UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

DESARROLLO DE UN MOTOR DE MATEMÁTICA SIMBÓLICA PARA LA HERRAMIENTA CHALKPY

Laura Salcedo Valderrama
Tutor: Luis Fernando Lago Fernández

Julio 2016

DESARROLLO DE UN MOTOR DE MATEMÁTICA SIMBÓLICA PARA LA HERRAMIENTA CHALKPY

AUTOR: Laura Salcedo Valderrama
TUTOR: Luis Fernando Lago Fernández

Departamento de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2016

Resumen

Resumen – La historia de la computación ha estado estrechamente relacionada con la resolución de problemas matemáticos. Durante siglos, grandes figuras en este campo han buscado la construcción de sistemas que deduzcan teoremas y resultados matemáticos. Es en este marco donde surgen los Sistemas de Álgebra Computacional, herramientas que permiten el cálculo simbólico mediante manipulaciones que reflejan los cálculos manuales que realizaría una persona.

Este Trabajo de Fin de Grado busca la creación de motores algebraicos computacionales, a través de un compilador, que proporcionen un sistema de reescritura de términos para la resolución de expresiones matemáticas por medio de manipulaciones. Como caso particular de estos sistemas se pretende conseguir un motor que sustituya el sistema actual de Chalkpy, una herramienta para mejorar la presentación de contenido matemático en el aula. De este modo, se pretende que la herramienta se adapte mejor a las especificaciones de un usuario.

La funcionalidad aportada soluciona los problemas que genera la necesidad de modificar un sistema, ante la introducción de nuevas funcionalidades, mediante la generación de código por un fichero de definiciones. Este recoge los elementos y las reglas de equivalencia que se desean utilizar en el sistema.

El desarrollo de este proyecto ha seguido un ciclo de vida en cascada con retroalimentación para la creación del compilador, y un ciclo de vida iterativo incremental para la parte de la generación del código de los motores. Para la creación del analizador léxico se ha utilizado la herramienta JFlex y para el analizador sintáctico/semántico el generador CUP.

El éxito de este proyecto se refleja en la creación de diversos motores entre los que se encuentra el motor algebraico para la nueva versión de Chalkpy.

Palabras clave – Sistema de Álgebra Computacional, gramática, compilador, Proyecto de Innovación Docente

Abstract

Abstract **–** Computer history has been always linked with mathematical problem solving. For centuries, well-known personalities have been looking forward to build a system capable of self-proving theorems and math results. Is on this scenario where Computer Algebra systems arise: tools that allow symbolic calculus by doing manipulations that reflect the human manual way.

This Bachelor Thesis is about using a compiler to create this kind of tools, to provide a rewriting term system for mathematical expression solving by manipulating them with equivalency rules. As a particular case, it is pretended to achieve a system that replaces the actual one used in Chalkpy, a tool used to improve the presentation of mathematical content in educational enviroment. Thanks to that, the created tool would be better adapted to the user requirements.

The developed functionality solves the problems caused by the need of modifying a system when adding new features by code generation, according to a definition file. This file contains the desired rules that will be used in the system once it is generated.

A waterfall with feedback lifecycle was the method choosen to create the compilator, and an iterative and incremental one was used for generating the engines code. To build the lexical analyzer was chosen, whilst JFlex was chosen; CUP was the tool used to build the sintactic/semantic analyzer.

This project can be considered successful as it allows generation of multiple algebraic engine, the Chalkpy algebraic system being a special case.

Keywords **–** Computer Algebra System, grammar, compiler, Innovation Teaching Program

Agradecimientos

En primer lugar, agradecer a Luis la oportunidad que me dio de poder realizar este proyecto en cuanto le hablé de la posibilidad de desarrollar un Trabajo de Fin de Grado que aplicase los conocimientos adquiridos en Autómatas y Lenguajes. Agradecerle también la paciencia que ha tenido, ya fuese cuando las cosas parecía que no salían o cuando, aunque salían, empezaba a agobiarme y no verle fin.

Del mismo modo, quiero agradecer a todos los profesores con los que me he cruzado en estos cinco años y que me han enseñado tanto en el ámbito académico como en el ámbito personal.

Gracias a Rodrigo por su infinita paciencia cuando le tocaba escucharme por los agobios de las prácticas, los exámenes, la representación estudiantil, el TFG… Ha sido un gran apoyo durante este largo proyecto. Seguramente sin sus ánimos no habría llegado hasta aquí.

Agradecer a Alejandro las horas y risas que hemos dedicado a hacer prácticas en los laboratorios. Pero sobretodo darle las gracias por las horas nocturnas programando como si no hubiese un mañana.

Y terminando los agradecimientos en el ámbito universitario, gracias a todos los compañeros y amigos de la facultad que he hecho entre estas paredes. No me olvido tampoco de mis compañeros de la DEISI y la representación estudiantil. Ellos también me han marcado y ayudado a dejar un poco de lado mi carga de trabajo y disfrutar la vida universitaria de otra manera.

En el ámbito personal, gracias a mi familia y, en especial, a mis padres. Confiaron en que sería capaz de llegar al punto donde hoy me encuentro, aunque hubo algunos años previos a la universidad en que parecía difícil. Destacar también a mi hermana, sin ella la física hubiera hecho que tirase la toalla en alguna que otra ocasión.

En resumen, gracias a todos aquellos que me han acompañado y ayudado a lo largo de estos años en los buenos momentos, pero sobretodo en los malos.

Y a ti pelirroja, simplemente gracias.

INDICE DE CONTENIDOS

1 Introducción 1
1.1 Motivación 1
1.2 Objetivos 2
1.3 Organización de la memoria 2

2 Estado del arte 5
2.1 Sistemas de Algebra Computacional 5
2.1.1 Precedentes 5
2.2 Herramientas CAS 6

3 Definición del proyecto 9
3.1 Alcance 9
3.2 Metodología 9
3.3 Tecnología y herramientas 10

4 Análisis 13
4.1 Requisitos funcionales 13
4.1.1 Generador de motores de matemática simbólica 13
4.1.2 Motor de matemática simbólica 14
4.2 Requisitos no funcionales 15
4.2.1 Generador del motor de matemática simbólica 15

5 Diseño e Implementación 17
5.1 Arquitectura del proyecto 17
5.2 Fichero de definiciones 17
5.2.1 Definición de elementos 18
5.2.2 Definición de reglas de equivalencia 20
5.3 Compilador 21
5.3.1 Analizador léxico 21
5.3.2 Tabla de símbolos 22
5.3.3 Analizador sintáctico/semántico y Gramática 22
5.3.4 Generador de código 25
5.3.5 Sistema de Álgebra Computacional 27

6 Pruebas 29
6.1 Pruebas de inspección de código 29
6.2 Pruebas de caja blanca 29
6.3 Pruebas de caja negra 30

7 Resultados 31
7.1 Un CAS para Chalkpy 31
7.1.1 Definición de elementos para Chalkpy 31
7.1.2 Definición de reglas de equivalencia para Chalkpy 33
7.2 Ejemplo 36
7.3 Integración con la Interfaz 37

8 Conclusiones y trabajo futuro 39

Referencias 41

Glosario 43

Anexos 45
A Expresiones regulares 45

INDICE DE FIGURAS

FIGURA 5-1. ARQUITECTURA DEL PROYECTO. 18

FIGURA 5-2. REPRESENTACIÓN DE LA EXPRESIÓN +[#[2.0],#[3.0]] COMO LISTAS..... 28

FIGURA 7-1. GUI DISEÑADA PARA EL TFG "INTERFAZ ANDROID PARA LA HERRAMIENTA CHALKPY". 37

INDICE DE TABLAS

TABLA 5.2-1. ETIQUETAS PARA LA DEFINICIÓN DE ELEMENTOS EN EL FICHERO DE DEFINICIONES. 19

TABLA 5.3-1. ELEMENTOS ADMITIDOS POR EL ANALIZADOR LÉXICO..... 22

TABLA 5.3-2. GRAMÁTICA DISEÑADA PARA EL COMPILADOR GENERADOR DE CAS. 23

TABLA 5.3-3ACCIONES ASOCIADAS A LAS REGLAS DE LA GRAMÁTICA. 25

TABLA 7.1-1. REGLAS DE EQUIVALENCIA PARA CHALKPY..... 36

TABLA A-1. SIGNIFICADO DE LAS EXPRESIONES REGULARES UTILIZADAS 45

INDICE DE CODIGOS

CÓDIGO 5.2-1. EJEMPLO DE ELEMENTOS PARA UN FICHERO DE DEFINICIONES. 20

CÓDIGO 5.2-2. EJEMPLO DE REGLAS DE EQUIVALENCIA PARA UN FICHERO DE DEFINICIONES..... 21

CÓDIGO 5.3-1. CÓDIGO GENERADO PARA UNA REGLA DE EQUIVALENCIA *OPERATE*. 26

CÓDIGO 7.1-1. DEFINICIÓN DE ELEMENTOS PARA CHALKPY. 32

1 Introducción

Este Trabajo de Fin de Grado, como parte de un Proyecto de Innovación Docente realizado en la Escuela Politécnica Superior de la Universidad Autónoma de Madrid [1], tiene como propósito el desarrollo de un motor de matemática simbólica para la herramienta Chalkpy. Se pretende reemplazar la librería SymPy [2], utilizada en la primera versión de dicha herramienta, haciendo posible la manipulación de expresiones algebraicas sencillas sin necesidad de grandes modificaciones a raíz de la introducción de nuevas operaciones.

En esta sección se detallará la motivación del presente TFG seguido de los objetivos principales y de una descripción estructural del documento.

1.1 Motivación

Chalkpy es una herramienta informática de apoyo a la docencia que permite la manipulación de expresiones algebraicas sencillas. Se realizó como parte de un Proyecto de Innovación Docente [3] y Trabajo de Fin de Grado [4] en la Universidad Autónoma de Madrid, utilizando Python y la librería de matemática simbólica SymPy. Su objetivo es promover la resolución dinámica de ecuaciones de primer grado con una incógnita, mitigando los problemas que puede presentar el uso de una pizarra tradicional como apoyo para la explicación de contenidos matemáticos.

SymPy, usada como base para la representación interna de las ecuaciones, es una librería de matemática simbólica enfocada a que el propio sistema resuelva las expresiones matemáticas de forma automática. Esto provoca que se introduzcan automáticamente simplificaciones que no se desean. La solución por la que se optó en el momento de su desarrollo fue usar la representación de las ecuaciones en forma de String, aunque no era demasiado conveniente. Por estos motivos se ha decidido sustituir el engranaje de Chalkpy por un Sistema de Álgebra Computacional [5] (CAS, del inglés Computer Algebra System) que se desarrollará como motor de la nueva versión de Chalkpy.

