Computer Algebra System

Dimitrios Dedoussis

Supervisor: Dr Victor Khomenko

School of Computing Science
Newcastle University

This dissertation is submitted for the degree of
MSc in Computer Science

September 2017

Abstract

Symbolic computation, also called computer algebra, is a field of computational mathematics, which refers to algorithms and software systems designed explicitly for the manipulation of symbolic expressions. Computer algebra systems are software packages specialised in symbolic computation. Apart from numerically evaluating algebraic expressions, these systems are capable of applying transformations such as differentiation or simplification on a given expression. This project investigates the extend to which this transformation feature can be approached from a more dynamic and user-oriented perspective. Rewriting methods are employed to allow the user to define a customised transformation applicable on any symbolic expression. The system implemented enables the user to represent, handle and manipulate symbolic expressions using custom transformations within a friendly and functional GUI environment.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation represents my own work in accordance with the requirements of the MSc in Computer Science of Newcastle University. This dissertation contains 18,173 words including appendices, bibliography, footnotes, tables and equations.

Dimitrios Dedoussis
September 2017

Acknowledgements

I would like to thank Dr Victor Khomenko for his supportive attitude and his guidance throughout the entire summer term. Furthermore, I would like to thank Dr Ellis Solaiman and Mr Daniel Nesbitt for their administrative assistance for the dissertation module . Finally, I would extend my gratitude to the School of Computing Science for providing me with the necessary resources and workspace to make this project a success.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Scene Setting - Project Specification

Computational mathematics is a field that involves mathematical research in scientific areas where computing is of vital role. By the early 1950s, computational mathematics emerged as a distinct topic of applied mathematics, which focused on algorithms involving numerical methods and symbolic computation. Today, areas such as computational geometry, algorithmic game theory, logic, cryptography and many more belong to the sphere of computational mathematics, making it an impressively impactful and interdisciplinary science.

Computer algebra, also called symbolic computation, is one of these branches that arose within the field of computational mathematics. It refers to the study of algorithms or software systems developed explicitly for the manipulation of symbolic mathematical expressions and objects. The term symbolic is of crucial importance as it emphasises the exact computation with expressions containing variables that are regarded as symbols (i.e. hold no given value). It has to be noted that computer algebra and symbolic calculations differ from numeric computation which uses approximate float point numbers.

Any software specialised for symbolic computation applications is called a Computer Algebra System. The primary goal of every Computer Algebra System is to automate and simplify laborious algebraic tasks. The difference of such a system with a conventional calculator is the ability to handle expressions symbolically rather than numerically. Manipulation of symbolic expressions in any scientific field can be a very resource demanding and computationally challenging task. Therefore, a computer algebra system is usually software of high complexity, that is composed by versatile applications and features which will be discussed in more detail in the subsequent chapters.

Computer Algebra systems are widely used for mathematical experiments as well as design of formulae involved in numerical procedures. They were first introduced in

the 1960s and evolved rapidly to satisfy the increasing demands of theoretical physicists and Artificial-Intelligence researchers. One of the first symbolic mathematics programs, focusing on high-energy physics, was Schoonschip developed by Dr. Martinus Veltman, later Nobel Prize winner. It was not until the 1980s when industry giants such as Hewlett-Packard and Texas Instruments included such systems in their products. As of today, Mathematica and Maple are considered to be the most common Computer Algebra Systems, used by research scientists and engineers. Over the last decades, these systems have not only revolutionised the use of calculus in Mathematics and Physics research, but also altered the education, shaping new approaches of Algebra learning.

The significance Computer Algebra System along with its intriguing complexity, made it an excellent topic for this dissertation project. More specifically, this dissertation concentrates on the development of a Computer Algebra System, which focuses on symbolic transformations, and exhibits high usability, simplicity and customisability. Following an extensive literature review and a requirement analysis, such a system was designed and proposed. As the development stage started entirely from scratch, the project required a solid mathematical and computing background.

The core of any Computer Algebra system is the comprehension and storage of the expression (data structure). In order for a routine to even begin manipulating an expression, it first needs to understand it and store it in memory. Therefore, the initial stage of development was allocated to the choice of design patterns and the implementation of data structures and hierarchy of the system. Abstract Syntax Tree was chosen as the internal data structure of our Computer Algebra System. Everything, from numeric constants, variables and operators to entire expressions and transformations, is stored and computed in the form of trees. A syntax tree is a linked data structure containing a single root node, with references to other children nodes and so on, resulting to a branched data structure. Such structures are widely used in compilers.

On the second stage, a user programming language and an interpreter were developed in order for any user text input to be converted to node-tree structures (i.e. a format that can be manipulated by system's routines). However, the most characteristic aspect of this system is the ability to construct custom transformations, a feature that offers endless possibilities to the user. Any custom set of symbolic equations (rules) can be provided by the user to form a unique transformation. This is achieved using a rewriting algorithm, which replaces sub-trees of an expression with other matching sub-trees specified by the respective transformation rule. The rewriting method will be discussed in detail later in the report.

The final part was to introduce a friendly GUI environment to the system, as well as perform testing and evaluation of the final end product of this project.

1.2 Aim of Project

The overall aim of this project was to develop a computer algebra software system that performs symbolic transformations (such as simplification and differentiation) of algebraic expressions. The user should be able to input an expression in common infix algebraic notation and instruct the program on how to manipulate it. The system should be capable of transforming any given expression with regards to a specified transformation, and in cases of numerical instances, evaluate. Another important objective, was to grant the user a sense of freedom and customisability. With the aid of a user programming language and a specialised interpreter, the application should let the user insert a custom set of equation rules, and construct any unique transformation. For the purposes of these feature, formula rewriting methods were to be understood and employed. Finally, it was intended to enclose the whole system in a user-friendly GUI application, equipped with a fully operating terminal as well as other appropriate features.

1.3 Project Objectives

In order to achieve the above aim and deliver a successful system that focuses on custom transformations, the project had to be organised in a series of clear stages. The following technical objectives were specified to allow the implementation of a solid practical solution.

- Carry out extensive research on the fields of symbolic computation and rewriting. Investigate state of the art Computer algebra systems.
- Conduct literature review on parsing as a technique and parsers as tools. Achieve a high level of familiarisation with ANTLR4 (parser generator).
- Plan and implement the design of the object oriented system. Develop the classes that compose the desired tree data structure along with their hierarchy. Develop auxiliary routines that complement the designed tree data structure.
- With the aid of a grammar syntax and a parser/interpreter, develop the specialised user programming language of the system. This language (different than the language of implementation) will be employed for the conversion of user text input, to node-tree structures than can be processed by the system and vice versa. Therefore, it must serve as an interaction medium between the user and the system.
- Develop functions that transform and evaluate any give node-tree expression.

- Enable custom transformations by implementing tree rewriting functionality. This involves the construction of a routine that determines whether two tree-nodes match, as well as a routine that replaces/rewrites a node with its matching one.
- Design and implement a friendly and usable GUI that embeds the full system. The heart of the GUI application should be a fully functional terminal, with programmed commands.
- Perform unit testing on every component. Explore the limits of the system by laborious system testing. Evaluate the system.

On a personal level, objectives include:

- Achieve a substantial background and experience level on parsing as well as tools and methods associated to it, such as ANTLR4.
- Expand knowledge and improve skills in areas such as Object Oriented software, symbolic computation and design patterns.
- Enhance various professional and academic soft skills. Time management, efficient communication and scientific research compose my main focus. Improving software engineering skills is another objective, achievable by the appropriate selection and application of a development model. Frequent supervisor meetings should also reinforce communication skills and assist critically via feedback and guidance provision.

1.4 Project Scope

A scope of work for this project has been shaped after gathering and analysing all the requirements, the majority of which were successfully met.

The project had to be designed, developed and delivered during a 3 month period (from June to September of 2017). It involves symbolic transformations of any algebraic expression. Expressions can be variables, constants, operators, other expressions or (as in most cases) a combination of each. Numeric constants support both integers and floating values as originally planned. Trigonometric and exponential functionality is achieved. The user can specify custom operations and transformations.

Assumptions:

- The manipulation of the node-tree structures obeys a top-to-bottom traversing approach. In the majority of the transformation cases, this strategy appears to be the most efficient. However, in transformations such as simplification, bottom-to-top strategy is optimal. This assumption is going to be decomposed in detail in chapter 3 of the report.
- Symbolic variables accept only alphabetical values.

1.5 Project Management

The software engineering development model employed in the context of this project was iterative prototyping. After initial requirement analysis and research, a preliminary terminal-based prototype was built, which continuously evolved through consecutive iterations. At the point when the prototype product was satisfactory, the project concluded with the stages of testing/verification and evaluation.

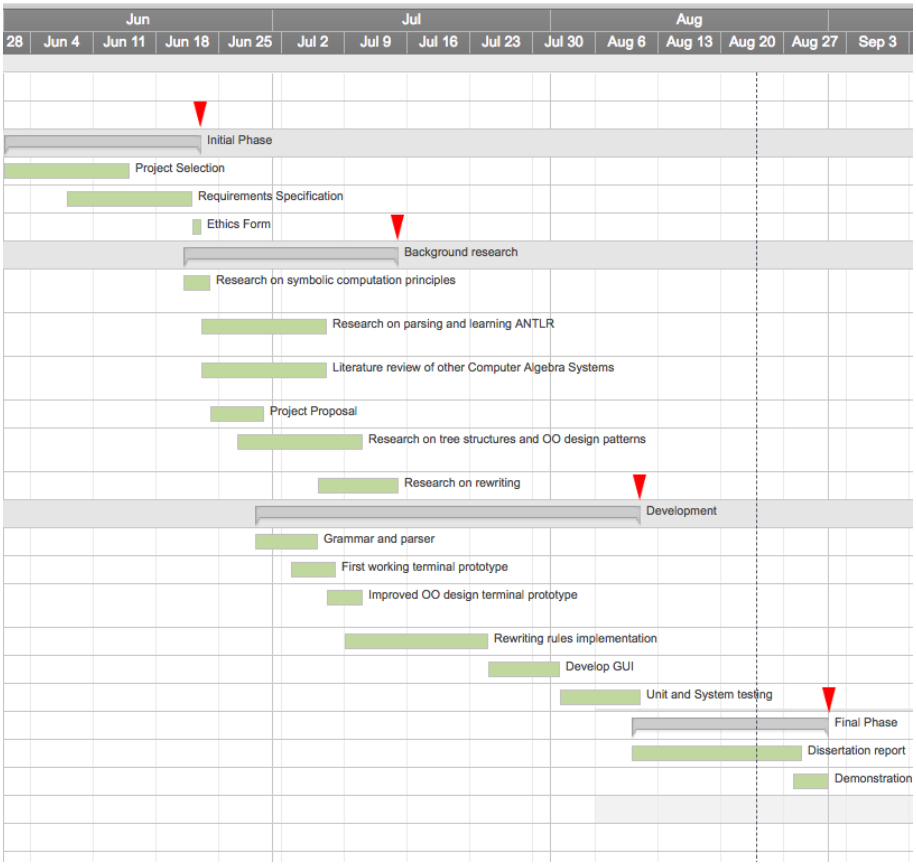

Fig. 1.1 Gantt chart. The red pointers indicate project milestones.

