones son muy interesantes y, las que no hayan sido contempladas, pueden añadirse a la lista de trabajos futuros.

13. Bibliografía

[1] MORENO VELO, Francisco José (2011) Francisco José Moreno Velo. Transparencias del programa teórico de la asignatura Teoría de Autómatas y Lenguajes Formales. Escuela Técnica Superior de Ingeniería de Huelva. «*Tema 1: Introducción*». Accessible en: http://mascvuex.unex.es/ebooks/sites/mascvuex.unex.es.mascvuex.ebooks/files/files/file/TeoriaAutomatas_9788469163450.pdf. Último acceso Septiembre 2014.

[2] JURADO MÁLAGA, Elena (2008) «Teorías de Autómatas y Lenguajes Formales» *Colección manuales uex – 55 (E.E.E.S.)* ISSN: 1135-870-X. ISBN: 978-84-691-6345-0. Accessible en: http://campusvirtual.unex.es/ebooks/files/file/TeoriaAutomatas.pdf. Último acceso Septiembre 2014.

[3] GÖDEL, K. (1931): «Über formal unentscheidbare Sätze der Principia Mathematica and verwandter Systeme», I. *Monatshefte für Mathematik und Physik*, 38, pp. 173-198.

[4] ALFONSECA MORENO, Manuel; DE LA CRUZ ECHEANDÍA, Marina; ORTEGA DE LA PUENTE, Alfonso; PULIDO CAÑABATE, Estrella. (2006) «*El libro Compiladores e intérpretes: teoría y práctica» Universidad Autónoma de Madrid*. Departamento de Ingeniería Informática. Editorial: Pearson Pretentice Hall. ISBN: 84-205-5031-0.

[5] SHANNON, C. (1938): «*A symbolic analysis of relay and switching circuits*», *Transactions American Institute of Electrical Engineerrs*, vol. 57, pp. 713-723.

[6] CHOMSKY, N. (1956): «*Three models for the description of language*», *IRE Transactions on Information Theory*, 2, pp. 113-124.

[7] CHOMSKY, N. (1959): «*On certain formal properties of grammars*», *Information and Control*, 1, pp. 91-112

[8] Real Academia Española. Accesible en: http://www.rae.es/recursos/diccionarios/drae. Último acceso Septiembre 2014.

[9] GALLEGO, Angel J. (2008): «*La jerarquía de Chomsky y la facultad del lenguaje: consecuencias para la variación y la evolución*» *Teorema: Revista internacional de filosofía*. ISSN: 0210-1602. Vol. 27, Número 2, pp. 47-60. Accesible en: http://dialnet.unirioja.es/servlet/articulo?codigo=2580734. Último acceso Septiembre 2014.

[10] PARDO VASALLO, Luis Miguel; GÓMEZ PÉREZ, Domingo. «*La Jerarquía de Chomsky. Apuntes sobre la Complejidad*» Universidad de Cantabria. Accesible en: http://ocw.unican.es/ensenanzas-tecnicas/teoria-de-automatas-y-lenguajes-formales/material-de-clase-nuevo/nuevo/1-4_Jerarquia_Chomsky.pdf. Último acceso Septiembre 2014.

[11] POLANCO FERNÁNDEZ, Daniel Francisco (2000). Proyecto fin de carrera: «*Evaluación y mejora de un sistema automático de análisis sintagmático*». *Capítulo 5. Analizador CYK.* Grupo de Tecnología del Habla y del GRIDS del Departamento de Ingeniería Electrónica de la ETSI Telecomunicación de la Universidad Politécnica de Madrid. pp. 50-60. Accesible en: http://lorien.die.upm.es/juancho/pfcs/DPF/capitulo5.pdf. Último acceso Septiembre 2014.

[12] GÓMEZ, Tomás. Área de lenguajes y sistemas informáticos de la Universidad de Burgos. *Ejercicio Guía para obtener FNC y FNG*. Accesible en: http://www.alumnos.inf.utfsm.cl/~dcontard/ili-255/documents/guia_FNC_FNG.pdf. Último acceso Septiembre 2014.

[13] DE CASTRO KORGI, Rodrigo. Universidad Nacional de Colombia, sede Bogotá. Facultad de ciencias. Apuntes de curso de Teoría de la Computación. Accesible en: http://www.virtual.unal.edu.co/cursos/ciencias/2001018/lecciones/Cap4ss11.pdf. Último acceso Septiembre 2014.

[14] VISWANATHAN, M. *CS 373 - Theory Of Computation - University Of Illinois, Urbana Champaign Study Resources*. University Of Illinois, Urbana Champaign. Accesible en: https://courses.engr.illinois.edu/cs373/sp2009/lectures/lect_15.pdf. Último acceso Septiembre 2014.

[15] DE CASTRO KORGI, Rodrigo. *Apuntes de curso de Teoría de la Computación.* Universidad Nacional de Colombia, sede Bogotá. Facultad de ciencias. Accesible en: http://www.virtual.unal.edu.co/cursos/ciencias/2001018/lecciones/Cap4ss14.pdf. Último acceso Septiembre 2014.

[16] PARDO VASALLO, Luis Miguel. *Algoritmos de Parsing Genéricos. El Análisis CYK*. Apuntes Teoría de autómatas y lenguones formales. Universidad de Cantabria. Accesible en: http://ocw.unican.es/ensenanzas-tecnicas/teoria-de-automatas-y-lenguajes-formales/material-de-clase-nuevo/nuevo/4-4CYK.pdf. Último acceso Septiembre 2014.

[17] AGUILAR ALCONCHEL, Miguel Ángel (2004): *Chomsky La gramática generativa*. Revista digital "Investigación y educación". Accesible en: http://www.csub.edu/~tfernandez_ulloa/spanishlinguistics/chomsky%20y%20la%20gramatica%20generativa.pdf. Último acceso Septiembre 2014.

[18] Pijama Surf (2012): *Noam Chomsky sobre los errores paradigmáticos en el desarrollo de la Inteligencia Artificial*. Accesible en: http://pijamasurf.com/2012/11/la-inteligencia-artificial-es-esencialmente-conductista-pero-nuestro-cerebro-no-es-tan-simple/. Último acceso Septiembre 2014.

[19] MOLINA-LOZANO, Herón (2010): *A Fast Fuzzy Cocke-Younger-Kasami Algorithm for DNA and RNA Strings Analysis*. Centro de Investigación en Computación del Instituto Politécnico Nacional México DF. Accesible en: http://link.springer.com/chapter/10.1007%2F978-3-642-16773-7_7. Último acceso Septiembre 2014.