Por otro lado, aunque tras el desarrollo de Chalkpy se concluyó que la herramienta cumplía con los objetivos buscados, tras la realización de pruebas por parte de los usuarios, estos solicitaron la ampliación de funcionalidad, incluyendo nuevas operaciones [4]. Aquí surgen nuevos inconvenientes. La lógica e interfaz son muy específicas y difíciles de extender ya que se desarrollaron, únicamente, para la resolución de ecuaciones lineales. La solución buscada en este caso ha sido la generación automática del CAS por medio de un compilador donde se pueden definir las operaciones necesarias. Durante el diseño de este compilador se ha observado que es capaz de generar diversos motores siendo, uno de ellos, el caso particular que se buscaba, el motor de Chalkpy.

La motivación de este Trabajo de Fin de Grado es dar solución a los problemas que presenta la versión actual de Chalkpy y poder generar otros motores de álgebra computacional. Esta solución final será, por tanto, una aplicación particular del generador de motores, consiguiendo un motor de matemática simbólica que permita adaptarse mejor a las especificaciones dadas por el usuario.

1.2 Objetivos

Este Trabajo de Fin de Grado se centra en el desarrollo de un motor de matemática simbólica para la herramienta Chalkpy, como caso particular de la creación de un compilador que genere dicho tipo de motores. Para ello se marcan dos objetivos de gran importancia. El primero de estos objetivos es la posibilidad de definir expresiones matemáticas generales. El segundo, definir diversas equivalencias entre expresiones que permitan transformar de una a otra a petición del usuario. Estas equivalencias se llamarán también reducciones.

A continuación se exponen las condiciones que se han marcado para cumplir con el proyecto:

- Concretar un lenguaje, tanto en símbolo como en forma, para la definición de expresiones y equivalencias matemáticas.

- Se podrá definir cualquier operador, mediante el lenguaje determinado, modificando diversos parámetros que tengan que ver con su simbología, su funcionalidad, sus argumentos y el cumplimiento de unas propiedades matemáticas básicas.

- Se podrá definir diversas equivalencias matemáticas, mediante el lenguaje definido, siguiendo una estructura establecida que contenga la expresión original y la resultante tras la aplicación de las reglas definidas.

- Desarrollo de un compilador que, a través de un fichero de definición de operadores y equivalencias, genere un fichero Java para un motor algebraico que permita la manipulación de las ecuaciones mediante dichas equivalencias.

- Creación de un programa de testeo que permita crear y manipular expresiones, a través de comandos por teclado, de acuerdo a las definiciones generadas.

- Interfaz que permita la conexión con la Interfaz Gráfica de Usuario. Esta condición forma parte del Trabajo de Fin de Grado “Interfaz Android para la herramienta Chalkpy” [6]. No obstante, se requiere la actuación conjunta de la interfaz y el motor.

1.3 Organización de la memoria

El presente documento se organiza en los siguientes capítulos:

- **Estado del arte:** ilustra el estado del arte haciendo una introducción sobre el álgebra computacional y señalando las herramientas que tienen un fin similar al presente proyecto.

- **Definición del proyecto:** define el proyecto mediante su alcance, la metodología utilizada en su desarrollo y las herramientas y tecnologías que ha sido necesario utilizar.

- **Análisis:** presenta los requisitos funcionales y no funcionales que definen la herramienta.

- **Diseño e Implementación:** describe el diseño y la implementación tanto de la aplicación para la creación del motor, como de la gramática utilizada en el compilador y el resultado de la ejecución de ambos.

- **Pruebas:** ilustra las pruebas que se han ido realizando a lo largo del desarrollo, así como aquellas ejecutadas tras su finalización para comprobar su correcto funcionamiento y el alcance de los objetivos.

- **Resultados:** indica los resultados del presente trabajando obteniendo el motor necesario para la creación de la nueva versión de Chalkpy.

- **Conclusiones y trabajo futuro:** se exponen las conclusiones del trabajo, así como las posibles mejoras futuras a desarrollar.

Al final de este documento se encuentran las referencias utilizadas, un glosario de términos y los apéndices, que contienen datos menos relevantes, pero de interés.

2 Estado del arte

En esta sección se explicará el fundamento matemático teórico sobre el que se apoya la aplicación, el álgebra computacional, y su evolución a lo largo del tiempo. También se examinarán diferentes herramientas basadas en estos sistemas.

2.1 Sistemas de Algebra Computacional

Un Sistema de Álgebra Computacional, conocido como CAS, es un programa informático, o calculadora avanzada, que modeliza la idea de algoritmo algebraico y facilita el cálculo simbólico. Es decir, son herramientas que permiten la manipulación de expresiones mediante ciertas reglas. Estas reflejan las operaciones que realizaría una persona cuando intenta resolver una expresión algebraica. La aplicación de dichas reglas se realiza por medio de los sistemas de reescritura de términos [7], que permiten transformar expresiones complejas en otras más sencillas de manipular.

El álgebra computacional surge a mediados del siglo XX, a principios de la década de los 60. No obstante, es conveniente hacer un repaso a la evolución del álgebra ligada a la informática, para entender los fundamentos en los que se basa. No se debe olvidar que los CAS están muy unidos con los sistemas de reescritura de términos ya mencionados. Existen diversas implementaciones de estos sistemas en lenguajes de programación como puede ser Prolog [8], Haskell [9] o LISP [10]. No obstante, es en el cálculo lambda, que tiene gran influencia sobre los sistemas anteriores, donde más se puede apreciar la extensión de los sistemas de reescritura de términos. El cálculo lambda [11], diseñado por Alonzo Church en 1932, es un sistema formal completo que modeliza la matemática y permite definir qué es una función computable.

2.1.1 Precedentes

Las raíces del álgebra se remontan a las matemáticas babilónicas alrededor del año 2000 a.C., el griego Diofanto en el siglo III d.C. y los árabes en la Edad Media [12]. Fueron estos últimos quienes desarrollaron un estudio más profundo sobre las ecuaciones algebraicas. Se enfocaron en resolver problemas mediante métodos que funcionaran para cualquier tipo de ecuación, es decir, buscaron la creación de algoritmos que se pudieran aplicar de una forma independiente.

Se considera al matemático árabe Al-Juarismi como el fundador del álgebra gracias a su obra *"Kitab al-jabr wa al-muqabala"* [12] donde usa la palabra *al-ğabr* que dio lugar al término actual. En él se ofrecía un método para resolver ecuaciones lineales y cuadráticas sin el símbolo algebraico, los números negativos o el cero. No será hasta finales del siglo XVI, con Viète, cuando se introducen letras para los coeficientes de ecuación y se desarrolla un lenguaje simbólico para expresar, de forma más simple, las expresiones polinómicas.

En la Edad Moderna Leibniz ansía crear un lenguaje formalizado para una máquina de cálculo que deduzca teoremas y resultados, llegando a crear una máquina aplicada en algunas calculadoras mecánicas. No obstante, es Charles Babbage quien conseguiría un gran avance al diseñar la máquina analítica, que fuese programable para cualquier tipo de cálculo, junto con Ada Lovelace quien escribió diversos programas para ella.

En el siglo XX, Hilbert formula el *"Problema de la decisión"* (*Entscheidungsproblem*), planteando si las matemáticas son completas, consistentes y decidibles. Es decir, si existe un algoritmo que, dada una fórmula matemática, y usando ciertos axiomas, decidiera si es, o no, un teorema. La respuesta al problema, que no tiene solución, vino de la mano de Gödel y Turing. El primero demostró que un conjunto de axiomas no podía ser completo y consistente a la vez [13]. Por su parte, Alan Turing [14], junto con su problema de la parada y la máquina de Turing, la cual realizaba manipulaciones con símbolos, concluyó que no se puede saber con certeza si una demostración obtendría solución. Eso lo consiguió demostrando que, en ocasiones, la máquina que diseñó no paraba, por lo que no se podía saber el resultado.

Esto indica que se puede considerar la máquina de Turing como la precursora del CAS: ambos tienen como base una actuación automática en función de un símbolo leído. Sin embargo, el primer ejemplo real de un CAS aparece en 1963, evolucionando entre la física teórica y la inteligencia artificial. Desarrollado por Martinus Veltman. Fue denominado *"Schoonschip"* y realizaba diversas manipulaciones de matemática simbólica. Un año después, Carl Engelman creó *MATHLAB,* usando LISP, que permite la inferencia lógica. Finalmente, en 1987 aparece la primera calculadora de mano con un CAS, denominada HP-28 y desarrollado por Hewlett-Packard. Dicho CAS podía resolver nuevas manipulaciones algebraicas de forma simbólica como series de Taylor o ecuaciones algebraicas. Desde entonces, y hasta ahora, los CAS han ido mejorando e introduciendo mayores funcionalidades. En el siguiente apartado se explican algunos de los más importantes.

2.2 Herramientas CAS

Desde la aparición del primer CAS hasta la actualidad, su funcionalidad y las herramientas que han ido usando como base estos sistemas ha ido creciendo. Su aplicación se puede encontrar en diferentes ámbitos, como se verá en las siguientes herramientas, despuntando el ámbito de las matemáticas, la física y la ingeniería. No obstante, y como se expuso en el Trabajo de Fin de Grado donde se desarrolló Chalkpy, el desarrollo de la tecnología ha permitido que tenga gran repercusión en las aulas [4]. De este modo, la transmisión de los conocimientos y su comprensión por parte de los alumnos se vuelve más dinámica y eficiente. Partiendo de esta base, se busca que el uso de estas herramientas no sustituya los cálculos manuales o el conocimiento de las reglas algebraicas necesaria para la resolución de los problemas. Por este motivo, tanto el presente trabajo como las herramientas mostradas, dan la posibilidad de efectuar las reducciones manualmente.

A continuación se describen algunos de los Sistemas de Álgebra Computacional que cumplen parte de los requisitos de este proyecto:

REDUCE

REDUCE [15] es un Sistema de Álgebra Computacional que empezó a ser desarrollado en la década de 1960 por Anthony Hearn, contando, posteriormente, con diversas colaboraciones para llevarlo a cabo. Está escrito en su propio dialecto de LISP, llamado *Standard LISP* [16].

Las similitudes con el CAS resultante del presente proyecto se basan en la simplificación de expresiones (tanto de forma automática como manual), la posibilidad de definir nuevas funciones, y la posibilidad de elegir diversos formatos para la expresión resultante. Por otro lado, hay que destacar que la manera en que se permite introducir las nuevas funciones requiere del conocimiento, o aprendizaje previo, de parte del lenguaje en el que está desarrollado el CAS.

Mathematica

Mathematica [17] es una herramienta de la compañía *Wolfram Research*, una empresa desarrolladora de algunos de los CAS más utilizados en la actualidad, como WolframAlpha [18]. Considerado un Sistema de Álgebra Computacional, es también un poderoso lenguaje de programación de proposito general.