1.6 Outline

This report consists of 5 main chapters.

- Chapter 1 is an introduction. Provides a scene setting, explaining the problem investigated and outlining the main aspects of the project.
- Chapter 2 includes a background research along with a literature review concerning state of the art existing technologies. Technical material and tools used are introduced.
- Chapter 3 presents all the software engineering aspects of the project. Design, implementation and testing strategies are explained in high-level detail.
- Chapter 4 displays the results of implementation along with an evaluation of the whole system.
- Chapter 5 discusses the extent to which initial objectives are achieved, in a conclusive manner. It is a retrospective look back describing what was learnt, what were the positive and negative elements of this effort. Future recommendations are included.

Chapter 2

Background Research and Technical Material

2.1 Literature Review

Developing a Computer Algebra System from scratch was a highly perplexing task which required extensive background research. The literature review was directed towards four main topics. In order for the project to have a solid foundation, these topics were investigated meticulously as presented below.

2.1.1 Symbolic Computation

As explained in the preceding chapter, computer algebra systems can perform both symbolic and numerical calculations. Symbolic computation can be defined as computation with symbols that represent mathematical objects. It involves the manipulation of symbolic expressions using mathematical operations. These range from standard operations such as differentiation, numerical evaluation and simplification to highly structural ones such as expression rearrangement and extensive list manipulation [2].

In symbolic computation, it is essential to specify and understand how data is stored and represented. Symbolic expressions consisting of numbers, variables and operators, are regarded as the data of computer algebra. Optimal representation of symbolic expression data varies and depends on the manipulator.

Human users are familiar with various notations [3]:

- **Infix notation:** The most common notation used in arithmetical and logical formulae. Its is characterised by operators being placed between operands (3+5). Although it is inconvenient to be parsed by computers it is widely used due to its familiarity.

- **Prefix (polish) notation:** Notation where operators precede operands (+3 5). It is a parentheses-free and computer friendly notation. Polish notations were invented by Prof. Jan Łukasiewicz in 1920s.
- **Postfix (reverse polish) notation:** Operators follow operands (3 5 +). Often used in stack-oriented programming languages.

Computer systems, on the other hand manipulate expressions optimally when these are represented in a tree data structure. The tree is a hierarchical linked data structure that is generated from a single root node. The root node contains data for its own entry and reference pointers to one or more children nodes. Similarly, the children nodes may reference to children of their own, producing a tree linking pattern. This allows items of an expression (numbers, variables, operators polynomials etc.) to be hierarchically referenced with one another.

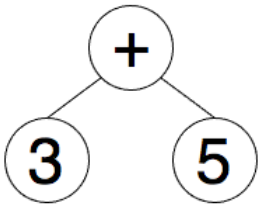

Fig. 2.1 Node tree example, where the operator "+" is the parent, and the operands 3 and 5 are the children.

Every node of the tree (apart from the root) has one parent. Terminal nodes, ie. nodes that have no children, are called leafs. Every non-terminal node represents an operator or a function, with their children being the operands. Every leaf node represents terminal symbols of an expression (they cannot have any children) such as numbers and variables. The hierarchy of operations and operands within the syntax tree structure clearly specifies the precedence of operations. Syntax trees offer very high flexibility, representing a great variety of symbolic expressions. However, the same symbolic expression can be represented with multiple syntax trees. Aside from this disadvantage, the syntax tree is a highly unambiguous and computer-friendly method of storing symbolic expressions

Memory allocation is a critical factor in symbolic computation. Computer algebra emphasises on exact computation with strictly represented data. This means that even when the input/output of the system is of modest size, large memory working space is required. This phenomenon is described by Tobey et. al. [4] as *intermediate expression swell* and exists from the earliest computer algebra systems. Expressions may expand to a large extend during intermediate stages of a calculation, significantly increasing the memory use. This is a very common vulnerability of computer algebra applications and should be considered in the design and development phases of this project. [4]

The number system is another topic worth investigating. Floating point numbers and integers of fixed bounded size are sufficient for numerical computation. However, this is not the case in symbolic computation due to large rounding errors and the expression swell problem. It is very common in computer algebra, to use the GMP library (library for arbitrary-precision arithmetic), which is considered as a genuine standard.

In general, numerical procedures are based in the theory of approximation using finite precision arithmetic. Symbolic computation is more resource intensive but also more precise and accurate. In principle, an algebraic procedure does not trade mathematical accuracy of output for efficiency to get it approximately. [5]

Part of the mathematical background research is exploring aspects such as equality, associativity and commutativity. For example, a standard way to deal with associativity is to allow operators of addition and multiplication have an arbitrary number of operand children (and not just two). This is also known as the levelling operators method. Equality, on the other hand, is a more complicated problem. There are two kinds of equality in mathematical expressions, syntactic and semantic. Syntactic refers to the representation-form of the expression while semantic refers to the true content of the expression [6]. Syntactic equality, although trivial, is simple to test. Semantic equality exists when two expressions represent the same mathematical object (ie. $a^{3+m} = a^3 \times a^m$). Testing semantic equality is a more complex, and in cases of exponential or logarithmic expressions impossible (Richardson's Theorem [7]). Therefore, semantic equality can only be tested in expressions such as polynomials and rational fractions.

Canonical order is a notion that allows equality testing of two expressions. In order to confirm the equality of $a+b=b+a$, a canonical ordering of terms needs to be defined. Thus, arranging a tree structure in canonical order, needs all children of a commutative node to be sorted using an ordering function. Sorting rules could involve type of child (variable or number), or alphabetical ordering.

2.1.2 Parsing

In order for symbolic expression data to be converted from human readable format (usually infix notation) to a computer optimised format (syntax tree structure) a procedure called *parsing* is required. Parsing is the process of analysing a string of symbols, with regards to the rules of a formal grammar. It is the separation of a string in order to enclose it in a more structured and semantic arrangement.

A *parser* is a software component that performs parsing. It receives text as input data and stores its semantic content in a data structure. In the case of computer algebra, parser is an important constituent of the system, as it converts text input (infix symbolic expressions) to abstract syntax trees ready to be manipulated by the computer. The rules under which

this conversion occurs are precisely specified within an agreed grammar. The first step of parsing is the lexical analysis, a process that uses regular expressions to split the input character stream into meaningful symbols declared in the grammar (*tokens*). The program that tokenises the input is called *lexer*, and is responsible for grouping the related token classes, such as integers and identifiers. Then, the parser program runs the syntactic and semantic analysis stages of parsing, resulting to a structural representation of the input.

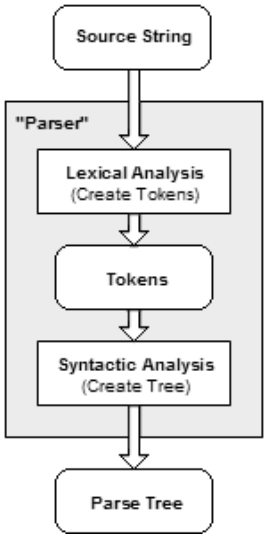

Fig. 2.2 Stages of parsing. Source Wikipedia (Parsing)

Top-down parsers and bottom up parsers are the two main types of parsers, both aiming to determine whether the string input can be derived from the starting symbol of the grammar.

Implementing a parser and a lexer analyser out of a specified grammar is a laborious task to undertake. Software tools called *parser generators* automate this tedious process. One of the earliest and surprisingly powerful parser generator was META II, developed in 1964. Since then many different instances of parser generators have emerged. For the purposes of this project, two well established parser generators were considered:

- ANTLR4
- JavaCC

ANTLR4 was chosen over JavaCC, due to its strong IDE integration and very well written documentation (there is a reference book provided by the author of ANTLR [8]). ANTLR is also widely known for generating human-readable output and being actively supported.

2.1.3 Computer Algebra Systems

A Computer Algebra System (CAS) is mathematical software that manipulates symbolic expressions. They are classified into two different groups:

- **Specialised:** Focusing on a specific part of mathematics, such as number theory or group theory.
- **General purpose:** Target group of users that work in any scientific field involving manipulation of symbolic expressions.

There are several fundamental features that are required in any general purpose CAS, for the computation of symbolic expressions:

1. User interface allowing the input and output of mathematical objects.
2. A large set of common mathematical routines (algorithms).
3. Memory manager, for the high resource demands during intermediate calculations.
4. Rewriting system.
5. A programming language, a parser and an interpreter.

To overstate the last point, it must be clarified that computer algebra systems possess two different languages. The internal language of CAS is the language of implementation (ie. the language in which the system is written). However, the end-user of the CAS also need to perform some kind of programming, to specify expressions and transformations. Therefore the external language of the CAS intended for user interaction. The external programming language also requires its own parser and interpreter within the system. [9]

As mentioned in chapter 1, development of CAS software emerged in the 1960s. Lisp (programming language) has been used for the development of many early suuch systems. MATHLAB is an example, created in 1964 within the AI research environment of MITRE. The first popular computer algebra systems were muMATH, Reduce and Macsyma. A popular version of Macsyma, named Maxima is still actively being maintained. Currently, Mathematica and Maple are considered to be the most popular commercial systems, widely used by the scientific community throughout the world (both are general purpose). It also has to be underlined that many established systems such as Axiom and SageMath are classified as open source.

Both Maple and Mathematica are generally very similar and extremely powerful computer algebra systems. Mathematica features a very strong definite and indefinite integrator, while Maple offers extraordinary ODE and PDE solution methods. As programming

languages they differ with each other. However, they both offer a broad collection of features, ranging from simple numeric and symbolic computation to data visualisation and signal processing [10].