[20] Oncina, J. *The Cocke-Younger-Kasami algorithm for cyclic strings*. Universidad de Alicante. Accesible en: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&tp=&arnumber=546859&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D546859. Último acceso Mayo 2015

[21] RODGER, S; FINLEY, T. JFLAP. *An Interactive Formal Languages and Automata Package, Duke University*. Accesible en: http://www.jflap.org/. Último acceso Septiembre 2014.

[22] *APLICACIONES DE LA REGULAR CONSTRAINT*. IGMRC. Accesible en: http://igmrc.diegoriquelme.cl/index.php/articulos/soluciones. Último acceso Mayo 2015.

[23] SCHMITZ, Lothar. *Jaccie Homepage - a visual compiler-compiler for educational purposes*. Accesible en: https://m.unibw.de/rz/dokumente/fakultaeten/getFILE?fid=395143&tid=fakultaeten. Último acceso Septiembre 2014.

[24] SCHMITZ, Lothar. *Visual Syntax Tools*. Accesible en: http://www2.cs.unibw.de/Tools/Syntax/english/. Último acceso Mayo 2015.

[25] *Plan 9 Manual*. Laboratorios Bell y Alcatel-Lucent. Informática Centro de Investigación de Ciencias, Murray Hill. Accesible en: http://plan9.bell-labs.com/magic/man2html/1/lex. Último acceso Mayo 2015.

[26] DONNELLY, C*harles; STALLMAN, Richard. «The Lex & Yacc Page»*. Accesible en: http://dinosaur.compilertools.net/. Último acceso Mayo 2015.

[27] NIEMANN, Tom. *«LEX & YACC TUTORIAL»*. epaperpress. Accesible en: http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf. Último acceso Mayo 2015.

[28] *Plan 9 Manual*. Laboratorios Bell y Alcatel-Lucent. Informática Centro de Investigación de Ciencias, Murray Hill. Accesible en: http://plan9.bell-labs.com/magic/man2html/1/yacc. Último acceso Mayo 2015.

[29] BÉJAR HERNÁNDEZ, Rubén. *«Introducción a Flex y Bison»*. Departamento de Informática e Ingeniería de Sistemas. Universidad de Zaragoza. Accesible en: http://webdiis.unizar.es/asignaturas/LGA/material_2004_2005/Intro_Flex_Bison.pdf. Último acceso Mayo 2015.

[30] *Computer science alumni*. Computer Science Department at the University of California, Riverside. Accesible en: http://alumni.cs.ucr.edu/~lgao/teaching/bison.html. Último acceso Mayo 2015.

[31] Universidad Carlos III de Madrid. *Normativa sobre la organización y evaluación de la asignatura "Trabajo Fin de Grado", aprobada por el consejo de gobierno en sesión de 17 de Junio de 2010*. Accesible en: http://e-archivo.uc3m.es/bitstream/handle/10016/15927/NormativaTrabajoFin%20de%20Grado.pdf?sequence=1. Último acceso Mayo 2015.

[32] Oracle Corporation. *Java. ¿Qué es Java y para qué es necesario?* Accesible en: https://www.java.com/es/download/faq/whatis_java.xml. Último acceso Septiembre 2014.

[33] AmericaTI. *Ventajas y Desventajas: Comparación de los Lenguajes C, C++ y Java*. Accesible en: http://www.americati.com/doc/ventajas_c.pdf. Último acceso Septiembre 2014.

[34] BELMONTE FERNÁNDEZ, Oscar. *Introducción al lenguaje de programación Java. Una guía básica*. Accesible en: http://www3.uji.es/~belfern/pdidoc/IX26/Documentos/introJava.pdf. Último acceso Septiembre 2014.

[35] KERNIGHAN, Brian W. *Programming in C: A tutorial*. Accesible en: http://www.lysator.liu.se/c/bwk-tutor.html Último acceso Septiembre 2014. Último acceso Septiembre 2014.

[36] IBM. Developer Works. *Iniciándose en la plataforma Eclipse*. Accesible en: http://www.ibm.com/developerworks/ssa/library/os-ecov/. Último acceso Septiembre 2014.

[37] Netbeans. *Bienvenido a NetBeans y www.netbeans.org*. Accesible en: https://netbeans.org/index_es.html Último acceso Septiembre 2014.

[38] Biblioteca de la Universidad de Alcalá. *Propiedad y derechos de autor*. Accesible en: http://www.uah.es/biblioteca/ayuda_formacion/pintelectual.html. Último acceso Junio 2015.

[39] GALLEGO RODRÍGUEZ, Manuel; MARTÍNEZ RIBAS, Manuel; RIUS SANJUÁN, Judit; BAIN Malcolm**.** *Aspectos legales y de explotación del software libre Parte I*. UOC

Formación de postgrado. Accesible en: http://www.uoc.edu/masters/oficiales/img/908.pdf. Último acceso Junio 2015.

14. Glosario de términos

- **Software**: conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora.
- **Windows:** sistema operativo desarrollado por Microsoft.
- **.jar:** (siglas del inglés Java ARchive). Archivo que permite ejecutar aplicaciones escritas en lenguaje Java.
- **.txt:** archivo compuesto por texto sin formato, es decir, sólo lo forman caracteres.
- **Main:** método principal de una clase en un archivo Java.
- **ArrayList:** lista de elementos que disminuye o aumenta dinámicamente según se introduzcan o eliminen elementos.
- **While:** sentencia que permite la iteración de estructuras que contiene en su interior.
- **for:** sentencia que permite ejecutar un conjunto de sentencias un número determinado de veces. fijado al principio del bucle y funciona por tanto como un contador.
- **int:** tipo de dato (entero) de 32 bits complemento a dos.
- **String:** tipo de dato que se utiliza como soporte especial para cadenas de caracteres.
- **Switch:** sentencia de un anidamiento múltiple de instrucciones if/else.
- **Java:** lenguaje de programación y primera plataforma informática creada por Sun Microsystems en 1995.
- **Clase:** es la unidad fundamental de programación en Java. Es una "plantilla" que describe un conjunto de objetos con atributos y comportamiento similares.
- **Método:** subrutina que define la interfaz de una clase, sus capacidades y comportamiento.
- **Gramática:** conjunto de reglas que definen un lenguaje.
- **Servidor:** aplicación software en constante ejecución capaz de atender las peticiones de un cliente y devolverle una respuesta.
- **Terminal:** símbolo perteneciente al lenguaje generado por una gramática.
- **No terminal:** símbolo auxiliar para la definición de la gramática.
- **Algoritmo:** Conjunto sistemático de operaciones las cuales permiten realizar los cálculos necesarios para la obtención de la solución de un problema.
- **Backtracking:** técnica algorítmica de resolución de problemas mediante una búsqueda sistemática de soluciones.
- **Stakeholder:** término inglés que hace referencia a la persona interesada en el proyecto.
- **Token:** componente léxico o cadena de caracteres que tiene un significado para un lenguaje concreto.