Una de las tecnologías destacables que utiliza es el *Wolfram Language*, un lenguaje simbólico que potencia el sistema y cubre parte de los objetivos del nuevo Chalkpy. El propósito de este lenguaje es la construcción de algoritmos que sean accesibles de forma automática a través de un lenguaje simbólico unificado, que permite la representación de diversos elementos como expresiones simbólicas. Del mismo modo, facilita su adaptación a otros programas de una manera simple. La funcionalidad completa de Mathematica está contenida en una herramienta que requiere el pago de licencia.

YACAS

YACAS (Yet Another Computer Algebra System) [19] es, como su propio nombre indica, *Otro Sistema Más de Álgebra Computacional*. Diseñado en su propio lenguaje de programación (el cual está muy relacionado con LISP), permite la ampliación de su funcionalidad por parte del usuario. Además, al igual que los anteriores, su funcionalidad no se limita a la resolución automática de ecuaciones, también permite la manipulación manual para las aplicaciones de reducciones.

Al contrario de lo que se busca en este trabajo, y al igual que pasaba con la *Mathematica*, ampliar su funcionalidad no es sencillo. Para ello se requiere aprender los diferentes tipos de elementos con los que trabaja YACAS, las diferentes funciones y reglas que utiliza, la estructura de sus programas o cómo evalúa las expresiones, entre otros conocimientos necesarios.

3 Definición del proyecto

En esta sección se detalla el alcance de la nueva versión de Chalkpy, la metodología empleada en el desarrollo del proyecto y las tecnologías y herramientas usadas para llevarlo a cabo.

3.1 Alcance

Este motor de matemática simbólica permite la creación automatizada de una nueva versión de Chalkpy. Este Chalkpy 2.0 puede resolver ecuaciones donde los límites se encuentran en las expresiones matemáticas y las reglas de equivalencia definidas por el usuario. En el desarrollo de este Trabajo de Fin de Grado, se pueden traducir los límites marcados por las expresiones en la definición de número, variable, constante cero, constante uno, constante menos uno, suma, resta, multiplicación, división, menos unario y la igualdad. Por otro lado, los límites que marcan la definición de elemento y reglas de equivalencia son más amplios y se encuentran en la imposibilidad de fijar conceptos complejos que, por ejemplo, contengan condiciones.

El generador de motores está enfocado para usuarios que tengan un mínimo conocimiento en programación, siendo necesario únicamente saber realizar diferentes operaciones matemáticas en Java. Dicha necesidad está motivada por el planteamiento de que el generador de motores sea una herramienta de diseño para un programa informático. Por ello, algunas definiciones necesitarán la escritura de una breve línea de código en dicho lenguaje.

3.2 Metodología

El problema de cómo desarrollar la herramienta estuvo presente desde el inicio de la definición del proyecto. Al querer crear un sistema algebraico propio, y de generación automática, era necesario diseñar, no solo el compilador que lo generase, sino también una estructura que permitiese la aplicación de las reglas de equivalencia independientemente de qué hicieran estas.

Para la elección de la metodología también se tuvo en cuenta que, aunque los requisitos del compilador estaban bien definidos, su capacidad dependía de la definición de lenguaje y la gramática. Es decir, podrían ser imposibles para el motor ciertas características factibles teóricamente, o bien, fueran posibles otras que, a priori, resultasen dudosas. Por último, se estudió el hecho de cómo se iba a ir creando la herramienta. Es decir, primero sería necesario desarrollar el fichero de definiciones y el analizador léxico y sintáctico para, posteriormente, llevar a cabo la generación de código y las diferentes reducciones.

Debido a todos los motivos expuestos se optó por un desarrollo en cascada con retroalimentación, seguido de un desarrollo iterativo e incremental [20]. El primero coincidía

con la definición del fichero de definiciones, la creación de la gramática y el lenguaje y las acciones del compilador, exceptuando la generación del código. Para esto último se llevó a cabo un análisis de requisitos funcionales y no funcionales, un diseño de las distintas partes que componían esta etapa seguido de su implementación y, por último, las pruebas para comprobar si se había alcanzado la funcionalidad deseada. El hecho de elegir cascada con retroalimentación se fundamentó en la posible necesidad de replantear el diseño de las reglas de la gramática, principalmente, en caso de no ser suficientes para la funcionalidad esperada.

Durante el desarrollo iterativo se realizó un análisis inicial de requisitos que complementara al anterior y se definieron las etapas a realizar en cada incremento: elección de la funcionalidad (reducción) a implementar, diseño de dicha funcionalidad, implementación y pruebas.

3.3 Tecnología y herramientas

En esta sección se definen las tecnologías y herramientas que se han empleado en la creación del nuevo Chalkpy, tanto en el ámbito de desarrollo como en el de documentación.

Tecnologías

El generador del motor de matemática simbólica que, en un futuro, podría integrarse en una aplicación móvil junto a dicho motor y su interfaz, se ha desarrollado en el lenguaje de programación *Java* [21]. Esto está motivado por el hecho de que Android [22] da soporte nativo a dicho lenguaje. Por el mismo motivo, el lenguaje del propio CAS, resultante de la ejecución del generado, que ya se integra en la interfaz Android para la herramienta Chalkpy, ha sido también Java.

Herramientas

A continuación, se expondrán las herramientas utilizadas para llevar a cabo el presente proyecto. No obstante, cabe destacar las dos que han sido más importantes:

- *JFlex* [23] es un generador de analizadores léxicos, o escáneres, basado en un autómata finito determinista [24]. Su funcionamiento se fundamenta en la generación del analizador a través de un conjunto de expresiones regulares asociadas a acciones. Este analizador, a su vez, lee una entrada que se parsea y, si corresponde con alguna de las expresiones especificadas, ejecuta la acción correspondiente.

- *CUP* [25] es un generador de analizadores sintácticos LALR [26] para gramáticas libres de contexto. Es necesario especificar los símbolos de la gramática y las reglas de producción. A cada una de ellas se le puede asignar una acción determinada para ejecutarse si alguna entrada corresponde con dicha regla.

Entorno

Eclipse [27] es el entorno de desarrollo (IDE) más extendido para la programación en Java. Su funcionamiento se basa, en gran medida, en plugins (complementos de una aplicación para agregar cierta funcionalidad) que facilita la introducción de herramientas durante el desarrollo. Además, cuenta con un plugin para la integración de JFlex y CUP. Por ello, y por contar con la herramienta para pruebas JUnit, se ha elegido este IDE para el desarrollo de todo el proyecto. Por último, señalar la ventaja de que integre Git para el control de versiones, explicado en el siguiente punto.

Control de versiones

El control de versiones es la gestión de los cambios que se realizan a lo largo del desarrollo de un proyecto. En este caso se ha utilizado una plataforma web y una aplicación de escritorio:

- *Bitbucket* [28] es un servicio de alojamiento basado en web con un sistema de control de versiones basado en Git. A diferencia de otros sistemas permite tener repositorios privados en las cuentas de planes gratuitos.

- *SourceTree* [29] es un cliente de Git para Windows y MacOS. Su aplicación de escritorio provee de una interfaz gráfica para gestionar el control de versiones de una manera fácil e intuitiva.

Gráficos

Todas las imágenes de este documento se han generado mediante *Cacco* [30], una herramienta web on-line que permite la creación de diagramas.

4 Análisis

En este capítulo se exponen los requisitos funcionales y no funcionales que son el resultado del análisis sobre las necesidades que debe cubrir el sistema. Estos requisitos han sufrido algunas modificaciones a lo largo del desarrollo, por lo que aquí se recogen los requisitos finales del sistema.

4.1 Requisitos funcionales

Se entiende por requisito funcional aquellas acciones fundamentales que deben tener lugar durante la ejecución del sistema. Para mayor claridad y orden se diferencian los requisitos del generador de motores de matemática simbólica y los pertenecientes al CAS generado.

4.1.1 Generador de motores de matemática simbólica

RF 1. El usuario podrá definir cualquier operación matemática, de forma simbólica, para su utilización en el CAS.

RF 2. El usuario podrá definir cualquier constante matemática, de forma simbólica, para su utilización en el CAS como una operación.

RF 3. El usuario podrá definir el concepto de número y variable, de forma simbólica, para su utilización en el CAS como una operación.

RF 4. Todos los elementos deberán recoger, como mínimo, las siguientes características:

- Nombre.
- Símbolo.
- Tipo de elemento (operación, número o string).
- Tipo de argumento (operación, número o string).
- Número de argumentos.
- Aparición como argumentos.
- Representación.
- Cumplimiento de la propiedad asociativa.
- Cumplimiento de la propiedad conmutativa.

Una explicación de cada una de ellas se puede ver en la Tabla 5.2-1

RF 5. El nombre de todos los elementos debe ser único.

RF 6. El símbolo de todos los elementos debe ser único.

RF 7. El usuario deberá definir obligatoriamente el nombre y símbolo de todos los elementos.

RF 8. La introducción del valor de las características de un elemento no debe obligar al usuario a seguir un orden determinado.

RF 9. El sistema deberá dar un valor por defecto a todas las características que no haya definido el usuario.

RF 10. El usuario podrá definir reglas de equivalencia utilizando los elementos que se hayan especificado.

RF 11. Las definiciones de elementos y reglas de equivalencia deben ser a través de un fichero de definiciones.

RF 12. El fichero de definiciones siempre deberá ser solicitado al usuario

RF 13. Las equivalencias para conmutar, asociar y desasociar expresiones con un número arbitrario de elementos se considerarán primitivas. Deberán ser generadas por el sistema, cuando se cumplan las propiedades correspondientes, sin necesidad de que las defina el usuario.

RF 14. Se permitirá la definición de reglas de equivalencia a criterio del usuario sin comprobar su corrección matemática.

RF 15. El sistema deberá generar el código necesario para la ejecución de las reducciones, acorde a las reglas de equivalencia, que será la pieza fundamental para la manipulación de expresiones.

RF 16. La localización del fichero del fichero de código generado debe ser daba por el usuario de manera obligatoria.

4.1.2 Motor de matemática simbólica

RF 17. El sistema deberá contar con una interfaz para probar su funcionamiento por un usuario externo.

RF 18. El usuario podrá realizar tres acciones para probar el sistema:

- Insertar ecuación.
- Aplicar reducción.
- Finalizar ejecución.

RF 19. El sistema permitirá la introducción de expresiones matemáticas formadas por los elementos definidos en el motor.

RF 20. Solo se permitirá trabajar con una expresión matemática a la vez.

RF 21. La expresión matemática con la que se trabaja podrá ser cambiada en cualquier momento.

RF 22. Dada una expresión matemática se podrá pedir que se aplique cualquier reducción definida.