2.1.4 Rewriting

A typical CAS applies a large collection of rules for transforming expressions. These rules could range from standard algebra rules to to much more advanced rules involving higher mathematical functions. The general transformation principle is simple. System receives an input expression on which successive transformation rules are applied, with the procedure terminating when no further rules are viable. [11]

The process involved in the application of transformation rules is called rewriting. Rewriting systems implement methods of replacing sub-terms of a formula with other terms. A rewriting system is essentially a set of term-rewriting rules. Each rule is of the format $LHS \rightarrow RHS$ and can be applied to a term s, if the LHS of the rule matches the term. In the case of a match, then the term s is rewritten by the corresponding RHS of the rule. If there is no match, then the next rule of the rewriting system is to be applied. As the application of rules in a rewriting system is sequential, it is very important to specify the order of rules correctly.

The concept of term-rewriting was first introduced by Evans and Knuth-Bendix. Originally, it was proposed for canonical term rewriting utilised as a decision procedure in theories where equality was to be validated [12]. A popular system that heavily focuses on rewriting is Maude, a free software developed in 1997. Maude is a high-level programming language and a high-performance system. The main advantage of Maude is that besides supporting equational computation and algebraic specification, it also enables rewriting software computation. Examining and understanding a powerful system such as Maude can provide useful insight into rewriting methods and applications. [13]

Every transformation within a CAS can be implemented as rewriting rule system. This methodology was applied for the transformation customisability aspect of this project, as explicitly discussed in chapter 3.

2.2 Technologies Used

2.2.1 Java and Swing

The Computer Algebra System within the context of this project was implemented entirely in Java. Java is a high-level general purpose programming language, which is also object-oriented, concurrent and class-based. It was particularly designed to reduce implementation

dependencies to a minimal extend. Java was originally developed by Sun Microsystems (now owned by Oracle) in 1995, and is currently one of the most popular programming languages, well-known for its simplicity, robustness and security.

Java version 1.8 including the Java SE Runtime Environment and Development Kit (JDK) was used for the development of this system. Application requires JRE SE 1.8 to be installed by the end-user.

For the design and implementation of the system's GUI, the Swing library of the Java SE platform was utilised. Swing is the most updated and powerful GUI widget toolkit for Java, highly adaptable across different platforms.

The IDE used for the Java development of this system was Eclipse Neon 4.6. It was preferred due to its extensive plug-in customisability system as well as its popularity across Java developers.

2.2.2 ANTLR4

ANTLR is a powerful parser generator featuring the reading, processing execution or translation of structured text and binary files. It was developed by Prof. Terence Parr in 1992. As of today, ANTLR is still actively being maintained (open-source) and is widely used in academia and industry for the development of all sorts of languages, tools and frameworks. It is integrated in engines and systems of high-profile industry giants such as Twitter and Oracle.

ANTLR receives a formal language description, called grammar, as input and generates a praser for that language that can automatically build parse trees. These trees represent how grammar matches the input. ANTLR can generate lexers, parsers, tree-parsers as well as combined lexer-parsers. After tokenising the user text input with the generated lexer, ANTLR uses these tokens to recognise the string structure. ANTLR-generated parsers build a syntax tree structure, that records how the parser recognised the format of the input sentence and its components. By producing a syntax-tree automatically, ANTLR delivers a convenient data structure containing semantic information, to the rest of the application [8].

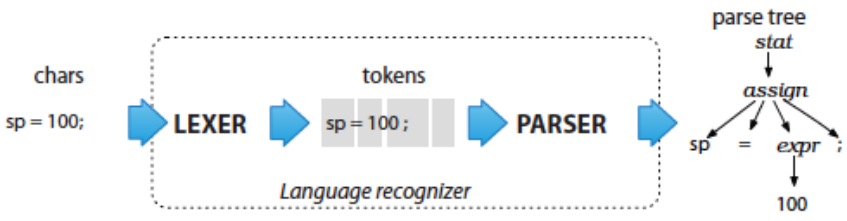

Fig. 2.3 Stages of ANTLR flow. Source refrence.

ANTLR operates on extended Backus–Naur form (EBNF) context-free grammars. EBNF grammars provide optional statements, repetition and grouping. The core-component of a context free grammar is the set of rules. Each rule consists of two parts: name and expansion of the name. Below is an example of such a grammar:

$$expr \rightarrow term + expr$$
$$expr \rightarrow term$$
$$term \rightarrow term \times factor$$
$$term \rightarrow factor$$
$$factor \rightarrow (expr)$$
$$factor \rightarrow const$$
$$const \rightarrow integer$$

ANTLR4 is the latest powerful version that will be used within the scope of this CAS. ANTLR4 uses the adaptive LL(*) parsing strategy that performs grammar analysis dynamically at runtime. Another crucial feature of ANTLR4 is left-recuursive nature of the self-referential rules, which proves to be very efficient for the matching of arithmetic syntactic structures.

```
expr : expr '*' expr  // match subexpressions joined with '*' operator
     | expr '+' expr  // match subexpressions joined with '+' operator
     | INT            // matches simple integer atom
     ;
```

Fig. 2.4 ANTLR4 automatically rewrites left-recursive rules such as expr into non-left-recursive equivalents. [8]

LL(k) parsers are top-down parsers that use a lookahead of k tokens when parsing a sentence. Lookahead defines the maximum number of incoming tokens utilised by the parser for the appropriate rule selection. Lookahead parsers make decisions by comparing the initial symbols of each alternative. If the lookahead token (next input token) meets more than 1 rules, the parser needs more lookahead tokens to understand which alternative will succeed. Therefore, the adaptive LL(*) strategy of ANTLR4, makes the lookahead coefficient adjustable, automatically changing its value with regards to the necessities of every decision, resolving any ambiguities [8].

Finally, ANTLR4 offers useful tree-walking mechanisms within its runtime library. Visitor and Listener are two design patterns that respond to events triggered by the build-in tree walker.

2.2.3 Complementary Material

For the purposes of system testing and evaluation, external symbolic and numerical computation systems were used. These included MATLAB (numerical) and Wolfram Alpha (symbolic). This complementary software assisted with the verification and validation of transformation and numerical evaluation results.

2.3 Summary of Chapter

This chapter covered the background research and technical material review of the project. Initially the focus of research was on symbolic computation as a science, followed by a literature review regarding of the state of the art Computer Algebra Systems. At this point it was well understood that the areas of parsing and rewriting will be critical for the success of this project. Further investigation of these two topics was undertaken, presenting the important aspects and analysing existing work. Java was chosen as the most suitable language of implementation for this system, with the IDE of Eclipse being preferred. Concerning the parsing generator, ANTLR4 was the favourable solution. The background material and knowledge collected within this phase of the project was vital for the succeeding stages of design and development.

Chapter 3

Development

3.1 Analysis and Design

In the software development life-cycle the early stages of planning, analysis and design are of critical importance as they establish a solid foundation for the successful implementation of the system. The significance of efficient design was highly valued in the context of this project, and was not outweighed by the implementation and coding phases. The following sections include a break-down of the requirements engineering procedure, an overview of the design, the feature specification of the system and finally a "look back" on the development process model employed.

3.1.1 Requirements engineering

Following the comprehensive background research and literature review presented in Chapter 2, a requirement analysis was conducted taking into consideration the available resources as well as the target user-base. Usability, accuracy and extensibility were three aspects that were highly valued during this process. Requirements are classified as functional and non-functional and range from low to high priority/importance.

Functional requirements

Functional requirements include the behaviour and the function of the system and are presented in Table 3.1.

Table 3.1 Functional requirements of the system, and associated priority

Requirement	**Priority**
Input Accept any symbolic infix expression as input. Expressions should be able to include algebraic symbols, trigonometric functions, operators and parentheses. The system should also allow user-specified functions followed by the respective parameters.	High
Parsing System requires a parser that interprets infix user input and generates a syntax tree structure containing all the semantic and syntactic information of the given expression.	High
Tree data structure The design and implementation of a linked tree data structure is required in order for the system to represent and store any expression or mathematical object within such a tree. This tree data structure should exploit a parent-to-child relationship pattern to link different types of mathematical objects (nodes) together. It is crucial to maintain the appropriate precedence.	High
Rewriting system A rewriting system should be included to enable custom transformations. A tree-node matching function is required for this task. It is important that the CAS allows the user to specify the set of rules for rewriting and create a custom transformation.	High
User programming language Specifying and implementing a user-programming language is vital for the interaction between the user and the system. This language should be designed exclusively for the characteristics of this CAS. Using this simple defined language, the user can specify an input expression, or the rewriting rules for a new transformation. The language should also account for conditional statements within the rules.	High
Evaluating features • Arithmetic evaluation: Routine that performs arithmetic evaluations within tree structures is required. • Condition evaluation: Routine that evaluates the possible conditions of any rewriting rule.	High
Auxiliary features • Cannonical form: Routine that converts an expression tree to its canonical form. There is also need for the specification of canonical form. • Printing: Routine that converts a tree structure to human readable infix notation.	Medium
GUI The whole system should be enclosed within a graphical user interface. The GUI is required to provide input and output facilities, as well as menu and access for all the available tools and features of the system. The input and output fields require large size capacity, as symbolic expressions may be long and complicated. For this purpose, the core of the GUI should adopt a terminal-like philosophy. The terminal to be developed within the GUI, will serve as an interaction medium, also providing a live history of actions and events. Interface for creating and managing transformations should also be provided.	Medium
Complementary debugging features Include a feature that displays the internal structure of a ny tree expression. A tree-display that shows how how nodes are interconnected.	Low

Non-functional requirements

Non-functional requirements specify how the system exhibits these behaviours and performs these functions (i.e. the functional requirements).