15. Manual de usuario

En este manual se explicarán todos los conceptos y acciones que se necesiten saber para el correcto funcionamiento de la aplicación.

Índice

1. Ejecución

Para poder inicializar el ejecutable .jar hay que acceder desde la consola al directorio en el que se encuentre. Una vez situados en él, hay dos formas de introducir los comandos de inicio.

Si quieres introducir desde el principio la ruta de los ficheros que utiliza el programa, los comandos deben seguir la siguiente estructura.

java –jar Naarpe.jar Ruta_fichero_gramáticas.txt Ruta_fichero_cadenas.txt

Por el contrario, si decides introducir los ficheros mediante los pasos que va indicando el programa, los comandos a introducir son los siguientes:

java –jar Naarpe.jar

Escogiendo los comandos en los que se introducen los ficheros, si estos están introducidos correctamente, aparecerá directamente el comienzo del programa y a continuación el menú de inicio del programa. En caso de escoger el comando que no necesita las rutas, cuando se introduzcan aparecerá los siguientes mensajes.

```
**********************************************************************
*                            INICIO                                  *
*                                                                    *
* Antes dee comenzar, se necesita la ruta de los ficheros necesarios para   *
* que el programa funcione. Escribalo segun se vaya preguntando.             *
* No se preocupe, puede modificarlos mas adelante cuando desees.             *
*                                                                    *
**********************************************************************

Introduzca la ruta del fichero .txt donde se encuenta la gramatica y pulse ENTER
.
```

```
Introduzca la ruta del fichero .txt donde se encuentan las cadenas a analizar y
pulse ENTER.
```

Después de cada uno de estos mensajes, debe introducir la ruta completa en la cual se encuentra el fichero que pregunte el programa. Un ejemplo de una ruta posible es:

c:/Users/Desktop/… /Gramatica.txt

Comprobado que los ficheros existen y se pueden abrir correctamente, aparecerá el menú de inicio.

```
Comienza el programa...

**********************************************************************
*                PROGRAMA GESTOR DE GRAMATICAS                       *
*                                                                    *
* La lista de las acciones que puede realizar es la siguiente:       *
* 1. Comprobar palabra en gramatica Forma Normal de Chomsky (algoritmo CYK)  *
* 2. Obtener la Forma Normal de Chomsky de una gramatica             *
* 3. Obtener la Forma Normal de Greibach de una gramatica            *
* 4. Comprobar palabra en gramatica Forma Normal de Greibach         *
* 5. Comprobar palabra mediante algoritmo de fuerza bruta            *
* 6. Cambiar la ruta de los ficheros de entrada                      *
* 7. Salir                                                           *
*                                                                    *
**********************************************************************
Introduzca el numero de la accion que desea realizar con la gramatica proporcion
ada y pulse ENTER.
```

Cuando se muestre el menú por pantalla, el usuario deberá introducir la opción deseada y pulsar *Enter*. Siempre que inserte una, el programa comprobará si lo escrito pertenece a una de las opciones disponibles. En caso de dar un valor inválido, el sistema imprimirá *OPCION INVALIDA*, mostrará el menú y volverá a pedir la introducción de la opción.

Todas las opciones, menos *Salir*, pueden ejecutarse tantas veces como se desee.

Las acciones de cada uno de los apartados se explican en el apartado *3. Acciones del programa gestor de gramáticas* de este manual.

2. Documentos iniciales

El programa solo se necesita un fichero en el cual esté descrita la gramática a estudiar y otro que contenga la palabra o palabras a analizar por dicha gramática. Estos ficheros *.txt* deben seguir una serie de características que se describen y explican a continuación.

2.1. Cómo escribir el fichero para la gramática

Este fichero debe contener la lista de reglas y la definición de cuáles son los *Terminales y No terminales* de dicha gramática y la definición de la palabra vacía. El nombre de este fichero no es relevante para que el programa funcione. Lo que sí es importante es la estructura que se debe seguir para definir cada una de las partes, y por ello se detalla con precisión a continuación.

Se debe empezar introduciendo la lista de *Terminales* y *No terminales*. Para ello se denomina en el fichero con ***LISTA TERMINALES:*** y ***LISTA NO TERMINALES:*** respectivamente. Esta forma de definirlo se tiene que seguir estrictamente. Tanto *LISTA TERMINALES:* como *LISTA NO TERMINALES:* tienen que estar definidos en el fichero. Tampoco se puede poner dos *LISTA TERMINALES:* o dos *LISTA NO TERMINALES:*. Si se da alguno de estos casos no se puede leer el fichero correctamente.

El orden de definir primero los *Terminales* o los *No terminales* es indiferente, pero ambos tienen que aparecer al principio del fichero. Tanto los *Terminales* como *los No terminales* pueden definirse en distintas líneas. Un ejemplo de esta explicación se muestra en las siguientes imágenes.

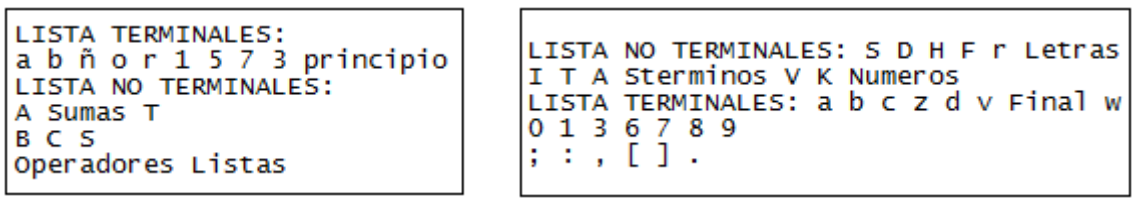


Ilustración 16. Terminales/no terminales

Para diferenciar los *Terminales* entre si, al igual que los *No terminales*, todos ellos deben ir separados por, como mínimo, un espacio. Como se puede observar en las imágenes, la definición de estos no tiene por qué ser con un solo carácter. Tampoco tienen por qué estar definidos estrictamente con minúsculas o mayúsculas.