RF 23. El sistema deberá detectar si la expresión matemática cumple las características para una reducción y, en caso de ser así, aplicarla.

RF 24. En caso de pedir la aplicación de una reducción y no se cumplan los requisitos definidos por la regla de equivalencia correspondiente, la expresión no deberá ser modificada.

4.2 Requisitos no funcionales

Se entiende por requisitos no funcionales aquellos que especifican criterios o restricciones del software que definen requerimientos de ejecución mínima.

4.2.1 Generador del motor de matemática simbólica

RNF 1. Si durante la ejecución del motor ocurre algún error deberá notificarse al usuario mediante la impresión de un mensaje de error.

RNF 2. Será necesario contar con la versión de Java 8 para poder ejecutar el generador.

5 Diseño e Implementación

En este capítulo se explican los aspectos más importantes de la fase de diseño e implementación del proyecto. Se detallan las decisiones de diseño tomadas para cumplir los requisitos fijados en el capítulo anterior. Sobre estos diseños se podrá señalar ciertos aspectos de codificación, o de decisiones de desarrollo, tomadas. Previamente se dará una visión general de la organización del proyecto en función de su arquitectura.

5.1 Arquitectura del proyecto

En esta sección se va a detallar la arquitectura del proyecto con el apoyo de la Figura 5-1. Se diferencian tres partes: el fichero de definiciones, el compilador y el Sistema de Álgebra Computacional.

El fichero de definiciones es un fichero de texto, sin formato, destinado a la definición de los elementos requeridos en los requisitos RF 1 a RF 3, y las reglas de equivalencia requeridas por el requisito RF 10.

El compilador, con la información del fichero de definiciones, generará el motor del CAS que permitirá trabajar con los elementos y equivalencias definidas por el usuario. Para ello se apoyará en un analizador léxico que parseará el fichero y le pasará la información al analizador sintáctico/semántico. Este, a su vez, se apoyará de una tabla de símbolos y un generador de código para crear la clase del motor.

Por último, el Sistema de Álgebra Computacional permitirá la manipulación de expresiones, formadas por los elementos definidos, a través de las reducciones que se elijan. Como se puede observar en la Figura 5-1 el CAS no es solo el motor, debe contar también con un elemento que modeliza la expresión o ecuación con la que se trabaja.

5.2 Fichero de definiciones

El fichero de definiciones, como ya se ha mencionado, es un fichero de texto, sin formato, destinado a la definición de los elementos y las reglas de equivalencia. Su estructura básica son dos bloques: el primero contiene la definición de los elementos, mientras que el segundo contiene las reglas de equivalencia. En las siguientes subsecciones se explica el diseño de ambos.

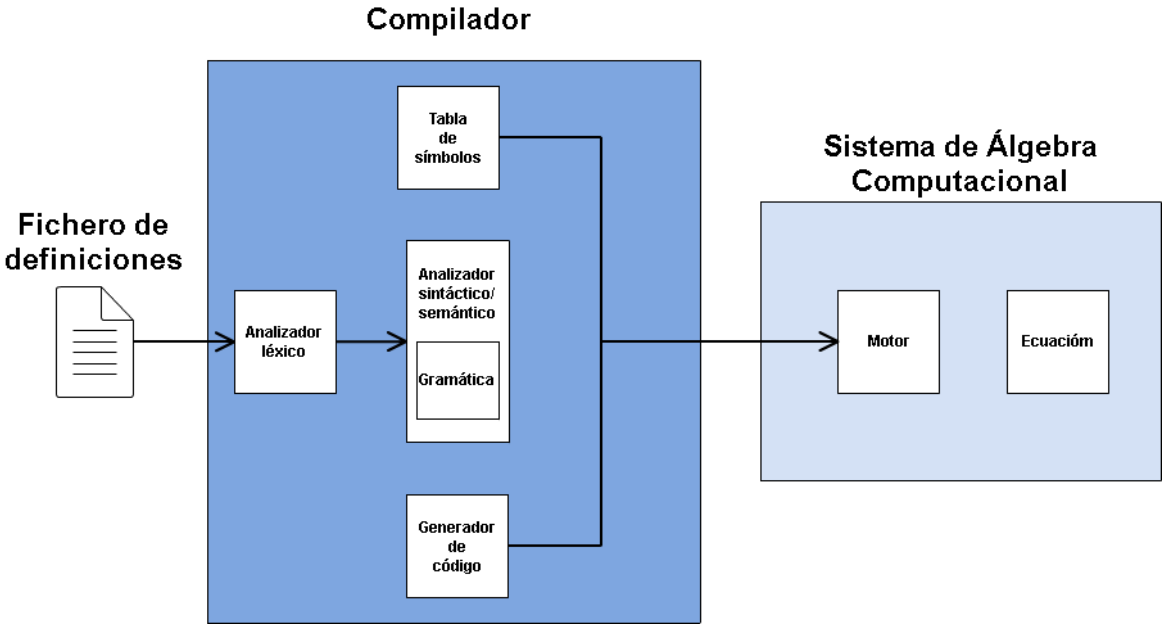

Figura 5-1. Arquitectura del proyecto.

5.2.1 Definición de elementos

Todos los elementos deben ser declarados en líneas diferentes y seguirán el estándar `NOMBRE(lista de opciones)`, donde el nombre es una cadena de letras en mayúscula. La lista de opciones puede ser variable, ya que solo es obligatorio definir el símbolo y el nombre del elemento como señala el requisito RF 7, y no cuenta con ningún orden estipulado.

Las opciones de un elemento se componen de una etiqueta, dos puntos y el valor asignado, separadas unas de otras por comas. Si una etiqueta aparece más de una vez, o su valor no corresponde a uno de los permitidos, cuando se intente utilizar el fichero en el compilador este informará del error. Las etiquetas junto a los valores que admiten y su valor por defecto, que buscan cumplir los requisitos RF 4 y RF 9, se definen en la Tabla 5.2-1.

Etiqueta	Función	Posibles valores	Valor por defecto
tag_symbol	Símbolo de la operación (entre comillas)	$ / % / & / * / + / - / # / = / ^ / @ seguido, opcionalmente, por letras mayúsculas.	-
tag_type	Tipo del elemento	oper para operaciones number para número string para variables	oper

Etiqueta	Función	Posibles valores	Valor por defecto
tag_type_args	Lista ordenada de tipo de argumentos (entre paréntesis)	oper para un elemento number para un número string para una variable oper… para N elementos	lista vacía (0 argumentos)
tag_rep	Tipo de representación	prefix para prefija infix para infija none sin representación	prefix
tag_des	Descripción del elemento (entre comillas)	Cadena que permite cualquier símbolo	vacía
flag_in_cf	Indica si el elemento se puede usar en las reglas de equivalencia	true false	true
flag_in_args	Indica si el elemento puede aparecer como argumento	true false	true
flag_associtive	Indica si el elemento cumple la propiedad asociativa	true false	false
flag_commutative	Indica si el elemento cumple la propiedad conmutativa	true false	false

Tabla 5.2-1. Etiquetas para la definición de elementos en el fichero de definiciones.

Es interesante puntualizar la manera en que se registra el número de argumentos y su tipo en un elemento. Como se observa en la Tabla 5.2-1 la etiqueta para estos datos recibe como valor una lista de elementos que indica tanto el tamaño como el tipo. La manera de hacerlo es mediante el orden en la que se colocan los valores. Si, por ejemplo, se asignase como valor a `tag_type_args` la lista (`oper`, `number`, `string`, `oper`, `number`, `oper`…), esto significaría que el elemento al que correspondiese dicha etiqueta recibiría seis argumentos como mínimo y un máximo indeterminado. Además, y de forma obligatoria, el segundo y quinto elemento serían de tipo numérico, el tercero de tipo string, y el primero, cuarto, sexto y en delante, de tipo operacional. Señalar que `oper`… solo puede aparecer una vez por elemento.

Un pequeño ejemplo de este bloque se puede encontrar en el Código 5.2-1 donde se definen los siguientes tres elementos:

- `SUM`: cumple la propiedad conmutativa y asociativa. Puede recibir como argumentos un número indeterminado de `oper` y se añade una descripción que aclara su función como operación suma. Por último, se indica que puede aparecer en el bloque de equivalencias o como parámetro de otras operaciones, siendo su

símbolo +. Como no se define el tipo de elemento, ni la representación, se le asignará el valor por defecto, es decir, será de tipo operacional.

- `NUMBER`: representa el concepto de número cuyo símbolo se muestra como #. Al igual que el anterior puede aparecer en las reglas de equivalencia y como argumento de otros elementos. Por otro lado, solo admite un parámetro de tipo `number` y no se representa. El resto de etiquetas, al no estar definidas, toman el valor por defecto.

- `ZERO`: define el 0. Se simboliza mediante *&ZERO* aunque este no se representa. Es un elemento de tipo operacional y el resto de etiquetas, al no estar definidas, toman el valor por defecto.

```
SUM(flag_conmutative:true,flag_associative:true,
tag_type_args:(oper...), flag_in_cf:true,
flag_in_args:true, tag_symbol:"+")

NUMBER(tag_symbol:"#", flag_in_cf:true,
flag_in_args:true,tag_type_args:(number))

ZERO(tag_symbol:"&ZERO", tag_type:oper)
```

Código 5.2-1. Ejemplo de elementos para un fichero de definiciones.

5.2.2 Definición de reglas de equivalencia

Las reglas de equivalencia se definen como `X >>> Y`, donde X indica la expresión inicial e Y indica la expresión resultante. Para formar dichas expresiones se utilizan los elementos definidos en el bloque anterior, nombrados por símbolo, y colocando sus argumentos entre corchetes (*SIMBOLO*`[`*argumentos*`]`). Se permite expresar la aparición de cualquier elemento arbitrario, ya sea número, operación o string, señalándolo con `_z`, donde `z` es cualquier letra minúscula. También se admite el recurso operar, por medio de la instrucción `operate`, que representa un argumento numérico.

Además, para la creación de las reglas de equivalencia se han detallado las siguientes condiciones:

- Si a ambos lados de la equivalencia se encuentra _z, ambos representarán el mismo elemento.

- En el lado derecho de la equivalencia solo podrán aparecer elementos arbitrarios que aparezcan en el lado izquierdo.

- Los elementos arbitrarios que no aparezcan como argumentos de ninguna operación, por lo que no reciben tipo, se tomarán como elementos de tipo operacional.

- Si un elemento arbitrario aparece más de una vez en una regla, el tipo que le pertenece, en función de su colocación en la regla, debe ser siempre el mismo.

- Si un elemento arbitrario aparece en el lado derecho de la equivalencia sin pertenecer a ninguna operación, su tipo deberá ser, obligatoriamente, operacional.