- Project should be developed within the 3-month summer term period. The CAS should be delivered and demonstrated by 25/8/2017.
- ANTLR4 tool is required for the generation of the appropriate parser and lexer analyser. In this context, a well defined and unambiguous grammar needs to be developed. This grammar will also specify the user programming language of the system.
- For the implementation of the tree data structure an object oriented language, such as Java, should be employed. It is important to design a hierarchy of classes that construct node objects which represent mathematical objects and can be referenced by one-another. All nodes should be derived from the same abstract super class.
- Every transformation (set of rewriting rules) shall be represented as an object. Thus a transformation class needs to be included, which instantiates transformation objects. Each transformation object contains the appropriate set of rules.
- A rewriting rule is defined as an equality of two symbolic expressions, followed by an optional boolean condition. This requires a boolean condition evaluation procedure within the rewriting system. The appropriate modifications in the grammar and the parser of the user-programming language are required in order for conditional statements to be supported.
- Every transformation involves the iteration over a set of rewriting rules. This requires restrictions in the number of loops to be performed in order to handle infinite loop cases and reduce redundant waiting time. A number of 1000 would be a reasonable loop restriction.
- For the purposes of efficient design it is desired that expression-node classes include a minimal number of mutating functions. All auxiliary, evaluating and complementary routines should be included in separate class or classes.
- In order to avoid null exceptions the GUI is required to restrict the user from entering or manipulating null data. Consistent and accurate investigation of input through conditional statements is required. Appropriate dialog messages should prompt the user to enter valid input data.

3.1.2 Design overview

Expression Tree

The core of this system is the expression tree representation. The design of a tree data structure that will serve the needs of algebra was a critical task that was handled using an analytical approach.

The conversion of a symbolic expression to a tree representation involves breaking down the expression into its components. These include operators and operands. In the tree representation, every node of the tree represents an item/component of the expression. Hence, every node of the tree is either an operator or an operand. The interconnection of nodes within the tree resembles the relationship between these operators and operands. However, there are different types of operators and operands. There are operators that involve multiple number of operands, while others that involve only 1sone (such as unary minus). Similarly, there are countless types of operands. Constant number values and specified variables can be operands. Even operators can be operands included under another parent-operator. Essentially, any mathematical object can be an operand.

The tree structure is designed so that every operator node has a certain number of children nodes. For example, in the $1/2$ expression, the operator node "/" has two operand children. In this case both operands are number values, as below:

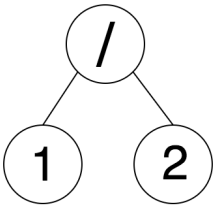

Fig. 3.1 Tree representation of "1/2" expression

In cases of unary operators, the operator node is linked to only one operand child:

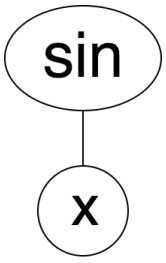

Fig. 3.2 Tree representation of $sin(x)$ expression

There could be circumstances where the expression is just an operand, such as a single number value or variable, as shown in Figure 3.3. Such expressions are handled as single node trees:

Fig. 3.3 Single node tree – represents expression of: 3

As mentioned above, operands can also be other operators. Therefore, an operator node can have another operator as a child node. For example, the expression $2 \times x + y^3$ is represented by the following tree:

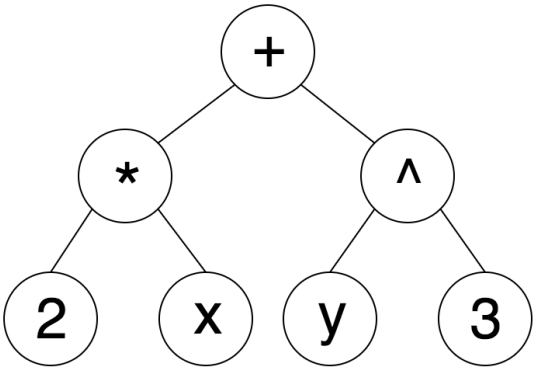

Fig. 3.4 Tree representation of $2 \times x + y^3$

The top node of a tree is called the root node, while the term *leaf* describes any terminal node of a tree. Intermediate nodes are the nodes between the root and the leafs. A tree may have no intermediate nodes (this is the case in figures 3.1, 3.2) or even no leafs (figure 3.3). As observed from the figures above, the intermediate nodes can only be operators, while leaf nodes can only be operands. The root node can be anything.

By designing the tree data structure in this way the precedence of operations is also specified. Operators that are farther from the root node are of higher precedence than operators that those closer to the root. This is vital for the mathematical accuracy of the system, as calculations involving operators of plus and minus have lower precedence than multiplication and division. This design aspect of the tree data structure also allows the implementation of parentheses functionality.

As explained, tree-nodes can vary in terms of their type and their behaviour. Thus, a hierarchical approach was adopted for the design of this data structure. In this hierarchy, there is an abstract node superclass. Every node in the expression tree is an abstract

node. More specialised node subclasses extend this higher abstractive node class. Such as example is operator nodes: all operator nodes are objects of this class. This class can define any kind of operator as there is no restriction in the number of children. Other types of node subclasses can be number nodes, which are operand nodes that only contain a numerical value. Variable nodes contain a specified string variable and dollar nodes can represent any instance of node.

Bellow is a UML class diagram displaying the hierarchy design of the expression-tree data structure of this CAS:

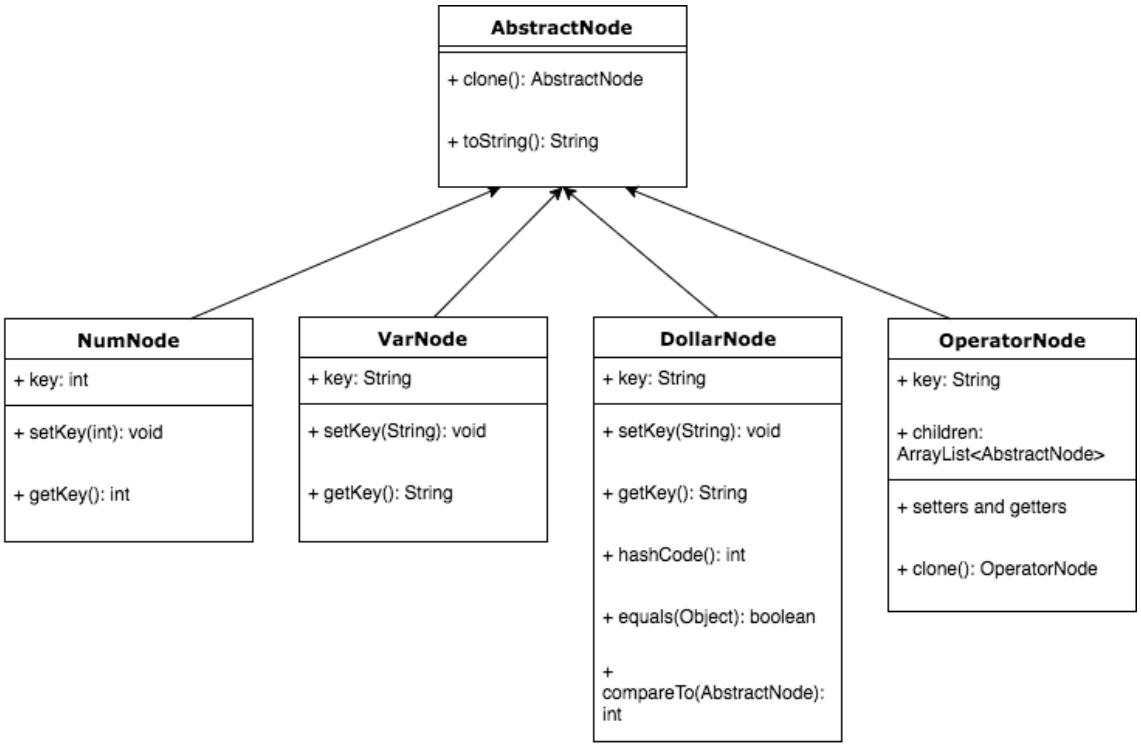

Fig. 3.5 UML class diagram of tree data structure

Each node subclass is designed to construct node objects that contain the data value of the node. This data could be a numeric value or a string depending on the type of node. In case of operator node the data of the object is the symbol of the operator. However, operator nodes also carry an array of children. Hence, operators reference the appropriate operands. It is important to note that children do not reference the parent. There is only top-down referencing. Access to the root node object allows access to all of its children node objects. Similarly, the children-nodes can reference to their own children and so own. **Therefore, when given a root-node, we can retrieve the full tree under this specific node.**

The node classes were designed on the principles of high cohesion, low coupling and simplicity. They do not contain irrelevant or redundant functions or fields, and strictly focus on the state of the respective node object.

Parsing

Designing the user programming language of the system involved a solid understanding of the ANTLR4 parser generator tool.

A grammar that instructs ANTLR on how to generate the parser and lexer analyser is required. The grammar should be able to interpret:

- **Infix symbolic expressions:** Grammar is designed to handle numerical values (0 to 9) and alphabetical symbols (lower and upper case). Arithmetical operator symbols as well as parentheses are also detectable tokens. The grammar is also designed to allow operators in the form of: operator(operands). This functionality does not only enable established operators such as sin(x) or log(x) but also custom operators such as fibonacci(x).
- **Rewriting rules that specify a transformation:** Implementing the custom operator design described above is vital for the definition of rewriting grammar rules. The grammar defines a rewriting rule as an equation of two expressions, followed by an optional boolean condition. In order for the user-programming language to be familiar, the boolean conditions were designed similarly to common programming languages. If-statement terminology was chosen. For example: $expression1 = expression2, if\ variable == constant$

The grammar was accurately structured to handle precedence, and to specify a programming language that is simple and easy to understand. The parser and lexer analyser was generated out of the designed grammar. Parser and lexer were invoked using a visitor class. Visitor design-pattern is supported by ANTLR4 and was vital for the efficient parsing of expression input. Implementation details are discussed in section 3.2.

Auxiliary routines

The system requires several auxiliary functions for the manipulation of expression trees. In order to maintain the cohesion of the tree data structure classes and maintain a simple and efficient design, these functions were included in a separate Node-Functions class. All of the functions included in this class involve the manipulation of tree-nodes.