Es muy importante tener en cuenta son los siguientes avisos.

AVISOS

- Después de la definición de ***LISTA TERMINALES:*** y ***LISTA NO TERMINALES:*** se tiene que poner un espacio.
- No se puede definir lo mismo en los terminales y no terminales, es decir, si se define ***A*** en los No terminales no se puede poner ***A*** en los Terminales. El programa no puede distinguir cuando es uno y cuando otro. En cambio, fijándonos en este caso, si se podría definir ***a*** dentro de los no terminales. Si que se distingue entre mayúsculas y minúsculas.
- No se puede usar el carácter **/** así como tampoco se reconoce un salto de línea, un espacio o un tabulador como terminal o no terminal. Si se desea introducir estos elementos en la gramática se puede usar ***t***, ***r*** y ***n*** o dar un nombre a ese tipo de caracteres. Por ejemplo:

LISTA TERMINALES: tabulador Salto_de_línea espacio BarraLateral

Cuando se tengan bien definidos los *Terminales* y *No terminales*, definimos cual será la palabra vacía. La definición comienza siempre introduciendo ***PALABRA VACIA:***. Al igual que con las otras definiciones, se tiene que seguir siempre este formato establecido. Hay gramáticas que no contienen la palabra vacía pero para que el programa funciones se tiene que definir OBLIGATORIAMENTE aunque luego no aparezca en las reglas. Se debe nombrar con cualquier valor distinto de los *Terminales* y los *No terminales*. También hay que tener en cuenta los siguientes avisos.

AVISOS

- Después de la definición de ***PALABRA VACIA:*** se tiene que poner un espacio y a continuación la denominación de la palabra vacía.

PALABRA VACIA:(espacio)Nombre_Palabra_vacía

- No se puede definir la palabra vacía igual que un terminal o no terminal. El programa no puede distinguir cuando es uno y cuando. También hay que tenerlo en cuenta aunque no aparezca la palabra vacía en las reglas.
- No se puede usar el carácter / así como tampoco se reconoce un salto de línea, un espacio o un tabulador como terminal o no terminal para definirla. Si se desea introducir estos elementos en la gramática, se puede usar ***\t***, ***\r*** y ***\n*** o dar un nombre a ese tipo de caracteres. Por ejemplo:

LISTA TERMINALES: tabulador Salto_de_línea espacio BarraLateral

Por último se define la gramática. Para ello hay que introducir ***GRAMATICA:*** al inicio de la definición. A continuación, se inserta un espacio y ya se puede definir las reglas. Por ejemplo:

GRAMATICA:(espacio)Lista_de_reglas

Cada una de las reglas hay que definirlas con un salto de línea. Lo que puede ser a gusto del usuario es si después de *GRAMATICA:* defines una regla o pones un salto de línea y las defines después. Esta explicación se ve claramente en la imagen *Ilustración 2. Gramáticas* que aparece posteriormente.

También puedes definir las reglas separadas por varios saltos de línea. El programa detectará como regla aquellas líneas que tengan contenido siguiendo la estructura de las reglas.

La estructura de las reglas es *No Terminal*, espacio, *->*, espacio y los *No terminales* y *Terminales* que la formen separados cada uno de ellos por un espacio. Por ejemplo:

T -> b K P E

El orden de definición de las reglas es indiferente, exceptuando la primera. La primera regla SIEMPRE debe ser una que la genere el axioma principal. Un ejemplo de esta explicación se muestra en las siguientes imágenes.

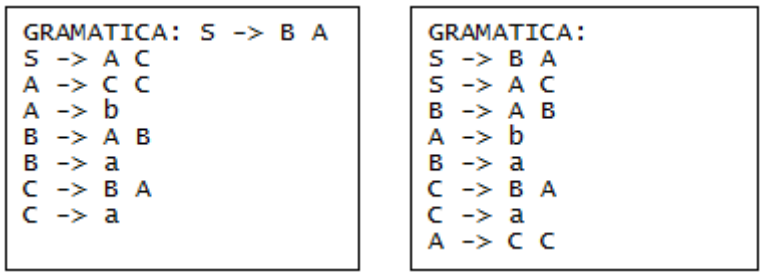


Ilustración 17. Gramáticas

Como ha ocurrido en las partes anteriores, aquí también hay que tener en cuenta los siguientes avisos.

AVISOS

- La definición de los no terminales y los terminales dentro de las reglas tienen que coincidir con algún elemento de los no terminales o los terminales. Esto quiere decir que si se ha definido el no terminal ***D*** en una de las reglas, y no está puesto como no terminal ni como terminal, entonces el programa da error.
- Es muy importante seguir la estructura de las reglas: No Terminal, espacio, ->, espacio y los no terminales y terminales que la formen. Si después del no terminal de la parte izquierda no se pone un espacio, o si después del símbolo -> no se pone espacio, el programa no puede funcionar correctamente.
- La primera regla del fichero tiene que ser la del terminal inicial.
- No denominar, bajo ningún concepto, un no terminal o un terminal con una letra en mayúscula del abecedario y seguidamente un número, como tampoco hacerlo con una letra en mayúscula del abecedario, al lado una A (como se refleja también en mayúscula) y seguidamente un número. Estas denominaciones están reservadas para el programa y no usarlas para no crear conflictos. Un ejemplo de los no terminales y terminales prohibidos son:

A2 *K58* *LA173*

Una vez definida toda la estructura, hay que mencionar que se puede poner comentarios en este fichero. Los comentarios deben de ser así:

/ Contenido del comentario */*

Puedes poner un comentario de varias líneas pero siempre tiene que empezar por **/*** y terminar por */. Esto es importante para que el programa detecte cuando empieza y termina el comentario.

Si aplicamos todos estos pasos, un ejemplo de fichero ya terminado quedaría como el que se muestra en la siguiente ilustración.