En el Código 5.2-2 se ilustra un ejemplo de cómo se crearían dos reglas de equivalencia para los elementos definidos en el ejemplo del Código 5.2-1. La primera regla indica que la suma de dos números cualesquiera es equivalente a otro cuyo argumento será el resultado de sumar los argumentos de los dos primeros números. La segunda regla indica que la suma de cualquier expresión y cero es el valor del elemento arbitrario inicial.

```
+[#[_a],#[_b]] >>> #[operate("_a+_b")]

+[_a,&ZERO[]] >>> _a
```

Código 5.2-2. Ejemplo de reglas de equivalencia para un fichero de definiciones.

5.3 Compilador

En esta sección se detallan las diferentes partes de las que se compone el compilador. Se dará una visión de cómo se ha diseñado el analizador léxico, la tabla de símbolos, la gramática, y el gestor para la generación de código.

5.3.1 Analizador léxico

El analizador léxico se ha diseñado de manera que reconozca los elementos que se recogen en la Tabla 5.3-1. Estos elementos son remitidos al analizador sintáctico/semántico para su análisis.

Elementos	
Espacio en blanco	//*Comentario*
(	)
[	]
:	,
>>>	(“*cadena*”)
tag_symbol	tag_type
tag_rep	tag_type_args
tag_des	flag_in_cf
flag_commutative	flag_associative

Elementos	
flag_in_args	string
oper	number
oper…	none
infix	prefix
true	false
"*cadena de caracteres*"	"*símbolo*"
Una o más letras mayúsculas	*_letra minúcula*

Tabla 5.3-1. Elementos admitidos por el analizador léxico

5.3.2 Tabla de símbolos

Una tabla de símbolos es una estructura de datos que usa un compilador para almacenar información necesaria durante la generación del código. En este caso, se han diseñado dos tablas, una global y otra local para cada reducción.

Tabla de símbolos global

La tabla de símbolos global guarda toda la información de los elementos que se van definiendo: nombre, símbolo, tipo, representación, descripción, etc. Cada elemento debe poder buscarse en la tabla por nombre o símbolo, dando como resultado el valor que toman todas las etiquetas para ese elemento.

Tabla de símbolos local para las reducciones

La tabla de símbolos local solo guarda la información relevante a la regla de equivalencia que está siendo tratada por el compilador. La información que recoge es el nombre de los elementos arbitrarios (entiéndase por nombre la letra precedida por el símbolo "barra baja") y el tipo que le asigna el compilador a dicho elemento. En la subsección siguiente se especificará cómo se consigue este dato.

5.3.3 Analizador sintáctico/semántico y Gramática

Se ha diseñado una gramática libre de contexto para el compilador que genera el motor de los CAS. Una gramática libre de contexto es un conjunto de reglas de formación que definen las cadenas de caracteres admisibles en un lenguaje. Estas reglas deben seguir la forma $V \rightarrow w$ donde V es un símbolo no terminal y w es una cadena de terminales y/o no terminales. La gramática resultante y las acciones que realizan, junto a una etiqueta para referenciarlas en este documento, se detallan en la Tabla 5.3-2. Una relación completa de las expresiones regulares utilizadas se puede encontrar en el Anexo A.

Etiqueta	Regla
R1	<file> ::= <definitions> <lst_reductions>
R2	<definitions> ::= <resource> <definitions>
R3	<definitions> ::= <resource>
R4	<resource> ::= <name> **(**<lst_options>)
R5	<name> ::= **[A-Z]+**
R6	<lst_options> ::= <lst_options > **,** <option>
R7	<lst_options> ::= <option>
R8	<option> ::= **tag_symbol :** <symbol>
R9	<option> ::= **tag_type :** <valuest_type>
R10	<option> ::= **tag_type_args : (<**lst_typesArgs> **)**
R11	<option> ::= **tag_rep : <**values_rep>
R12	<option> ::= tag_desc : "[^\"]+"
R13	<option> ::= **flag_in_cf :** <values_boolean>
R14	<option> ::= **flag_in_args :** <values_boolean>
R15	<option> ::= **flag_associative :** <values_boolean>
R16	<option> ::= **flag_conmmutative :** <values_boolean>
R17	<lst_typesArgs> ::= <lst_typesArgs> **,** **<**values_generic>
R18	<lst_typesArgs> ::= <values_generic>
R19	<symbol> ::= "[$|%|&|*|+|-|#|=|^|@][A-Z]+"
R20	<symbol> ::= [$|%|&|*|+|-|#|=|^|@][A-Z]+
R21	<values_generic> ::= **string**
R22	<values_generic> ::= **oper…**
R23	<values_generic> ::=
R24	<values_generic> ::= <values_type>
R25	<values_type> ::= **oper**
R26	<values_type> ::= **number**
R27	<values_rep> ::= **prefix**
R28	<values_rep> ::= **infix**
R29	<values_rep> ::= **none**
R30	<values_boolean> ::= **true**
R31	<values_boolean> ::= **false**
R32	<lst_reductions> ::= <reduction> <lst_reductions>
R33	<lst_reductions> ::= <reduction>
R34	<reduction> ::= <oper> <equivalence> <oper>
R35	<equivalence> ::= **>>>**
R36	<oper> ::= <symbol> **[** <args> **]**
R37	<oper> ::= **_[a-z]**
R38	<oper> ::= operate ("[^"]+")
R39	<args> ::= <lst_args>
R40	<args> ::=
R41	<lst_args> ::= <lst_args> **,** <oper>
R42	<lst_args> ::= <oper>

Tabla 5.3-2. Gramática diseñada para el compilador generador de CAS.

Junto al diseño de la gramática se tuvieron que detallar las acciones que debían realizar algunas de las reglas. En la Tabla 5.3-2 se explican las acciones que finalmente fueron necesarias, referenciando las reglas a las que corresponden. Las reglas que no aparecen únicamente propagan valores, no siendo necesaria su inclusión en la tabla.

Regla	Acción
R1	Cierra el fichero de código generado escribiendo previamente el final del mismo.
R2	Controla que ha terminado el bloque de definición de elementos.
R3	Abre el fichero para el código generado del CAS y escribe la cabecera y las funciones generales (constructor, impresión de expresiones, etc.).
R4	Almacena el elemento definido.
R5	Controla que el nombre de los elementos no se repita.
R8	Controla que no se repita la etiqueta `tag_symbol`.
R9	Controla que no se repita la etiqueta `tag_type` en el elemento actual y actualiza su valor del tipo del elemento.
R10	Controla que no se repita la etiqueta `tag_type_args` en el elemento actual y actualiza su lista de los tipos de argumentos.
R11	Controla que no se repita la etiqueta `tag_rep` en el elemento actual y actualiza su valor de representación.
R12	Controla que no se repita la etiqueta `tag_desc` en el elemento actual y actualiza su descripción.
R13	Controla que no se repita la etiqueta `flag_in_cf` en el elemento actual y actualiza su bandera de aparición en la definición de equivalencias.
R14	Controla que no se repita la etiqueta `flag_in_args` en el elemento actual y actualiza su bandera de aparición como argumento.
R15	Controla que no se repita la etiqueta `flag_associative` en el elemento actual y actualiza su bandera de propiedad asociativa.
R16	Controla que no se repita la etiqueta `flag_commutative` en el elemento actual y actualiza su bandera de propiedad conmutativa.
R17	Controla que solo pueda aparecer una vez el valor *`oper...`* en los argumentos.
R19	Durante el bloque de definición de elementos comprueba que no se repita el símbolo de los elementos definido.
R20	Durante el bloque de definición de reglas de equivalencia comprueba que todos los símbolos utilizados hayan sido definidos previamente.
R34	Comprueba que el tipo de los elementos a ambos lados de la equivalencia son operaciones. Además llama a la clase apropiada para la generación del código de la reducción correspondiente.
R35	Controla que se va a tratar el lado derecho de una regla de equivalencia.
R36	Comprueba que la operación que se reduce puede aparecer en el bloque de reglas de equivalencia y que el número y tipo de los argumentos coincida con el definido en el elemento. Si un argumento no tuviese tipo asignado se le dará el que marque el elemento según la posición que ocupe.
R37	En el lado izquierdo de una regla de equivalencia debe comprobar si el elemento genérico tiene ya un tipo asignado para darle ese o uno genérico. En el lado derecho de una regla de equivalencia debe comprobar que el elemento genérico haya sido declarado en el lado izquierdo.

Regla	Acción
R38	Controla que se esté en el lado derecho de la equivalencia ya que solo ahí se puede operar. Además, debe comprobar que los elementos genéricos se hayan declarado en el lado izquierdo y que sean números.

Tabla 5.3-3Acciones asociadas a las reglas de la gramática.

5.3.4 Generador de código

Como ya se ha señalado, la regla R34 es la encargada de que el código para el motor del CAS sea generado. Para ello, existirán dos clases que se encargarán de generar dicho código, enfocándose cada una en un tipo de funcionalidad. Esto se debe a que el motor contará con funciones que se limiten a la manipulación de la expresión con la que se trabaje, y otras que se encarguen de trabajar con la estructura de dicha expresión. A continuación se explicará cómo se genera el código de una reducción. Posteriormente, en la siguiente sección, se hablará de la clase que representa el motor algebraico y sus funciones más importantes.

Cuando se quiere generar el código de una reducción es necesario conocer tanto el elemento de la izquierda como el de la derecha de la equivalencia. Lo primero que se hace, es generar distintas comprobaciones sobre el elemento de la izquierda para ver si la ecuación sobre la que se ejecutará el código cumple las condiciones para la reducción. Posteriormente se generará el código que lleve a cabo la creación del elemento derecho en la estructura de representación del CAS, tal y como está definido en la regla de equivalencia. Todo esto se hace de forma recursiva sobre los argumentos de los elementos.