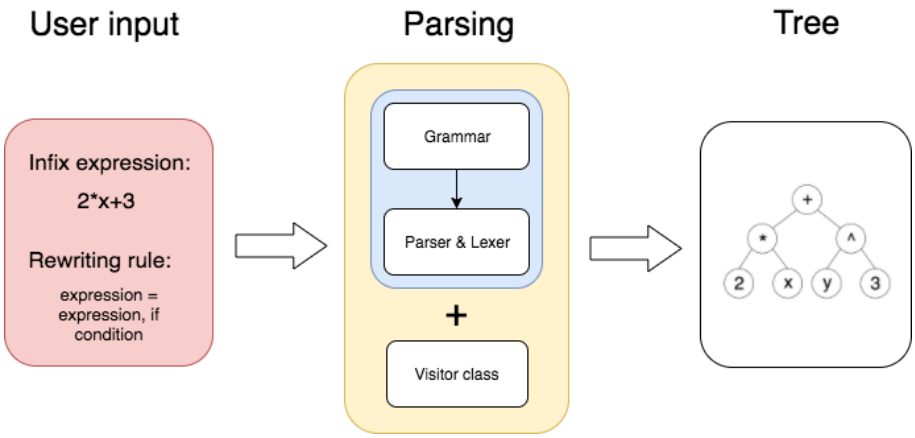

Fig. 3.6 Parsing process design

In the previous section, the procedure of converting infix user input to tree was explained. However, after the completion of manipulations, a tree should be converted back to user-friendly format, in order to be presented to the user. Hence, an auxiliary function is designed to convert a given parameter node into an infix string notation. The routine takes into account precedence, inserting parentheses in the appropriate locations.

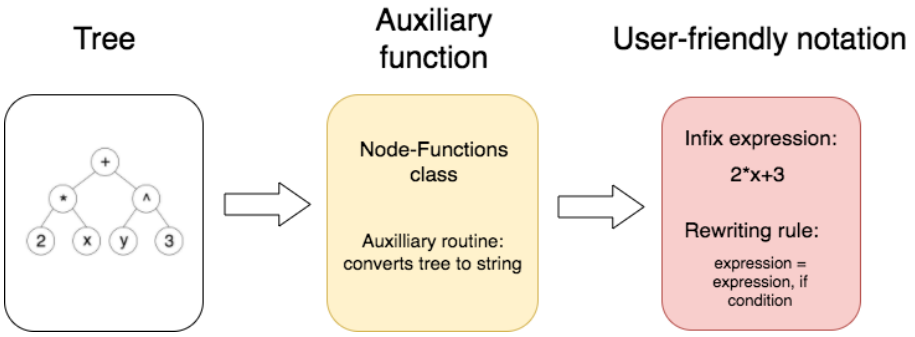

Fig. 3.7 Printing tree structure

The Node-Function class includes many other routines vital for the functionality of the system. These auxiliary methods are used on multiple instances throughout every part of the system.

Transformations

A transformation can be designed as an object that contains a list of rewriting rules. Each rule can also be defined as an object. Every rewriting rule is an equality of two expressions, followed by an optional boolean condition. Therefore a rule is conceptualised as an object of three member fields. These are the left-hand-side expression of the equation, the right-hand-side one and the boolean condition. All three of them are represented and stored as

trees. In the case that there is no boolean condition the third member field is blank. The transformation object contains only a list of rules (rule objects). Hence transformation objects have only one member field, promoting the simplicity in the object oriented design of this project.

A transform routine is designed to apply the rules of a transformation object on any given node parameter. Within this transform routine, auxiliary methods (included in Node-Functions class) are called to determine a match between the parameter node, and any rule contained in the particular transformation object. In case of match, a second auxiliary method is called to perform rewriting of the parameter node, with regards to the matching rule.

GUI application design

The terminal component is the core of the GUI application. All the other features of the GUI are designed to complement and improve the functionality of the terminal.

The aim of the design was to produce a typical command line terminal. This consists of an active string field as well as a history text field. In the active string field the user can enter input text commands which are directly appended in the history text field on every return key press. Subsequently, the history text field is updated live, maintaining a history of the input commands followed by the respective outputs. The terminal is also designed to provide timing and agent of each command. Agent could be either human user or system. A screen-shot of the terminal is shown in Figure 3.8.

An attribute of the system that was carefully considered during the design process, is the manipulation of significantly long symbolic expressions. Consecutive transformations may generate lengthy outputs. Thus, both the active string field and the history field of the terminal are large and resizeable text fields, providing enough space for the needs of the system.

An important concept in the design of this application is the *current expression*. This essentially defines the most recent expression handled by the terminal. It may either be the last expression entered by the user or the last expression generated by the system.

The main panel of the GUI does not only contain the terminal, but also the transformation list. The transformation list includes all the transformations (transformation objects) stored in the system and an example is shown in Figure 3.9. With the aid of the appropriate visual component (combobox), the user can alternate through specified transformations. After selecting a transformation, the user can transform the tree of the *current expression*. The transformed tree is returned by the system to the terminal and becomes the *current expression*.

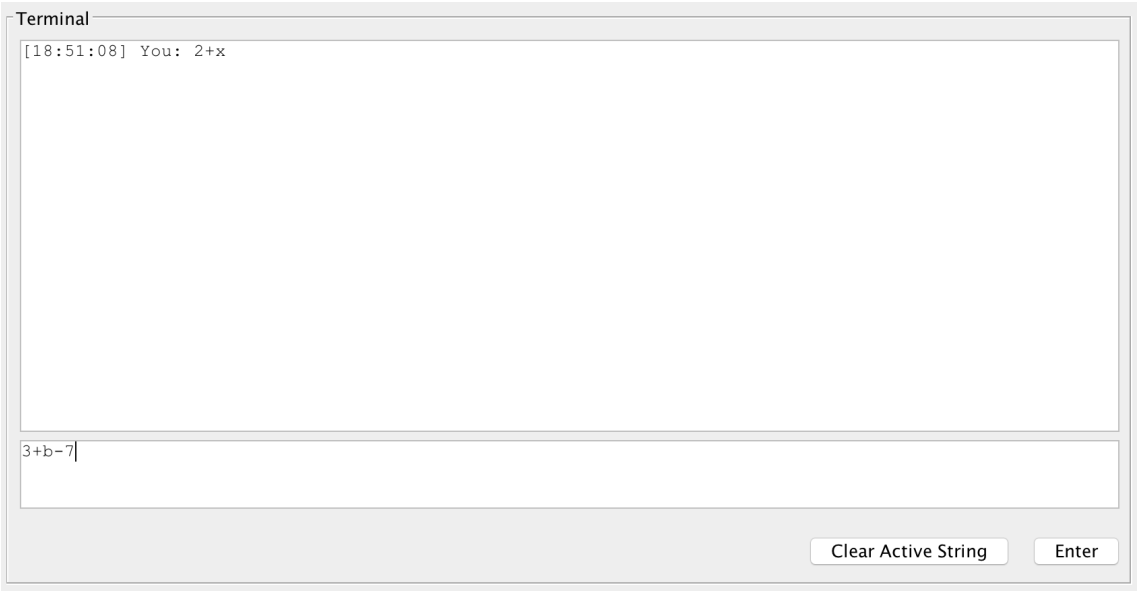

Fig. 3.8 Design of terminal. Current expression is shown.

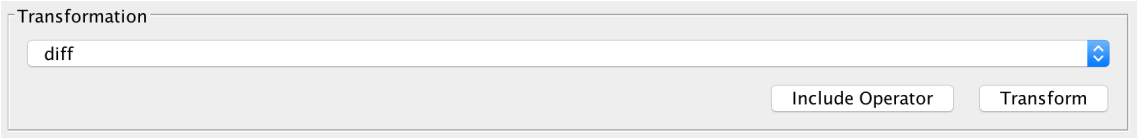

Fig. 3.9 List of transformations in the main panel. By repeatedly pressing the transform button, the current expression is consecutively transformed.

The main panel also includes a top menu bar which enables access to all other features of the system. The most important of these include the *Create new transformation wizard* as well as the *Manage transformation facility*. In the latter the user can preview, edit and delete stored transformations.

The system focuses on the customisability of transformations. It is designed so that the user has the freedom to create any kind of transformation. There is no limit to the number of saved transformations. The user can easily switch between different transformations when manipulating a symbolic expression. It is very efficient in applying successive transformations on the same tree.

The system is also designed to deny invalid input entered by the user. Appropriate messages prompt the user on how to proceed, preventing undesired exceptions.

3.1.3 System features

File

- **New**: Initiates a new CAS application frame window.
- **Import**: Imports a .txt file of a previous exported terminal history. Loads the imported history into terminal.
- **Export as .txt**: Creates a .txt file of the terminal contents.
- **Close**: Closes current CAS application frame window.

Transformations

- **New transformation wizard**: Initiates the facility where the user can insert a set of rewriting rules and create a new transformation. The transformation is stored in the system for later use.
- **Manage Transformations**: Initiates the facility where the user can preview, edit or delete already stored transformations.

Tools

- **Show tree**: Displays the internal tree structure design of the current expression.
- **Clear terminal**: Deletes terminal history.
- **Load setup**: Automatically loads the pre-set transformation objects of differentiation and simplification, storing both of them in the transformation list of the system.

Help:

- Provides links to online documentation and user manual.

About

- Presents the software credits.

3.1.4 Development process model

As described in section 1.5, the iterative prototype model was selected for the development of this project. Following some initial research and learning, the first prototype was designed and built, achieving a project milestone. As requirements and design were continuously reconsidered, new improved prototypes were being implemented, leading the final competent version. The iterations as the prototype evolved through the course of this dissertation are summarised below:

1. **Terminal interface without rewriting system**

   The first prototype of the system was a terminal application (no GUI) that supported the parsing of symbolic expressions and the formation of syntax trees. In terms of transformations, it could only perform differentiation and simplification. However, even these two were performed statically, without utilising the rewriting philosophy.

2. **Terminal interface with rewriting system**

   Again in this prototype version there is no GUI, but a simple text terminal interface. However, a working rewriting system is implemented, allowing custom transformation. System now behaves dynamically. There are also significant improvements in the object oriented design.

3. **First instance of GUI**

   The system is embedded within a preliminary GUI, that allows a better representation of the system and offers easier access to other features. History is implemented. However, this version does not include a terminal component inside the GUI. In terms of the rewriting system, boolean conditions are implemented in rules.

4. **Final GUI version**

   GUI is entirely revamped, as the terminal-like component is implemented and becomes the heart of the system. Menus are added and final back-end issues are resolved.