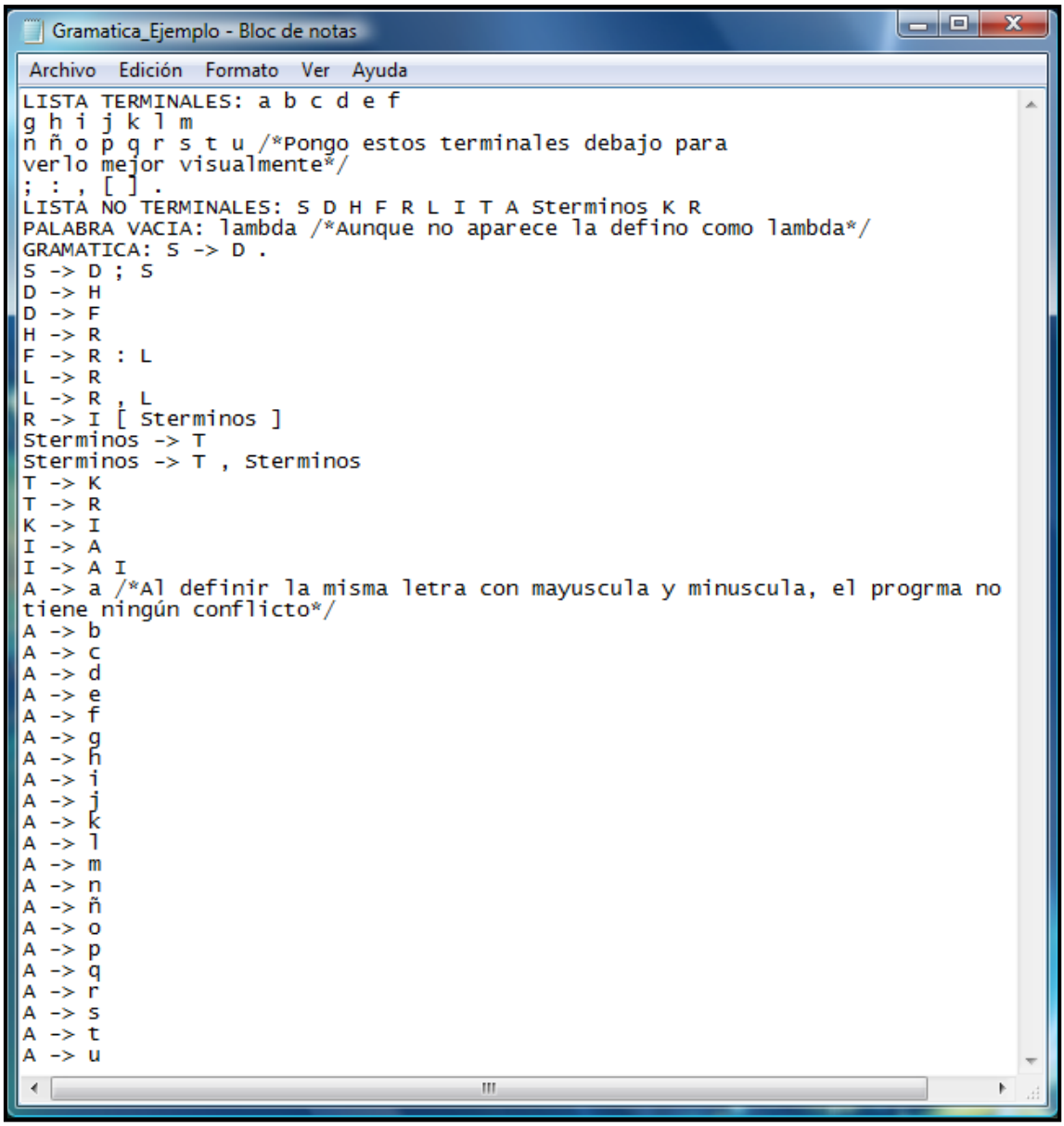


Ilustración 18. Ejemplo fichero de gramática

2.2. Cómo escribir el fichero de las palabras a analizar

En este fichero se pueden introducir varias palabras a analizar. Cada una de ellas se separará con un salto de línea. En este caso, también el programa permite insertar líneas vacías y comentarios. Los comentarios deben de ser así:

/ Contenido del comentario */*

Puedes poner un comentario de varias líneas pero siempre tiene que empezar por **/*** y terminar por */. Esto es importante para que el programa detecte cuando empieza y termina el comentario.

Cada uno de los *Terminales* que componen las palabras a analizar debe ir separado por un espacio. Es importante este punto ya que así el programa puede distinguir los distintos *Terminales* que la componen.

Si aplicamos todos estos pasos, un ejemplo de fichero ya terminado quedaría como el que se muestra en la siguiente ilustración.

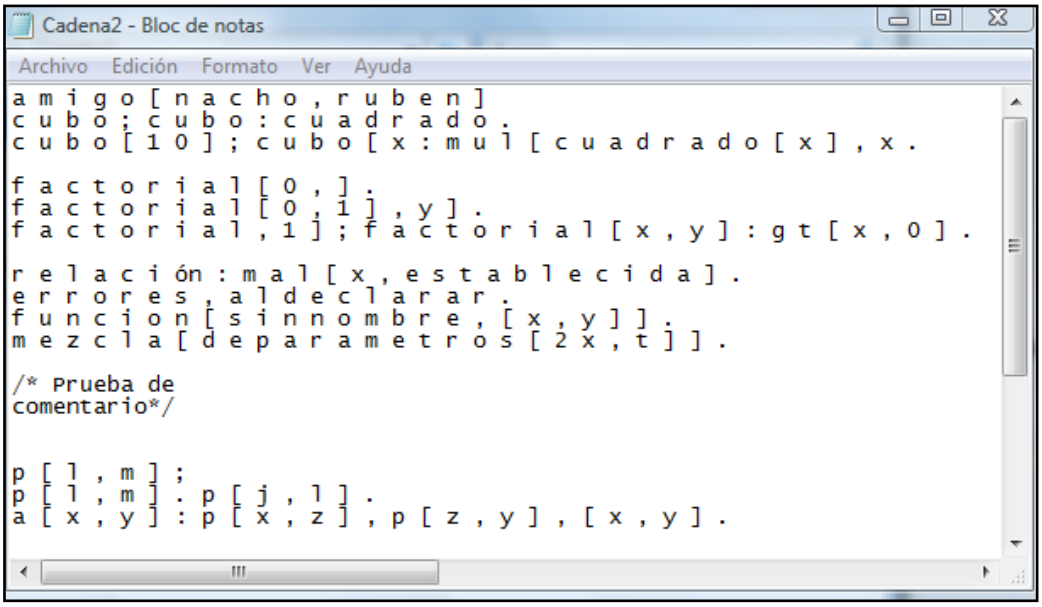


Ilustración 19. Ejemplo fichero de cadenas

3. Acciones del programa gestor de gramáticas

En este punto se explica qué es lo que hace el programa en cada una de las opciones que muestra.