En el Código 5.3-1 se muestra, a modo de ejemplo, el código generado para la primera regla del ejemplo mostrado en el Código 5.2-2. En primer lugar, se comprueba que el elemento general sea de tipo `SUM` y tenga dos argumentos. A continuación, se lleva a cabo las comprobaciones del primer y segundo argumento por orden. En ambos casos, al ser elementos definidos, se controlaría que el elemento es `NUMBER` y que cuenta, únicamente, con un argumento. De este último argumento solo debe comprobarse que sea numérico. Una vez pasadas todas las comprobaciones comienza a generarse la nueva lista. Para ello se crea una nueva expresión y se genera un argumento vacío en el que se introducirá la suma de los números pedidos. Estos ocupan la posición representada por `_a` y `_b` en el código anterior. Finalmente se actualiza, de forma recursiva para todos los elementos creados, un campo llamado identificador del padre explicado en la siguiente sección.

```
public Operation reduction0(Operation oper){

      if (!oper.getOperId().equals("SUM")){
            System.err.println("Reduccion no aplicable. "
                        + "La operación no coindice.");
            return oper;
      }


      if (oper.getNumberOfArgs()!=2){
            System.err.println("Reduccion no aplicable. "
                        + "Numero de argumentos no validos.");
            return oper;
      }
```

```
      if (!oper.getArg(0).getOperId().equals("NUMBER")){
            System.err.println("Reduccion no aplicable. "
                        + "La operación no coindice..");
            return oper;
      }

      if (oper.getArg(0).getNumberOfArgs()!=1){
            System.err.println("Reduccion no aplicable. "
                        + "Numero de argumentos no validos.");
            return oper;
      }

      if (!oper.getArg(0).getArg(0).isNumber()){
            System.err.println("Reduccion no aplicable. "
                        + "El argumento no es un numero.");
            return oper;
      }

      if (!oper.getArg(1).getOperId().equals("NUMBER")){
            System.err.println("Reduccion no aplicable. "
                        + "La operación no coindice..");
            return oper;
      }

      if (oper.getArg(1).getNumberOfArgs()!=1){
            System.err.println("Reduccion no aplicable. "
                        + "Numero de argumentos no validos.");
            return oper;
      }

      if (!oper.getArg(1).getArg(0).isNumber()){
            System.err.println("Reduccion no aplicable. "
                        + "El argumento no es un numero.");
            return oper;
      }

      Operation newOper = new Operation();
            newOper.setOper(new Operation("NUMBER"));
            newOper.getArgs().add(new Operation());
            Float res = oper.getArg(0).getArg(0).getNumericArg()
                  +oper.getArg(1).getArg(0).getNumericArg();

      newOper.getArg(0).setNumericArg(res);

      newOper.setParentIdsRec(oper.getParentID());
      return newOper;
}
```

Código 5.3-1. Código generado para una regla de equivalencia *operate*.

5.3.5 Sistema de Álgebra Computacional

El motor algebraico se compone de dos clases: una que representa el motor en sí y es generada por el compilador, y otra que representa la ecuación y es igual para todos los CAS. Se va a dar una visión general sobre ambas clases indicando los aspectos más importantes.

AlgebraicEngine

Su cometido principal es la manipulación de las expresiones algebraicas. Todo su código es autogenerado por funciones primitivas (como la reducción de la propiedad asociativa) o por las reglas de equivalencia, siendo estas últimas funciones resultado de implementar las manipulaciones definidas en el fichero de definiciones.

La propia clase guarda una ecuación que se corresponde con la expresión con la que se está trabajando. A dicha expresión es posible aplicarle cualquiera de las reducciones generadas mediante una llamada a la función que corresponda. Hay que tener en cuenta que al ser código generado las distintas funciones de reducción se diferencian por un número, este corresponde a la posición en la que se encontraba la equivalencia en el fichero de definiciones.

Operation

La función de esta clase es modelar una expresión matemática en forma de lista de listas. Esta estructura de datos se puede ver como una operación general que tiene un identificador, un atributo con su valor representativo de elemento, y una lista de argumentos. A su vez, cada argumento es una operación con un valor asociado y otra lista como argumento. Incluso los números y variables son listas, aunque sin argumentos. Para poder diferenciar los operadores de los números y variables se definen tres atributos, de valor representativo, que puede tomar la operación: un string para el nombre de la operación, un string para la variable y un `float` para los valores numéricos. Un ejemplo de esta representación de listas se puede ver en la Figura 5-2, donde se representa una suma de dos números, el 2 y el 3.

Cada lista tiene asignado un identificador único y una referencia al identificador del padre. Estos dos atributos se introdujeron para que cualquier interfaz que se utilice sobre el CAS pueda buscar, a través del motor, un elemento determinado de forma inequívoca. También permite que, teniendo al elemento padre de una ecuación, sea posible aplicar una reducción a cualquier subexpresión. Basta con recorrer las diferentes listas y modificar dicha subexpresión, tras aplicar la reducción, por el valor que se obtenga.

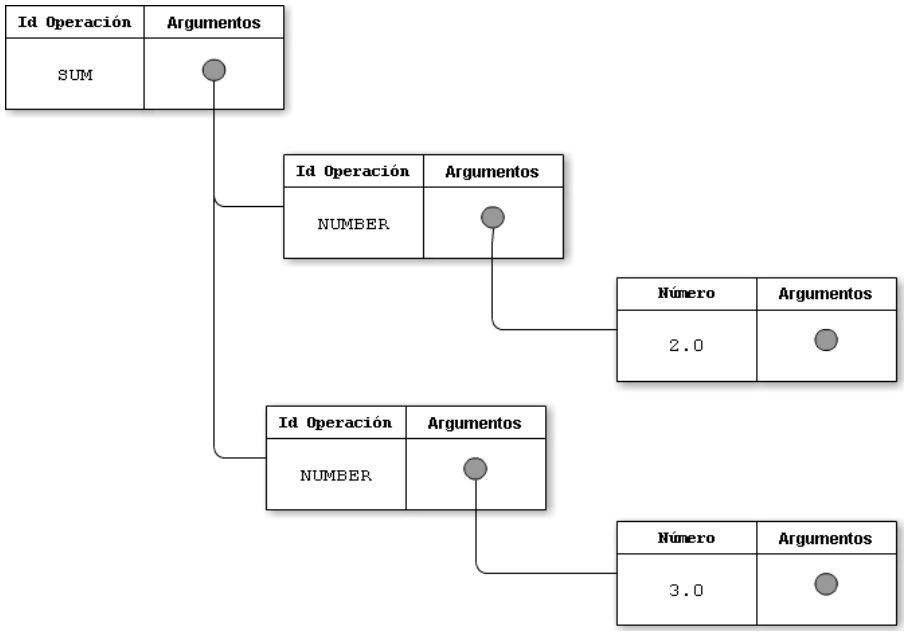

Figura 5-2. Representación de la expresión +[#[2.0],#[3.0]] como listas.

6 Pruebas

Las pruebas que se han realizado durante todo el desarrollo del trabajo se han centrado, principalmente, en detectar los posibles errores del compilador y del CAS generado, que formará parte de Chalkpy, junto a la Interfaz de usuario y la interfaz de comunicación desarrollados en el Trabajo de Fin de Grado "Interfaz Android para la herramienta Chalkpy" [6]. Debido a que ambas interfaces quedan fuera del alcance del proyecto solo se han realizado pruebas internas. Se entiende por pruebas internas aquellas que se centran en el código de la aplicación.

6.1 Pruebas de inspección de código

Estas pruebas consisten en revisar el código desarrollado, o generado, en función del módulo que se esté verificando. Este tipo de revisión se ha llevado a cabo durante el desarrollo en cascada cada vez que se terminaba un módulo, y durante el desarrollo iterativo tras la finalización de cada incremento. En cada etapa de prueba se inspeccionó el código del módulo o funcionalidad que se acaba de desarrollado y aquellos otros que habían sufrido alguna modificación.

Los errores que se han encontrado con estas pruebas se concentraron en su mayoría en la generación del código, donde faltaban ciertas instrucciones. En casi todos los casos esto estaba provocado por una mala asignación en algunos tokens que conllevaba la no generación de algunas comprobaciones en las reducciones. En este punto fue posible resolver la mayoría de los problemas encontrados, no obstante, para solucionar otros, se necesitó la ejecución de pruebas de caja blanca.

6.2 Pruebas de caja blanca

Las pruebas de caja blanca se centran en comprobar el funcionamiento interno, o flujo del programa, en función de su lógica. Su base reside en probar todos los caminos posibles que puede ejecutar el código desarrollado mientras se busca provocar posibles errores o probar los casos extremos.

Para estas pruebas se utilizó el depurador de Eclipse para seguir las instrucciones que se iban ejecutando, junto a sus acciones, en función del fichero de definiciones introducido. También se depuró la ejecución de las reducciones según ciertas expresiones. Como apoyo en las pruebas para el compilador se generaba un fichero de depuración en el que se imprimían las reglas por orden de ejecución y un resumen de qué acción había generado.

Como se ha mencionado en la sección anterior, estas pruebas permitieron la resolución de errores en las comprobaciones que debe realizar una reducción.

6.3 Pruebas de caja negra

Las pruebas de caja negra se centran en la salida que debe dar una cierta funcionalidad por la entrada que haya recibido, sin tener en cuenta las instrucciones ejecutadas para conseguirlo.

Su uso se centró en probar las funcionas de las clases que utilizaba el compilador para generar el código. Además, se realizaron pruebas de caja negra cada vez que se añadía nueva funcionalidad al motor de Chalkpy, es decir, cada vez que se generaba una nueva reducción. Estas últimas se ejecutaron introduciendo casos correctos, o incorrectos, para la reducción que se deseaba probar y comparando su salida con la esperada.

Al centrarse estas pruebas en el código generado casi todos los problemas habían sido resueltos con las pruebas anteriores. No obstante, fueron decisivas para encontrar un error en la generación del código de las reglas de equivalencia, las cuales precisaban la creación de un nuevo operador. Esta detección ocurrió la primera vez que se generó el código para una regla de este estilo y consiguió que el problema no se extendiese a otras que siguiesen la misma lógica.

7 Resultados

Tras el desarrollo de este Trabajo de Fin de Grado se ha obtenido como resultado un compilador cuya función es la generación de código, para motores de matemática simbólica, en base a unos elementos y reglas de equivalencia, determinados por el usuario en un fichero de definiciones.

Un caso particular es el motor de matemática simbólica para Chalkpy que se buscaba como primer objetivo en este trabajo y se expone en el siguiente apartado. Este motor es fácilmente extensible con la introducción de nuevas definiciones en el fichero de definición que se va a detallar.

7.1 *Un CAS para Chalkpy*

Como ya se ha explicado, este proyecto buscaba la creación de un motor de matemática simbólica, que se adaptase mejor a las especificaciones que pudiese requerir un usuario, para la nueva versión de Chalkpy. Debido al compilador desarrollado se ha podido generar dicho motor y, además, permitir la ampliación de funcionalidad, introduciendo nuevas operaciones, de forma automática y sencilla gracias al fichero de definiciones.

A continuación, se detallará el fichero de definiciones utilizado para Chalkpy, desglosándolo en los dos bloques explicados en el capítulo 5, que refleja su funcionalidad y qué tipo de expresiones puede resolver.

7.1.1 Definición de elementos para Chalkpy

En el Código 7.1-1 se puede observar el bloque de definición de elementos del fichero utilizado para la creación del nuevo Chalkpy. En este caso se han definido 10 elementos que serán los que se puedan utilizar en las reglas de equivalencia y en la creación de expresiones con las que trabajar.

La siguiente lista expone, uno por uno, todos los elementos definidos en el mismo orden que aparecen, y referenciados con el mismo nombre. A su vez, las etiquetas de cada uno de ellos también se detallarán en orden.