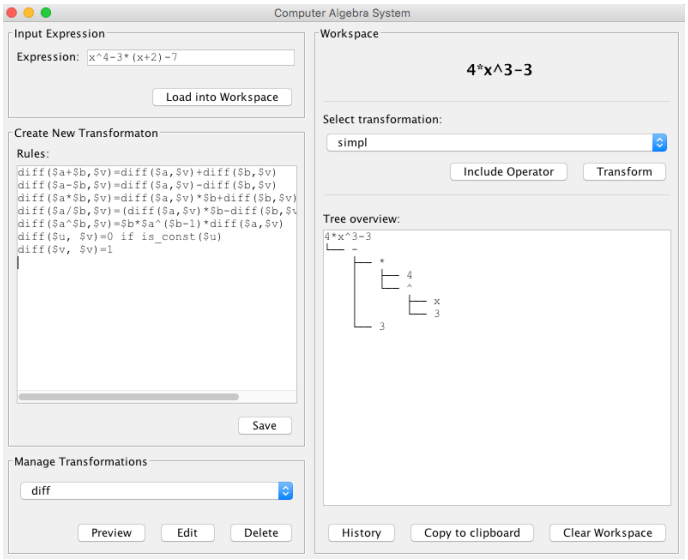

Fig. 3.10 First GUI instance.

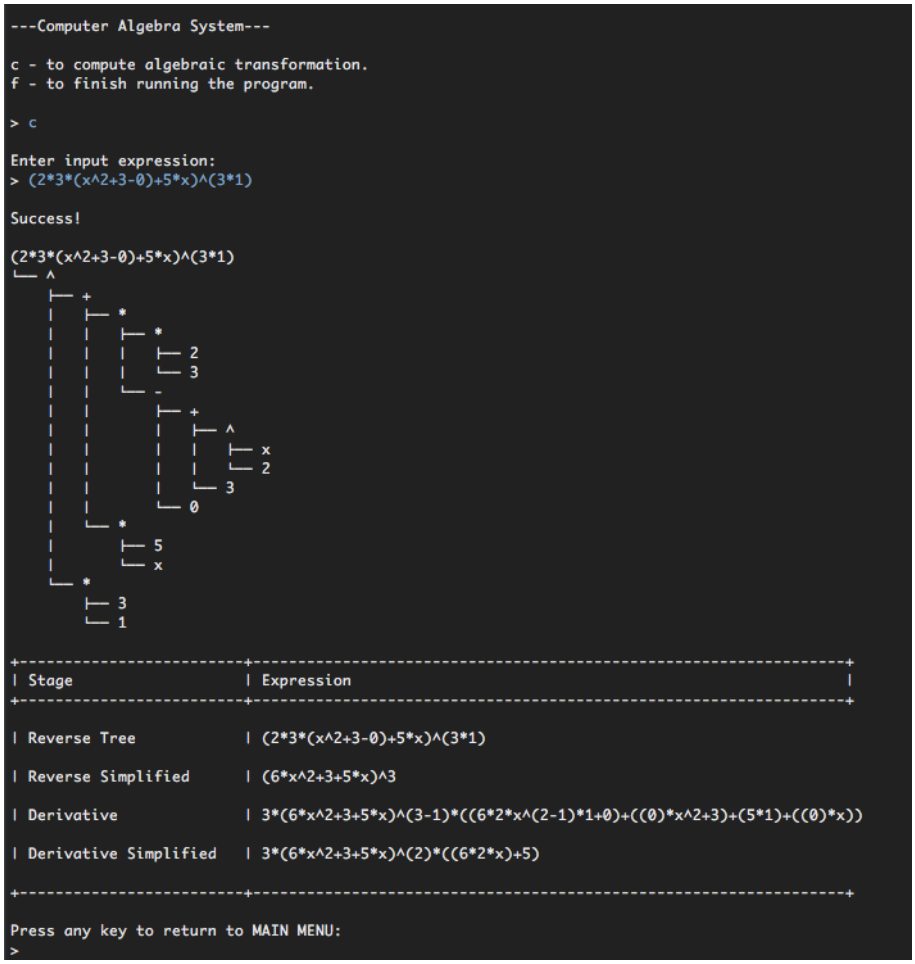

Fig. 3.11 Terminal interface prototype.

3.2 Implementation

The system was implemented in Java using Eclipse IDE. The project contains the following four packages:

- **common.parser:** Includes the grammar and the Java classes generated by ANTLR4 (parser and lexer).
- **common.system:** Includes the back-end implementation of the system. Contains tree-node classes, rewriting system and auxiliary routines.
- **common.views:** Encloses all the Swing visual Java classes that shape the GUI of the application.
- **common.testing:** Contains all the unit-testing classes

3.2.1 The Syntax Tree Structure

The syntax tree data structure is composed of the following five classes, all enclosed in the common.system package:

- AbstractNode.java
- NumNode.java
- VarNode.java
- DollarNode.java
- OperatorNode.java

The member fields and class functions are described in detail below.

Abstract Node

Abstract class is a class that defines any node object. Every node of a tree is an AbstractNode. There are no member fields.

The class functions are listed and briefly explained below:

1. **clone**: No parameters to accept, returns equivalent AbstractNode object with different reference identity. It is implemented from super class to support immutability within the system.
2. **toString**: Receives no parameters and returns the infix and user-friendly expression string of the node. It is implemented for calling and returning the expression (AbstractNode node). The auxiliary method is included in the NodeFunctions class, which is explained in detail in the following subsection).

This class includes no constructor. However, it is highly utilised within the system to specify the handling of any possible node type.

Number Node

NumNode class defines a node which contains a numerical value. It extends AbstractNode without overriding any of the super-class methods.

Member fields:

- **Key**: Integer type field containing the arithmetic value of the number.

Class functions:

1. **Constructor**: Receives only one integer parameter that gets stored within the key field.
2. **Setter** function for key field.
3. **Getter** function for key field.

Variable Node

VarNode class defines a node which contains a symbolic variable. It extends AbstractNode without overriding any of the super-class methods. It implements a comparable interface.

Member fields:

- **Key**: String type field containing the alphabetical name of the variable.

Class functions:

1. **Constructor**: Receives only one string parameter that sets the key member field.
2. **Setter** function for key field.
3. **Getter** function for key field.
4. **compareTo**: Receives a VarNode object as a parameter. Compares the key strings between the two objects and returns an integer value. This method is utilised in sorting within auxiliary functions.

Dollar Node

The DollarNode class defines an arbitrary node which could represent anything, ranging from a single number to a long complicated expression. It extends AbstractNode without overriding any of the super-class methods and implements Comparable interface.

Member fields:

- **Key**: String type field containing the alphabetical name that follows the dollar sign. The key string does not include the dollar sign prefix. Dollar sign is added only in auxiliary printing methods (such us expression() in NodeFunctions class). It extends AbstractNode without overriding any of the super-class methods, also implementing Comparable interface.

Class functions:

1. **Constructor**: Receives only one string parameter that sets the key member field.
2. **Setter** function for key field.
3. **Getter** function for key field.
4. **hashCode and equals**: Both methods overridden, to achieve efficient hashing. Equals method returns true only if the two objects considered are of DollarNode type, and have matching key fields. For example, two DollarNode objects with a key matching to "a", will return a true equals value, despite having different reference id.
5. **compareTo**: Receives a DollarNode object as a parameter. Behaves similarly to VarNode class.

DollarNode nodes are extensively used in rewriting rules. They are nodes that can be replaced by any kind of expression.

Operator Node

Operator Node class defines nodes that behave like operators. It is the only type of node that has references to other nodes (children). It extends AbstractNode class, overriding the clone function.

Member fields:

- **Key**: String type field containing the symbol or name of operator. For example "+" or "diff".
- **Children**: ArrayList of AbstractNodes. An ArrayList was chosen as a data structure for the children field in order to provide complete flexibility in terms of the number of children an operator could have. A division operator has 2 children, while there could be custom transformation operators that may have 5 or more children.

Class functions:

1. **Constructor**: Receives only one string parameter that sets the key member field.
2. **Setter** function for key and children fields.
3. **Getter** function for key and children fields.
4. **Clone**: Clone method is overridden from AbstractNode super-class. Apart from cloning the OperatorNode object itself, the AbstractNode children objects within the ArrayList are also cloned.

3.2.2 Rule object

The rule object is defined by the rule class. As previously mentioned, a rewriting rule is an equation of two expressions followed by an optional boolean condition.

A typical rewriting rule is of the following format:

```
diff($u,$v)=0 if is_const($u)
```

where:
`diff($u,$v)` is the left hand side expression,
0 is the right hand side expression,
and `is_const($u)` is the boolean condition.

Hence the rule object includes three member fields as shown in the figure below:

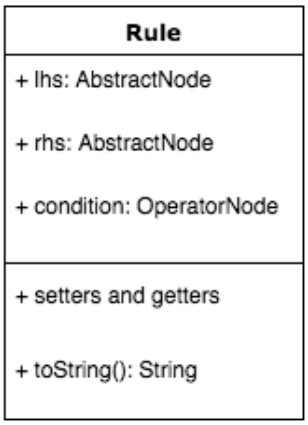

Fig. 3.12 Rule object class. Member fields and methods

Class also includes a constructor that gets parametrised with a left-hand-side AbstractNode, a right-hand-side AbstractNode and an OperatorNode condition.

Note that conditions are represented and evaluated as OperatorNodes. For example, the above `is_const($u)` condition is represented as the left tree of figure 3.13. The right tree of Figure 3.13 displays the `($u==$v) AND ($k==3)` condition.

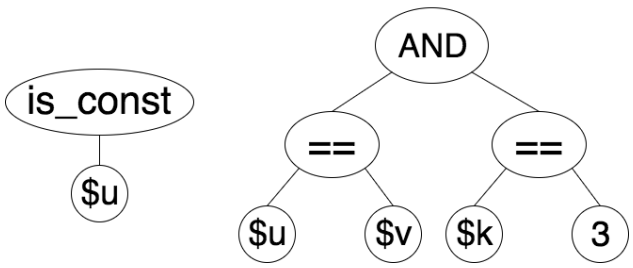

Fig. 3.13 Boolean conditions tree representation

Bellow is the table of the boolean conditions that this CAS supports:

Table 3.2 Supported boolean conditions

Boolean Conditions of CAS

Name of operator	Children	Description
AND	2	Logical AND
OR	2	Logical OR
NOT	1	Logical NOT
==	2	Numerical equality
is_const	1	Parameter is instance of NumNode
depends	2	First parameter includes the second one.

All these conditions can be evaluated as true or false. There is a condition evaluating auxiliary method in the NodeFunctions class.