3.1. Comprobar palabra en gramática Forma Normal de Chomsky (algoritmo CYK)

En este apartado se comprueba que una o más palabras pertenecen a la gramática usando el algoritmo CYK como comprobante.

Para utilizar este analizador se necesita tener el fichero de la gramática y las cadenas. El sistema transformará la gramática a Forma Normal de Chomsky y a continuación analiza con el algoritmo CYK. Según la cadena y la gramática obtenida te dice si se pueden generar o no. En caso de que sí se pueda, obtiene el árbol correspondiente (que se puede comprobar con el fichero que genera para los árboles explicado en el punto *4.3. Arbol CYK* del manual.)

3.2. Obtener la Forma Normal de Chomsky de una gramática

En este punto solo se genera la gramática con Forma Normal de Chomsky. Para ello solo se necesita el fichero que contenga la gramática. El resultado se guarda en un fichero con

una determinada estructura, el cual se explica en el punto 4.*1. Forma Normal de Chomsky* de este manual.

3.3. Obtener la Forma Normal de Greibach de una gramática

En este punto solo se genera la gramática con Forma Normal de Greibach. Para ello solo se necesita el fichero que contenga la gramática. El resultado se guarda en un fichero con una determinada estructura, el cual se explica en el punto *4.4. Forma Normal de Greibach* de este manual.

3.4. Comprobar palabra en gramatica Forma Normal de Greibach

En este apartado se comprueba si una o más palabras pertenecen a una gramática en Forma Normal de Greibach. En este punto se necesita tener el fichero de la gramática y las cadenas.

Para realizar el análisis, primero se transforma la gramática a la Forma Normal de Greibach, por lo que se generará el fichero correspondiente a este paso que se explica en el punto *4.4. Forma Normal de Greibach*, y a continuación se analizan las palabras. Los resultados se guardan en un fichero que se genera (que se explica en el punto *4.5. Resultados palabras Forma Normal de Greibach* de este manual). Si se puede generar se crea el árbol correspondiente (que se explica en el punto *4.6. Arbol Forma Normal de Greibach* del manual*).*

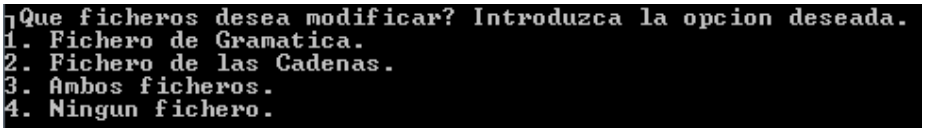
3.5. Comprobar palabra mediante algoritmo de fuerza bruta

En este punto se determina si una o más palabras pertenecen a la gramática usando el algoritmo de Fuerza Bruta como determinante. En este punto, también se leen ambos ficheros, el de la gramática y el de las cadenas.

Los resultados se guardan en un fichero que se genera (que se explica en el punto *4.7. Resultados palabras Fuerza Bruta* de este manual). Si se puede generar se crea el árbol correspondiente (que se explica en el punto *4.8. Arbol Fuerza Bruta* de este manual*).*

3.6. Cambiar la ruta de los ficheros de entrada

Si se escoge esta opción, se podrá cambiar la ruta de los ficheros en los que se encuentra la gramática y las palabras a analizar. Se puede cambiar un solo fichero o ambos. El menú de opciones se muestra en la siguiente imagen.



El usuario solo debe introducir por teclado la opción y pulsar *Enter*.

3.7. Salir

Si se pulsa esta opción, el programa se cerrará por completo. Para poder volver a usarlo tienes que empezar desde el principio, es decir, ejecutando por consola el comando para el *.jar*.

4. Documentos nuevos que genera el programa

En este apartado se explicará el contenido de los ficheros que genera el programa.

4.1. Forma Normal de Chomsky

En este documento se mostrará primero cómo es la gramática que se ha introducido, después cómo queda la gramática al ejecutar los pasos para bien formar la gramática y posteriormente la gramática en Forma Normal de Chomsky. En los pasos intermedios, se especificarán que reglas han sido eliminadas de la gramática. Estos son los que se muestra los que se enumeran a continuación.

1. La gramática tal cual se introdujo en el fichero. Se denomina este paso con el título: ***Gramatica pasada por parámetro***.
2. Eliminación de las reglas innecesarias. Se denomina este paso con el título: ***GRAMATICA reglas Innecesarias***.
3. Eliminación de las reglas no generativas. Se designa este paso con el título***:*** ***GRAMATICA reglas no generativas***.
4. Eliminación las reglas cuyo *No terminales* a la izquierda sean inaccesible. Se denomina este paso con el título: ***GRAMATICA simbolos Inaccesibles***.
5. Eliminación de las reglas del tipo *No terminal* para dar un *No terminal*. Se llama este paso con el título: ***GRAMATICA reglas De Redenominacion***.
6. El resultado de la gramática en Forma Normal de Chomsky. Se nombra este paso con el título: ***GRAMATICA de Forma Normal de Chomsky***.

Los pasos ***GRAMATICA reglas Innecesarias, GRAMATICA reglas no generativas, GRAMATICA imbolos Inaccesibles*** y ***GRAMATICA reglas De Redenominacion*** se repiten tantas veces como sea necesario, y se especifican todas las repeticiones en el fichero. Esta repetición se muestra con el mensaje ******Seguimos comprobando todos los pasos para que quede la gramática limpia*******.

4.2. Tabla del CYK

En este fichero se muestran cada una de las tablas que genera el algoritmo CYK para comprobar si una palabra pertenece o no a esa gramática. Dichas tablas se representan mediante saltos de línea, tabulaciones y el símbolo |. Un ejemplo de tabla es el siguiente:

```
C1 F1 J1 S1 |       |       |       |   E SS A   |
M1 |       |   L   |   B1 E1 R1   |       |
A1 D1 G1 Q1 |   H1   |       |       |       |
N F SS E |       |       |       |       |
A1 D1 G1 Q1 |       |       |       |       |
```

Esta tabla representa una tabla con siguiente formato:

C1 F1 J1 S1				**E SS A**
M1		**L**	**B1 E1 R1**	
A1 D1 G1 Q1	**H1**			
N F SS E				
A1 D1 G1 Q1				

Cada vez que aparezca un símbolo | será que se pasa a otra columna. Y cada uno de los saltos de línea será cada una de las filas de la tabla.