- *SUM*: representa la operación suma. Este elemento cumple la propiedad conmutativa, puede tener un número variable de argumentos de tipo operación (`oper`) y cumple con la propiedad asociativa. Su descripción aclara, por si fuera necesario, que, como ya se ha señalado, este elemento representa la suma. Por último, se permite su utilización en la definición de las reglas de equivalencia y como argumento de otros elementos. Su símbolo es el +. Como no se define el tipo de elemento ni el valor de representación, se les el valor por defecto.

```
SUM(flag_conmutative:true,
tag_type_args:(oper...), flag_associative:true,
tag_desc:"Es una suma", flag_in_cf:true,
flag_in_args:true, tag_symbol:"+")

MINUS(tag_type_args:(oper), tag_symbol:"-",
tag_desc:"Representa el menos unario")

PROD(tag_symbol:"*", flag_conmutative:true,
flag_associative:true, tag_type_args:(oper...))

EQU(tag_symbol:"=", tag_type:oper,
tag_type_args:(oper,oper), flag_in_args:false)

NUMBER(tag_symbol:"#", flag_in_cf:true,
flag_in_args:true, tag_type_args:(number))

VAR(tag_type:oper, tag_symbol:"$",
tag_type_args:(string))

ZERO(tag_symbol:"&ZERO", tag_type:oper)

ONE(tag_symbol:"&ONE", tag_type:oper)

MONE(tag_symbol:"&MONE", tag_type:oper)

INV(tag_symbol:"@INV", tag_type_args:(oper))
```

Código 7.1-1. Definición de elementos para Chalkpy.

- *MINUS*: representa el menos unario para conseguir la resta por medio de la suma. Este elemento solo acepta un argumento de tipo operación, siendo su representación el símbolo −. La descripción aclara que representa el menos unario y no la resta. El resto de etiquetas, al no estar definidas, toman el valor por defecto.

- *PROD*: representa la multiplicación con el símbolo: *. Cumple tanto la propiedad asociativa como la conmutativa. Al igual que la suma acepta un número arbitrario de elementos de tipo operación. El resto de etiquetas, al no estar definidas, toman el valor por defecto.

- *EQU*: representa la igualdad y se define su símbolo como =. Es un elemento de tipo operación cuyos argumentos son dos operaciones. A diferencia de los elementos definidos, este no puede ser utilizado como argumento de otros. El resto de etiquetas, al no estar definidas, toman el valor por defecto.

- *NUMBER*: representa el concepto de número. Se define su símbolo por # permitiendo que aparezca en el bloque de definición de reglas de equivalencia y como argumento de otros elementos. Solo tiene un argumento de tipo numérico. El resto de etiquetas, al no estar definidas, toman el valor por defecto.

- *VAR*: representa el concepto de variable numérica. Es de tipo operación y se define simbólicamente por $. Al igual que el elemento anterior solo tiene un argumento, pero es de tipo string. El resto de etiquetas, al no estar definidas, toman el valor por defecto.

- *ZERO*: representa el 0 y se simboliza mediante `&ZERO`. Es un elemento de tipo operacional y el resto de etiquetas, al no estar definidas, toman el valor por defecto.

- *ONE*: representa el 1 y se simboliza mediante `&ONE`. Es un elemento de tipo operacional y el resto de etiquetas, al no estar definidas, toman el valor por defecto.

- *MONE*: representa el -1 y se simboliza mediante `&MONE`, siendo un elemento de tipo operacional. El resto de etiquetas, al no estar definidas, toman el valor por defecto.

- *INV:* representa el inverso, para conseguir la división en conjunto con la multiplicación. Solo acepta un argumento de tipo operacional. Es resto de etiquetas tomarán el valor por defecto.

7.1.2 Definición de reglas de equivalencia para Chalkpy

Las reglas de equivalencia de Chalkpy, y sus explicaciones, se detallan en la Tabla 7.1-1. Estas reglas permiten la operación de ecuaciones de primer grado con una incógnita obteniendo, mediante manipulaciones del usuario y si así lo desea, el resultado. De la misma forma, es posible resolver operaciones matemáticas sin incógnitas a través de manipulaciones.

Id	Regla de equivalencia	Significado
RE1	`+[#[_a],#[_b]] >>> #[operate("_a+_b")]`	La suma de dos números es equivalente a otro, cuyo argumento es el resultado de sumar los argumentos del número iniciales.
RE2	`+[#[_a], &ONE[]] >>> #[operate("_a+1")]`	La suma de un número y la constante uno es otro, cuyo argumento se obtiene incrementando en uno el elemento inicial.
RE3	`+[&ONE[],&ONE[]] >>> #[operate("(float) 1+1")]`	La suma de dos constantes uno es un número cuyo argumento sea el resultado de sumar uno más uno.
RE4	`+[_a,&ZERO[]] >>> _a`	La suma de cualquier elemento más la constante cero es el elemento.

Id	Regla de equivalencia	Significado
RE5	`_a >>> +[_a,&ZERO[]]`	Cualquier elemento es equivalente a la suma de sí mismo y cero.
RE6	`+[#[_a],&MONE[]] >>> #[operate("_a-1")]`	La suma de un número y la constante menos uno es equivalente otro número, su argumento es el resultado de restar uno al valor el elemento inicial.
RE7	`*[#[_a],#[_b]] >>> #[operate("_a*_b")]`	La multiplicación de dos números es equivalente a otro número cuyo argumento, sea el resultado de multiplicar los valores de los elementos iniciales.
RE8	`*[_a,&ZERO[]] >>> &ZERO[]`	La multiplicación de un elemento cualquiera y la constante cero es la constante cero.
RE9	`*[_a,&ONE[]] >>> _a`	La multiplicación de un elemento y la constante uno es el elemento.
RE10	`_a >>> *[&ONE[], _a]`	Cualquier elemento es equivalente a la multiplicación de sí mismo y la constante uno.
RE11	`-[-[_a]] >>> _a`	La negación de la negación de un elemento es dicho elemento.
RE12	`-[#[_a]] >>> #[operate("-1*_a")]`	La negación de un número es otro número cuyo argumento es el resultado de multiplicar menos uno por el argumento del número.
RE13	`-[_a] >>> *[&MONE[], _a]`	La negación de un elemento es equivalente a la multiplicación de la constante menos uno y el elemento.
RE14	`-[&ZERO[]] >>> &ZERO[]`	La negación de la constante cero es cero.
RE15	`-[&ONE[]] >>> &MONE[]`	La negación de la constante uno es la constante menos uno.
RE16	`-[&MONE[]] >>> &ONE[]`	La negación de la constante menos uno es la constante uno.
RE17	`+[_a,-[_a]] >>> &ZERO[]`	La suma de un elemento y su negado es la constante cero.
RE18	`+[#[_a],-[#[_b]]] >>> #[operate("_a-_b")]`	La suma de un número con un numero negado es equivalente a otro número cuyo argumento es la resta del primero con el segundo.
RE19	`+[_a,-[&ZERO[]]] >>> _a`	La suma de un elemento y la constante cero negada, es equivalente al elemento.
RE20	`+[&ZERO[],-[_a]] >>> -[_a]`	La suma de la constante cero y un elemento negado es equivalente al elemento negado.

Id	Regla de equivalencia	Significado
RE21	`+[*[_b,_a],*[_c,_a]] >>> *[+[_b,_c],_a]`	Representa el factor común.
RE22	`*[_a,+[_b,_c]] >>> +[*[_a,_b],*[_a,_c]]`	Representa la propiedad distributiva.
RE23	`@INV[@INV[_a]] >>> _a`	El inverso del inverso de un elemento es dicho elemento.
RE24	`@INV[-[_a]] >>> -[@INV[_a]]`	El inverso de un número negado es equivalente a la negación del inverso del número.
RE25	`@INV[#[_a]] >>> #[operate("(float) 1.0/_a")]`	El inverso de un número es equivalente a otro número cuyo argumento es la división de uno entre el argumento del primero.
RE26	`@INV[&ONE[]] >>> &ONE[]`	El inverso de la constante uno es la constante.
RE27	`@INV[&MONE[]] >>> &MONE[]`	El inverso de la constante menos uno es la constante.
RE28	`*[_a,@INV[_a]] >>> &ONE[]`	La multiplicación de un elemento por su inverso es equivalente a la constante uno.
RE29	`*[-[_a],@INV[_a]] >>> &MONE[]`	La multiplicación de un elemento negado y el inverso del elemento es equivalente a la constante menos uno.
RE30	`*[_a,@INV[-[_a]]] >>> &MONE[]`	La multiplicación de un elemento y el inverso del elemento negado es equivalente a la constante menos uno.
RE31	`*[#[_a],@INV[#[_b]]] >>> #[operate("(float) _a/_b")]`	La multiplicación de un número y el inverso de otro número es igual a un tercero cuyo argumento es la división de los valores de los elementos iniciales.
RE32	`=[+[_a,_b],_c] >>> =[_a,+[_c,-[_b]]]`	La igualdad entre la suma de dos elementos y un tercero es equivalente a la igualdad entre el primer elemento, y la suma del tercer elemento con el segundo negado.
RE33	`=[_a,_a] >>> =[+[_a,-[_a]],&ZERO[]]`	La igualdad entre el mismo elemento es equivalente a la igualdad entre la suma, del elemento y su negado, y la constante cero

Id	Regla de equivalencia	Significado
RE34	`=[*[_a,_b],_c] >>> =[_a,*[_c,@INV[_b]]]`	La igualdad entre la suma de dos elementos y un tercero es equivalente a la igualdad del primer elemento y la multiplicación del tercero y e inverso del segundo.
RE35	`+[&MONE[],&MONE[]] >>> -[#[operate("(float) 1+1")]]`	La suma de la constante menos uno consigo es equivalente a la negada del número cuyo argumento es la suma de uno más uno.
RE36	`*[_a, &MONE[]] >>> -[_a]`	La multiplicación de un elemento y la constante menos uno es equivalente a la negada del elemento.
RE37	`_a >>> *[&MONE[], -[_a]]`	Un elemento es equivalente a la contante menos uno por dicho elemento.
RE38	`*[#[_a], -[#[_b]]] >>> -[#[operate("_a*_b")]]`	La multiplicación de dos números, siendo el segundo negativo, es equivalente a la negación de un nuevo número cuyo argumento es la multiplicación del de los dos primero.
RE39	`=[+[_a,-[_b]], _c] >>> =[_a, +[_c,_b]]`	La igualdad entre la suma de dos elementos, uno negado, y un tercero elemento es equivalente a la igualdad entre el no negado y la suma de los otros dos elementos.
RE40	`=[_a,_b] >>> =[&ZERO[], +[_b,-[_a]]]`	La igualdad entre dos elementos es equivalente a la igualdad entre la constante cero y la suma del primer elemento más el negado del segundo.