3.2.3 Parsing system

The version of ANTLR used in the context of this project was 4.7. Below is the grammar that generates the parser and lexer analyser.

```
grammar Expr;
prog: stat+ ;
stat: expr NEWLINE # printExpr
        | NEWLINE # blank;

expr: expr op='^' expr # Pow
        | expr op=('*'|'/') expr # MulDiv
        | expr op=('+'|'-') expr # AddSub
        | expr '=' expr ( ' if ' bexp)? # rule
        | '(' expr ')' # parens
        | '$' ID # dollar
        | '-' expr # unary
        | op=ID '(' expr ((',') expr)* ')' # RuleOpe
        | INT # int
        | ID # id;

bexp: bexp op=('AND'|'OR') bexp # andor
        | ID '(' bexp ')' # opcond
        | '(' expr '==' expr ')' # equality
        | 'depends(' expr ',' expr ')' # depends
```

```
        | 'is_const(' expr ')' # const;

ID   : [a-zA-Z_]+ ; // match identifiers
INT  : [0-9]+ ; // match integers
NEWLINE:'\r'? '\n' ; // return newlines to parser
WS   : [ \t]+ -> skip ;
```

The rules named with capitals define the lexer rules (ID, INT, NEWLINE WS), while prog, stat, expr and bexp specify the parser rules.

ANTLR's lexer tokenises the input using a top to bottom approach. Hence, tokens defined first within a rule have higher precedence. Expr and bexp are the two important parsing rules of this grammar. Expr determines an expression while bexp defines a boolean expression, commonly used in conditional statements. As observed in the expr rule, the power operator is specified prior to multiplication-division and therefore has higher precedence. Grammar rules are left-recursive, an attribute that is particularly evident in the expr rule.

An expression can range from a single number or variable to a lengthy and complex expression. This long expression can be decomposed into simpler expression components specified in other cases of the expr rule. Thus, a rule can be self-referential, allowing recursion, a vital feature for the needs of this CAS grammar.

The rule of bexp is used in the `#rule` case of expr, enclosed in an optional if-statement.

Using this grammar, ANTLR4 generates the six following files that are stored in the common.parser package:

- ExprBaseVisitor.java
- ExprLexer.java
- ExprParser.java
- ExprVisitor.java
- Expr.tokens
- ExprLexer.tokens

with Expr.g4 being the grammar file.

In order to walk a parse tree, ANTLR4 provides a visitor design pattern. The common.system package of this CAS includes a myVisitor.java class which extends the ExprBaseVisitor.java and implements the visitor interface. Visitor interface allows the control of the tree walk. Methods can be explicitly called to visit specific children. Within the

myVisitor class, methods for every rule case are implemented, returning AbstractNode objects.

3.2.4 Node Functions class

The NodeFunctions.java class of the common.system package contains all the auxiliary node methods. These are described below, along with their parameters and return output.

Expression

Parameters: AbstractNode node
Returns: String infix notation
This function converts any node tree structure to a string infix notation. It enables user-friendly representation of tree-expressions. It accounts for precedence, inserting parentheses where needed. It also implements the toString method of the AbstractNode class.

Parentheses allocation is determined by investigating the children of an operator node. If a child of an operator node is itself an operator node of lower precedence, then parentheses should surround the respective child. For example, if a "/" operator node has a "+" operator child, then the plus operator child should be printed within parentheses (addition is of lower precedence than division). This function is implemented using switch case statements, individually handling the precedence case of each operator with a recursive manner.

Tree

Parameters: AbstractNode node
Returns: String that graphically displays a tree structure
For example a node containing the $7+x^4+3*x-sin(x)$ expression will return the string displayed in figure 3.14. It is recursive as every other tree-parsing method of this CAS.

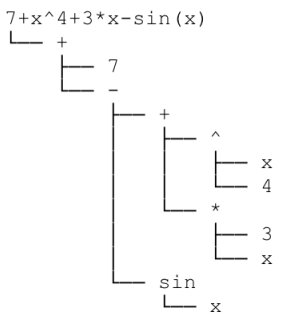

Fig. 3.14 String tree method

Eval

Parameters: AbstractNode node
Returns: Evaluated AbstractNode
Method that evaluates any node that can be evaluated within an abstract node tree. A node that can be evaluated is any operator node whose children are only number node (NumNode) instances. For example the OperatorNode 3*4 gets evaluated using eval() and returned as a NumNode of value 12. On the other hand, an OperatorNode 3*x, cannot be further evaluated and is returned as it is.

This eval function accounts for unary minus operator. It is recursive, and when given a whole tree that cannot be fully evaluated, it only evaluates the components of the tree that can be evaluated and returns the updated tree.

Match

Parameters:

- AbstractNode left-hand-side of a rewriting rule
- AbstractNode node to be matched

Returns: HashMap<DollarNode, AbstractNode> that maps every DollarNode object encountered in the matching LHS to the respective AbstractNode expression component. This method is vital for the rewriting system. It investigates whether there is a match between a left-hand-side node of a rewriting rule, and a given node. Rewriting rules may contain DollarNodes (nodes that could represent any expression). There are three possible return states:

- **Null**: Method returns a null value. This means that there is no match between the two parameter nodes.
- **Empty HashMap**: Method returns an instantiated but empty HashMap object. This means that there is a match. However, the left-hand-side of the rule did not contain any DollarNodes and thus there is no need for mapping.
- **Non-empty HashMap**: Method returns a non-empty HashMap. There is a match. Left-hand-side contains DollarNodes. The HashMap maps these DollarNodes to the respective matching children of the parameter node. This HashMap will be later utilised in rewriting.

Below are some examples displaying the functionality of the method:

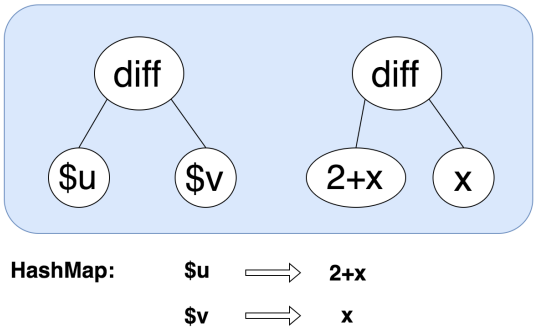

Fig. 3.15 The two nodes match. Non-empty HashMap returned.

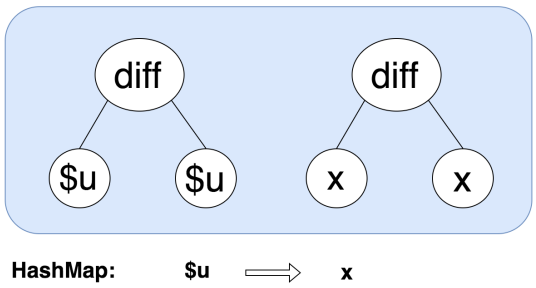

Fig. 3.16 The two node match. Note that there is only one DollarNode contained in the HashMap set of keys

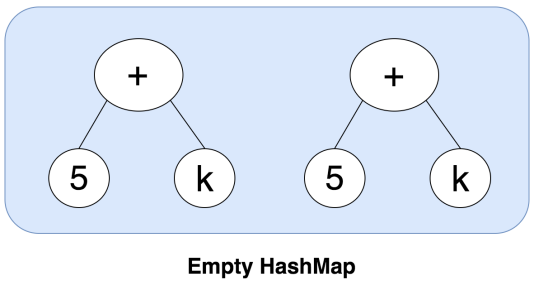

Fig. 3.17 There is a match. Empty HashMap returned as there are no DollarNodes in the LHS node.

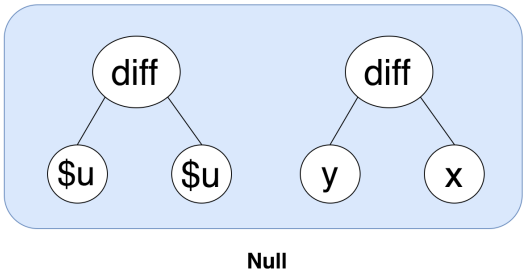

Fig. 3.18 There is no match. In the left branch $u matches with y and in the right $u matches with x. This cannot be true thus a null is returned.

The match method is recursive and implemented to traverse whole tree structures. A match between two operator nodes also indicates a match between their corresponding children. Hence it determines a complete match.

Rewrite

Parameters:

- AbstractNode right-hand-side of the matched rewriting rule
- Matching HashMap returned by the match method.

Returns: AbstractNode object. Returns the rewritten right-hand-side node. The DollarNodes of the given right-hand-side node are used as keys for the HashMap parameter. These DollarNode keys are then replaced by their respective mapping AbstractNode values. The overridden methods of hashCode and equals within the DollarNode class provide efficient and accurate hashing. It is a recursive function vital for the custom transformation feature.

Evaluate Condition

Parameters: OperatorNode condition. The node of a rewriting rule containing the boolean condition
Returns: True or False. Evaluates the condition and returns a boolean result.
The boolean conditions that this CAS supports are stated and explained in table 3.1. This function handles every specified boolean condition, using a switch-case statement. For each boolean condition case there is a standard procedure to follow in order to evaluate it. For example, in the case of `depends($u,$v)` condition, the evaluating method traverses the whole tree of `$u` and tries to find a node that matches with `$v`. Even if a single match is found, a true value is returned. Else the boolean condition is false. Again this auxiliary function is recursive, and important within the rewriting system.

In the cases of AND, OR logical operators, short-circuit evaluation is implemented, in order to avoid redundant exceptions and improve performance.

Short-circuit evaluation:

- **AND logical operator**: In order for an AND logical operator to return a true value, all of its children must be true. Even if only one out of many children is false, then the whole operator returns a false value. Short-circuit evaluation of logical AND involves the evaluation of each children step by step, starting from the first one (leftmost) to the last one (rightmost). In case of a children evaluating as false, the condition evaluating method of the AND operator returns an overall false, without further

examining the rest of the children. Assume the following condition expression is to be evaluated:
`NOT(x==0) AND (6/x==2)` with x assigned a value of 0.

Without short circuit evaluation, both `NOT(x==0)` and `(6/x==2)` boolean conditions will be evaluated. With `x=0`, the execution of `(6/x==2)`, would cause a divide by zero exception. However using short-circuit, the system first evaluates the left `NOT(x==0)` condition, and only if true, proceeds to evaluating the next one. As `NOT(x==0)` with `x=0` is false, the system will not attempt to evaluate the `(6/x==2)` condition, and will terminate the method by returning an overall false value. Therefore, the exception will be avoided.