Para separar las distintas palabras analizadas, se pone como título ***Tabla del CYK:*** y debajo la palabra analizada y debajo su correspondiente tabla.

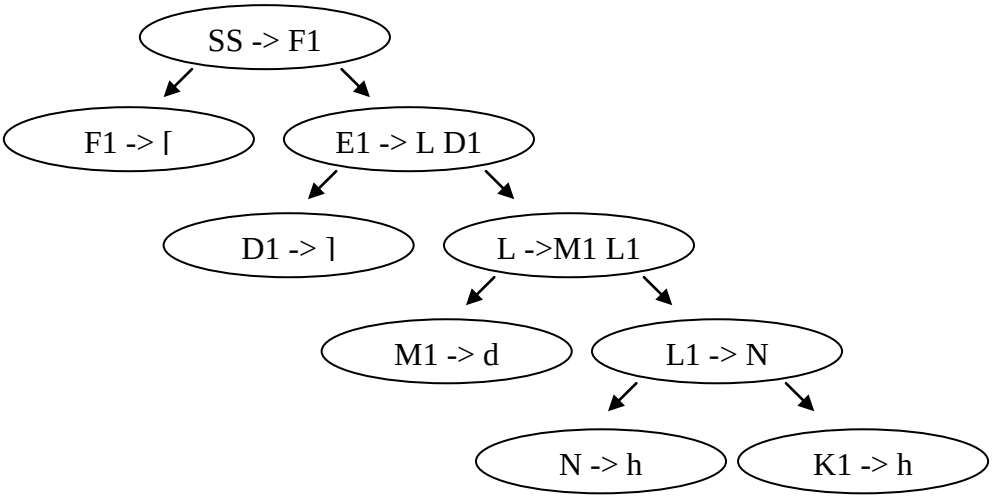
4.3. Arbol CYK

En este fichero se muestran los árboles formados, en caso de que se puedan dar, al mirar cada una de las palabras en la gramática con Forma Normal de Chomsky. Los árboles se muestran de la siguiente manera:

```
Arbol del CYK:
--------------------
De la cadena [ d h h ]

SS -> F1 E1
|--- F1 -> [
|--- E1 -> L D1
|        |--- L -> M1 L1
|        |        |--- M1 -> d
|        |        |--- L1 -> N K1
|        |        |        |--- N -> h
|        |        |        |--- K1 -> h
|        |--- D1 -> ]
```

Este formato corresponde con el que se sigue en la siguiente figura:



El símbolo | sirve para que se vea visualmente a qué nivel del árbol se corresponde la regla que se escribe después de ---.

En el caso de que la palabra a analizar no se pudiese dar, en este fichero se mostraría de la siguiente manera.

```
Arbol del CYK:
--------------------
De la cadena [ d h h

No hay árbol posible
```

Para separar los árboles de las distintas palabras analizadas, se pone como título ***Arbol del CYK:*** y debajo la palabra analizada con su correspondiente árbol a continuación.

4.4. Forma Normal de Greibach

En este documento se mostrará primero cómo es la gramática que se le ha introducido, después cómo queda la gramática al ejecutar los pasos para bien formar la gramática y posteriormente la gramática en Forma Normal de Greibach. En los pasos intermedios, se especificarán qué reglas han sido eliminadas de la gramática. Estos son los que se muestra los que se enumeran a continuación.

1. La gramática tal cual se introdujo en el fichero. Se denomina este paso con el título: ***Gramatica pasada por parámetro***.
2. Eliminación de las reglas innecesarias. Se denomina este paso con el título: ***GRAMATICA reglas Innecesarias***.
3. Eliminación de las reglas no generativas. Se designa este paso con el título***:*** ***GRAMATICA reglas no generativas***.
4. Eliminación las reglas cuyo *No terminales* a la izquierda sean inaccesible. Se denomina este paso con el título: ***GRAMATICA simbolos Inaccesibles***.

5. Eliminación de las reglas del tipo *No terminal* para dar un *No terminal*. Se llama este paso con el título: ***GRAMATICA reglas De Redenominacion***.
6. Para realizar la gramática en Forma Normal de Greibach hay que especificar el formato inicial adecuado en el cual se analizarán las reglas. Se denomina este paso con el título: ***GRAMATICA Ordenacion De Reglas***.
7. Eliminación de las reglas que tengan recursividad a izquierdas. Se denomina este paso con el título***: GRAMATICA Eliminacion Recursividad Izquierdas***.
8. El resultado de la gramática en Forma Normal de Greibach. Se nombra este paso con el título: ***GRAMATICA de Forma Normal de Greibach.***

Los pasos ***GRAMATICA reglas Innecesarias, GRAMATICA reglas no generativas, GRAMATICA imbolos Inaccesibles*** y ***GRAMATICA reglas De Redenominacion*** se repiten tantas veces como sea necesario, y se especifican todas las repeticiones en el fichero. Esta repetición se muestra con el mensaje ******Seguimos comprobando todos los pasos para que quede la gramática limpia*******.

4.5. Resultados palabras Forma Normal de Greibach

En este fichero se muestra cada una de las decisiones de pertenencia a la gramática por cada palabra que se ha introducido en el fichero correspondiente. El formato de las decisiones que se muestra en el fichero es el siguiente:

```
Resultado Forma Normal de Greibach:
-----------------------------------------------
La cadena [ d f v h ]
SI puede ser generada

Resultado Forma Normal de Greibach:
-----------------------------------------------
La cadena b [ d f v h ]
NO puede ser generada
```

Para separar las distintas palabras analizadas, se pone como título ***Resultado Forma Normal de Greibach:*** y debajo la palabra analizada con su correspondiente decisión del análisis debajo de la palabra.