Tabla 7.1-1. Reglas de equivalencia para Chalkpy.

7.2 Ejemplo

En esta sección se muestra un ejemplo de la resolución de una expresión matemática, paso por paso, utilizando el motor Chalkpy. En casa paso se muestra cómo está la ecuación, qué regla se va a aplicar y a qué elemento. Esto último se hará resaltándolo en azul. Las reglas se referenciarán por el identificador de la Tabla 7.1-1, o por su función si es una primitiva.

La ecuación que se va a representar es: $(x * 6) + (4 * x) = 2 * 7$

$$= \left[+ \left[* \left[\$[x], \#[6.0]\right], * \left[\#[4.0], \$[x]\right]\right], * \left[\#[2.0], \#[7.0]\right]\right] \quad \text{Conmutativa}$$

$$= \left[+ \left[* \left[\$[x], \#[6.0]\right], * \left[\#[4.0], \$[x]\right]\right], * \left[\#[2.0], \#[7.0]\right]\right] \quad \text{RE21}$$

$$= \left[* \left[+ \left[\#[6.0], \#[4.0]\right], \$[x]\right], * \left[\#[2.0], \#[7.0]\right]\right] \quad \text{RE1}$$

$$= \left[* \left[\#[10.0], \$[x]\right], * \left[\#[2.0], \#[7.0]\right]\right] \quad \text{RE7}$$

$$= \left[* \left[\#[10.0], \$[x]\right], \#[14.0]\right] \quad \text{Conmutativa}$$

$$= \left[* \left[\$[x], \#[10.0]\right], \#[14.0]\right] \quad \text{RE34}$$

$$= \left[\$[x], * \left[\#[14.0], @INV\left[\#[10.0]\right]\right]\right] \quad \text{RE31}$$

$$= \left[\$[x], \#[1.4]\right] \quad \text{Solución}$$

7.3 Integración con la Interfaz

Finalmente, el motor de matemática simbólica para la nueva versión de Chalkpy, ha podido integrarse con la Interfaz de Usuario. Esto se ha conseguido por medio de una Interfaz de comunicación desarrollada, junto a la de usuario, en el Trabajo de Fin de Grado “Interfaz Android para la herramienta Chalkpy” [6].

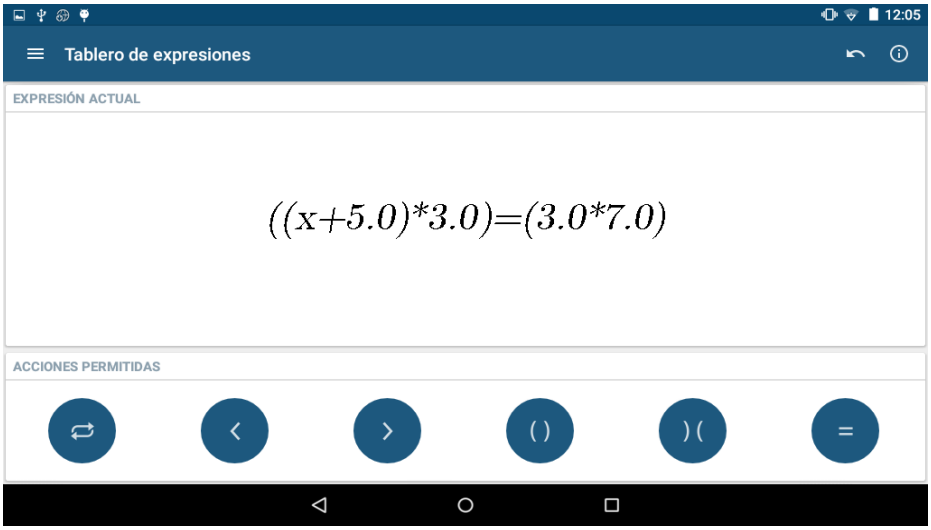

Figura 7-1. GUI diseñada para el TFG "Interfaz Android para la herramienta Chalkpy".

8 Conclusiones y trabajo futuro

Para presentar las conclusiones de este Trabajo de Fin de Grado se detallan los logros alcanzados en su desarrollo, expuesto en este documento, que han llevado a cumplir los objetivos marcados e, incluso, superar el objetivo principal de crear el motor de matemática simbólica para Chalkpy:

- Se ha diseñado una gramática para especificar un fichero de definiciones que contiene la declaración de elementos generales, para tratarlos por medio de la matemática simbólica, y reglas de equivalencia que utilicen dichos elementos.

- Se ha desarrollado un compilador, usando JFlex y CUP, que, tomando como entrada el fichero de definiciones, genera una clase Java para modelizar un Sistema de Álgebra Computacional. Es decir, se ha conseguido crear un generador de motores CAS.

- El uso del compilador permite modificar los motores de forma automática y sencilla introduciendo, o eliminando, los elementos y reglas deseadas en el fichero de definiciones.

- Como caso particular de los motores generados por el compilador, se ha construido el motor de matemática simbólica para la nueva versión de Chalkpy.

- Como prueba final, se ha integrado el motor en una aplicación Android desarrollada en el TFG "Interfaz Android para la herramienta Chalkpy" [6]. Esto permite que un usuario efectúe manipulaciones en diversas expresiones que son realizadas por el motor.

Incluso habiendo alcanzado estos logros se pretende seguir con el desarrollo de este trabajo. Esto está motivado por la posibilidad de poder seguir ampliando la funcionalidad y capacidad tanto del compilador como de los motores que puede generar. Algunas mejoras que se pueden realizar en el futuro incluyen:

- Explorar las posibilidades de la herramienta para generar motores en otro contexto.

- Poder introducir condicionales en las reglas de equivalencia para utilizar ciertas operaciones como los sumatorios.

- Controlar que si existen equivalencias repetidas en el fichero de definiciones solo se genere una vez la misma reducción.

- Introducir la posibilidad de que el sistema resuelva las expresiones sin la necesidad de manipulaciones por parte del usuario.

Referencias

[1] Convocatoria de Proyectos de Innovación Docente-UAM, *ChalkPy2.0: Extensión de la herramienta ChalkPy para la presentación de contenidos matemáticos mediante dispositivos móviles (EPS_05.15)*, 2015-2016.

[2] SymPy. URL: http://www.sympy.org/es/. [Último acceso: 23 06 2016].

[3] Convocatoria de Proyectos de Innovación Docente-UAM, *ChalkPy: Una herramienta para mejorar la presentación de contenidos matemáticos en el aula*, 2014-2015.

[4] G. S. Pantoja, *Herramienta para mejorar la presentación de contenidos matemáticos en el aula*, 2016.

[5] B. Buchberger, G. E. Collins, R. Loos y R. Albrecht, *Computer Algebra: Symbolic and Algebraic Computation*, Springer-Verlag, 2012, p. 283.

[6] R. d. B. García, *Interfaz Android para la herramienta ChalkPy*, Julio 2016.

[7] F. Baader y T. Nipkow, *Term Rewriting and All That*, Cambridge University Press, 1999, p. 316.

[8] Prolog, . URL: https://es.wikipedia.org/wiki/Prolog. [Último acceso: 23 06 2016].

[9] Haskell. URL: https://www.haskell.org/. [Último acceso: 23 06 2016].

[10] LISP. URL: https://es.wikipedia.org/wiki/Lisp. [Último acceso: 23 06 2016].

[11] H. Barendregt, W. Dekkers y R. Statman, *Lambda Calculus with Types*, Cambridge University Press, 2013, p. 833.

[12] D. J. Struik, *A Concise History of Mathematics*, 4 ed., Courier Corporation, 2012, p. 256.

[13] D. Hofstadter, *Godel, Escher, Bach: un Eterno y Grácil Bucle*, Tusquets Editores, 2007, p. 920.

[14] A. M. Turing, *On computable numbers, with an application to Entscheidungsproblem*, Proceedings of the London Mathematical Society, vol. 43, nº 2, pp. 230-265, 1936.

[15] REDUCE. URL: http://reduce-algebra.com/. [Último acceso: 23 06 2016].

[16] J. Marti, A. C. Hearn, M. L. Griss y C. Griss, *The Standard Lisp Report*, AMC SIGPLAN Notices, nº 10, pp. 48-68, 1979.

[17] Mathematica. URL: https://www.wolfram.com/mathematica/. [Último acceso: 23 06 2016].

[18] WolframAlpha. URL: https://www.wolframalpha.com/. [Último acceso: 23 06 2016].

[19] YACAS. URL: http://www.yacas.org/. [Último acceso: 23 06 2016].

[20] I. Sommerville, *Ingeniería del software*, Pearson Educación, 2005, p. 687.

[21] Java. URL: https://www.java.com/es/. [Último acceso: 23 06 2016].

[22] Android. URL: https://www.android.com/. [Último acceso: 24 06 2016].

[23] JFlex. URL: http://jflex.de/. [Último acceso: 22 06 2016].

[24] P. Linz, *An Introduction to Formal Languages and Automata*, Jones & Bartlett Publishers, 2011, p. 437.

[25] CUP. URL: http://www2.cs.tum.edu/projects/cup/. [Último acceso: 22 06 2016].

[26] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2007, p. 1009.

[27] Eclipse. URL: https://eclipse.org/. [Último acceso: 24 06 2016].

[28] Bitbucket. URL: https://bitbucket.org/. [Último acceso: 24 06 2016].
[29] SourceTree. URL: https://www.sourcetreeapp.com/. [Último acceso: 24 06 2016].
[30] Cacco. URL: https://cacoo.com/. [Último acceso: 26 06 2016].

Glosario

Expresión Cadena de símbolos que cumple ciertas reglas y admite una interpretación matemática.

Regla de equivalencia Fórmula que permite la transformación de una expresión matemática a otra con igual solución.

Sistema de Álgebra Computacional Herramienta que permite realizar manipulaciones, sobre expresiones matemáticas, mediante el uso de reducciones o reglas de equivalencia.

Token Elemento más básico sobre los que se desarrolla la traducción por parte de un compilador. Son componentes léxicos que tienen significada enmarcados en un lenguaje de programación.

Anexos

A Expresiones regulares

En la Tabla A-1 se da una relación de las expresiones regulares utilizadas, en este trabajo, y su significado.

Expresión regular	Significado
`[A-Z]+`	Una o más letras mayúsculas.
`[a-z]`	Exactamente una letra minúscula.
`("[^\"]+")`	Cadena de uno o más caracteres, excepto las comillas, entre paréntesis y comillas.
`"[^\"]+"`	Cadena de uno o más caracteres, excepto las comillas, entre comillas.
`[ \t]`	Carácter de espacio, tabulador.

Tabla A-1. Significado de las expresiones regulares utilizadas