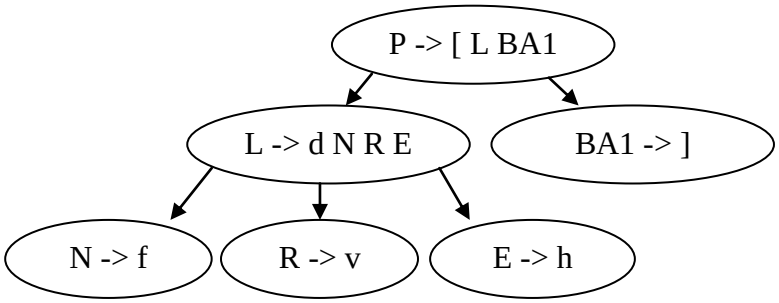
4.6. Arbol Forma Normal de Greibach

En este fichero se muestran los árboles formados, en caso de que se puedan dar, al mirar cada una de las palabras en la gramática con Forma Normal de Greibach. Los árboles se muestran de la siguiente manera:

```
Arbol Forma Normal de Greibach:
----------------------------------------
De la cadena [ d f v h ]

P-> [ L BA1
|--- L -> d N R E
|          |--- N -> f
|          |--- R -> v
|          |--- E -> h
|--- BA1 -> ]
```

Para separar los árboles de las distintas palabras analizadas, se pone como título ***Arbol Forma Normal de Greibach:*** y debajo la palabra analizada con su correspondiente árbol a continuación. Este formato corresponde con el que se sigue en la siguiente figura:



El símbolo | sirve para que se vea visualmente a qué nivel del árbol se corresponde la regla que se escribe después de ---.

En el caso de que la palabra a analizar no se pudiese dar, en este fichero se mostraría de la siguiente manera.

```
Arbol Forma Normal de Greibach:
----------------------------------------
De la cadena [ d f v h

No hay árbol posible
```

4.7. Resultados palabras Fuerza Bruta

En este fichero se muestra cada una de las decisiones de pertenencia a la gramática por cada palabra que se ha introducido en el fichero correspondiente. El formato de las resoluciones que muestra el fichero es el siguiente:

```
Resultado Fuerza Bruta:
-----------------------
La cadena b [ 0 9 ] .
SI puede ser generada

Resultado Fuerza Bruta:
-----------------------
La cadena b [ 0 9 ]
NO puede ser generada
```

Para separar las distintas palabras analizadas, se pone como título ***Resultado Fuerza Bruta:*** y debajo la palabra analizada con su correspondiente decisión del análisis debajo de la palabra.

4.8. Arbol Fuerza Bruta

En *Arbol Fuerza Bruta* se muestran los árboles formados, en caso de que se puedan dar, al mirar cada una de las palabras en la gramática que se pasa inicialmente por el fichero. Los árboles siguen la misma estructura que en los apartados anteriores, en el caso de que se puedan dar, y se muestran de la siguiente manera:

```
Arbol Fuerza Bruta:
-------------------------
De la cadena b [ 0 9 ] .

P -> D .
|--- D -> H
|          |--- H -> R
|          |          |--- R -> I [ S ]
|          |          |          |--- I -> A
|          |          |          |          |--- A -> b
|          |          |          |--- S -> T
|          |          |          |          |--- T -> N
|          |          |          |          |          |--- N -> O N
|          |          |          |          |          |          |--- O -> 0
|          |          |          |          |          |          |--- N -> O
|          |          |          |          |          |          |          |--- O -> 9
```

En caso contrario, se especifica que no hay árbol posible con el formato que se muestra en la siguiente imagen.

```
Arbol Fuerza Bruta:
-------------------------
De la cadena b [ 0 9 ]

No hay árbol posible
```

Para separar los árboles de las distintas palabras analizadas, se pone como título ***Arbol Fuerza Bruta:*** y debajo la palabra analizada con su correspondiente árbol a continuación (en caso de ser posible).

16.Anexo I

En este apartado, incluimos la gramática y las palabras analizadas en la prueba contemplada en el apartado *9.4. Gramática de miniProlog*. Las cadenas a analizar son:

```
padre[luis,maria].

padre[luis,maria];padre[jose,luis].

abuelo[x,y]:padre[x,z],padre[z,y].

padre[luis,maria];padre[jose,luis];abuelo[x,y]:padre[x,z],padre[z,y].

factorial[0,1].

factorial[0,1];factorial[x,y].

factorial[0,1];factorial[x,y]:gt[x,0].

factorial[0,1];factorial[x,y]:gt[x,0],is[z,min[x,1]].

factorial[0,1];factorial[x,y]:gt[x,0],is[z,sub[x,1]],factorial[z,w].

factorial[0,1];factorial[x,y]:gt[x,0],is[z,sub[x,1]],factorial[z,w],is[y,mul[x,w]].
```

A continuación, se muestra todas las reglas pertenecientes a la gramática.

Programa → Declaracion .

Programa → Declaracion ; Programa

Declaracion → Hecho

Declaracion → Regla

Hecho → Relacion

Regla → Relacion : ListaRelaciones

ListaRelaciones → Relacion

ListaRelaciones → Relacion , ListaRelaciones

Relacion → Identificador [ListaTerminos]

ListaTerminos → Termino

ListaTerminos → Termino , ListaTerminos

Termino → Variable

Termino → Constante

Termino → Numero

Termino → Relacion

Constante → Identificador

Identificador → Letra

Identificador → Letra Identificador

Letra → a | b | … | z

Variable → w|x|y|z

Numero → Digito

Numero → Digito Numero

Digito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

17. Anexo II

En este apartado se mostrará cómo queda la gramática en FNC. El resultado es el que se muestra en las siguientes columnas (la unión de las 3 forma la gramática completa):

Expresion → Expresion B1

Expresion → Expresion D1

Expresion → Expresion F1

Expresion → Expresion H1

Numero → Digito Numero

Digito → 0

Digito → 1

Digito → 2

Digito → 3

Digito → 4

Digito → 5

Digito → 6

Digito → 7

Digito → 8

Digito → 9

S → Expresion J1

S → Expresion L1

S → Expresion N1

S → Expresion P1

Expresion → Digito Numero

Numero → 0

Numero → 1

Numero → 2

Numero → 3

Numero → 4

Numero → 5

Numero → 6

Numero → 7

Numero → 8

Numero → 9

S → Digito Numero

S → 0

S → 1

S → 2

S → 3

S → 4

S → 5

S → 6

S → 7

S → 8

S → 9

Expresion → 0

Expresion → 1

Expresion → 2

Expresion → 3

Expresion → 4

Expresion → 5

Expresion → 6

Expresion → 7

Expresion → 8

Expresion → 9

A1 → +

B1 → A1 Expresion

C1 → -

D1 → C1 Expresion

E1 → *

F1 → E1 Expresion

G1 → /

H1 → G1 Expresion

I1 → +

J1 → I1 Expresion

K1 → -

L1 → K1 Expresion

M1 → *

N1 → M1 Expresion

O1 → /

P1 → O1 Expresion