- **OR logical operator**: The short-circuit evaluation of the OR operator node is implemented in a similar manner. However, in this case the true/false short-circuit behaviour is reversed. When evaluating an OR operator condition, the evaluating method will execute the next children only if the previous ones were false. In case of a true child encountered, the evaluating condition method is terminated, returning a true value.

Canonical Form

Parameters: AbstractNode node
Returns: Returns the input node arranged in a system-specified canonical order.
Function is recursive. If an OperatorNode exists within the given AbstractNode parameter, its children will be ordered using the following rules. The canonical order is decided:

- By type of node. Priority from higher to lower: NumNode, VarNode, DollarNode, OperatorNode.
- Lexicographically. Ordering children-nodes of the same type is implemented using lexicographical sorting. VarNodes and DollarNode use alphabetical sorting.

The example in figure 3.19 is simple. However, when inserting a large tree structure, this process is performed on every operator node of the tree, ensuring that the whole tree is returned in a canonical order. Computer systems cannot associate a+b with b+a, considering them different-unequal expressions. Canonical method resolves this issue.

Parsing auxiliary functions

Parameters: Infix notation string
Returns: AbstractNode tree data structure There are two parsing auxilliary methods:

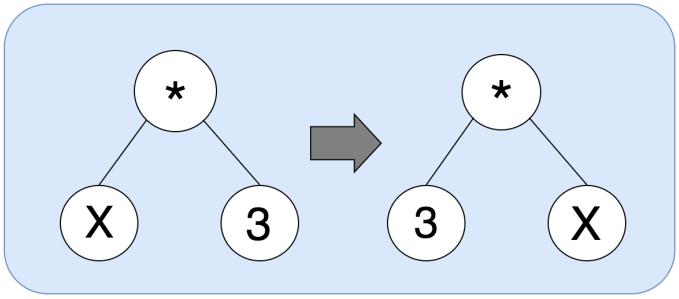

Fig. 3.19 Canonical method converts $x * 3$ to $3 * x$.

- Expression to Node: Single expression parsing.
- Set of rules to Transformation: Multiple line parsing.

Within these methods, the ANTLR4 generated parsers are called, and combined with the myVisitor class, trees are parsed from the text input string.

3.2.5 Rewriting system

The rewriting system of this CAS is enclosed in the Transformation.java class of the common.system packet.

This class specifies and constructs transformation objects. These are objects that essentially contain a set (ArrayList) of rewriting rules. The Transformation class has only 1 member field and several class functions as presented in the class diagram below.

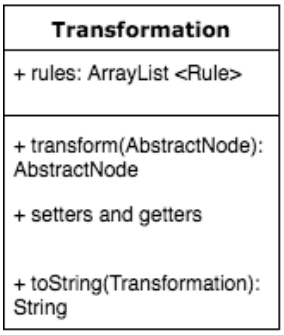

Fig. 3.20 Transformation class diagram.

The rewriting process is carried out by the transform method. It receives an AbstractNode as a parameter, and returns it transformed.

The rewriting process involves the following sequence of steps:

- **Step 1**: The parameter AbstractNode is initially evaluated using the eval() method, and then canonically ordered using the canonical() auxiliary method.

- **Step 2**: Iterate over the array-list of rules contained in the Transformation object. For each rule, check if there is a match between the left-hand-side of the rule and the given AbstractNode parameter. This is achieved using the match() auxiliary function. In case of a match, a matching HashMap is returned.
- **Step 3**: If there is a match, the boolean condition of the matched rule is evaluated. This is achieved using the evalCondition() auxiliary method. If condition is true, then we can proceed to the rewriting step.
- **Step 4**: If a left-hand-side match is successful and the boolean condition is met, the right-hand-side of the matched ruled is rewritten with regards to the matching HashMap. This is achieved using the rewrite() auxiliary method.
- **Step 5**: We iterate over the array-list rules again, but now in order to find a match for the rewritten right-hand-side node resulted from Step 3. We repeat the same procedure until a further match cannot be found.
- **Step 6**: Utilising the recursive manner of this transform method, each children of the resulting RHS rewritten node is being investigated for ay transformation opportunities. In case of a child matching a rule, rewriting occurs using the same procedure. This ensures that the whole expression tree is completely rewritten/transformed. Process ends when no further matches can be detected, and the initial and now transformed AbstractNode parameter is returned.

If no match occurs, the method returns the initial AbstractNode parameter.

3.2.6 GUI

The GUI of this CAS is composed of 4 Swing Java classed enclosed within the common.views package. The GUI inherits a system look-and-feel, adapting to the design principles of different operating system platforms.

- MainGUI.java
- Trans.java
- Tree.java
- About.java

Main GUI

This class implements the main panel of the GUI. It extends JFrame.

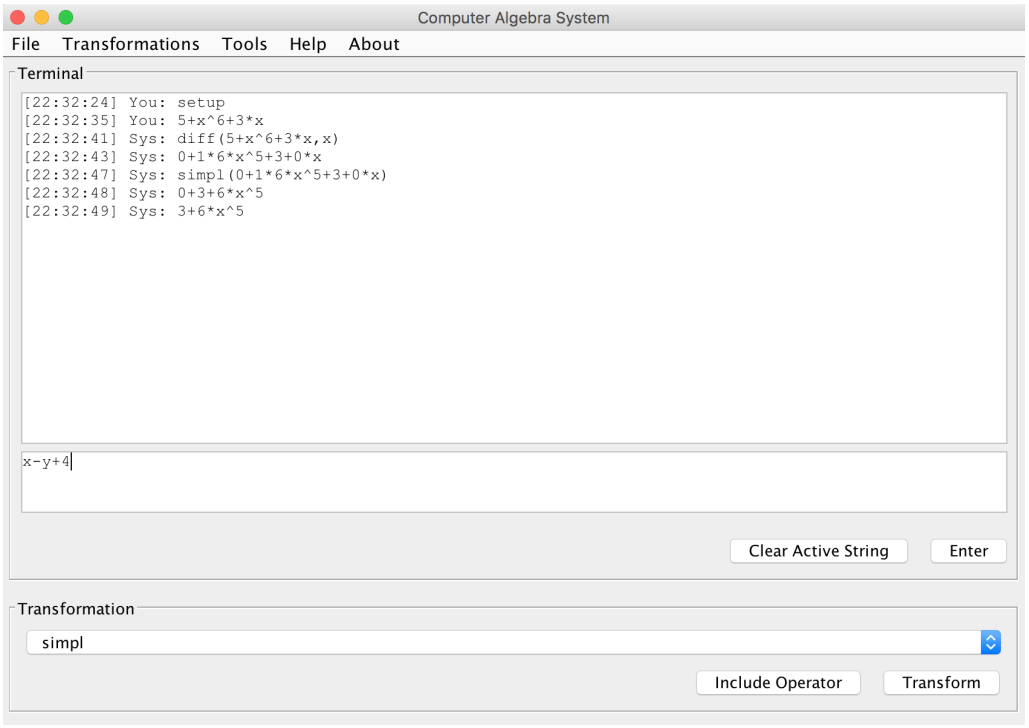

Fig. 3.21 Main panel of GUI.

The main panel consists of 3 main components: top-menu, terminal, transformation panel.

The **top menu**, is a horizontal drop down menu that allows access to all the features of the system. These are specified in section 3.1.3.

The **terminal**, is implemented using two large and scrollable text area fields allowing large expression inputs and outputs. The upper one is intended for history and is not editable, while the lower one acts as an active and editable string input box.

MainGUI class contains among others two important member fields:

- Current expression: Named as workspace within the code. Contains the AbstractNode object that the system currently handles. It is the last node object returned by the user or the system.

- History: Stack of AbstractNodes. Each time the current expression is updated, a copy of it is stored within a history stack. This stack maintains an object history of the terminal, rather than a text history such the one included in history text area.

The history stack is particularly designed for the implementation of keyboard responsiveness within the active string text area. Similarly to most common terminal environments, the active string of this terminal updates to previously registered commands/expressions on pressing the up arrow key. An event listener focusing on the pressing

of up arrow key is utilised within the class. Additionally, a command/expression can be registered directly by pressing the enter-return key.

Furthermore, right-click functionality was included in the active string text area.

Fig. 3.22 Right-click menu implemented for active string text area.

The two features above provide a fully functioning active string text area for the terminal of this CAS.

Terminal also provides time-stamp and agent (You or Sys) of every expression/command registered.

The history of the terminal can be exported as a .txt file stored within the folder of the running jar application. The name of the txt file includes a date and time stamp.

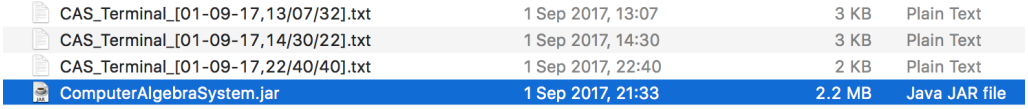

Fig. 3.23 Export terminal history as txt file.

An import facility is also implemented to load any previously saved terminal history. After using a dialog box to select the appropriate txt file, the history text area of the terminal gets updated to match the imported history.

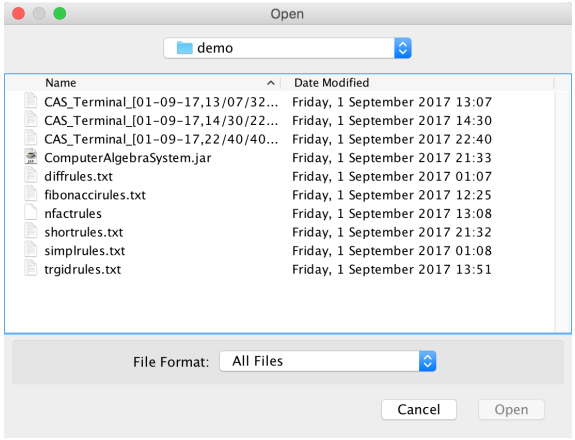

Fig. 3.24 Import a history txt file, and load it onto the terminal.

The terminal system also supports a group pre-set commands, that provide greater usability and accessibility. The horizontal top-menu is not the only tool allowing access to

every feature of this CAS. The user can access all the features of the system by registering pre-set commands in the terminal environment. Terminal pre-set commands explained in the Figure 3.25 below:

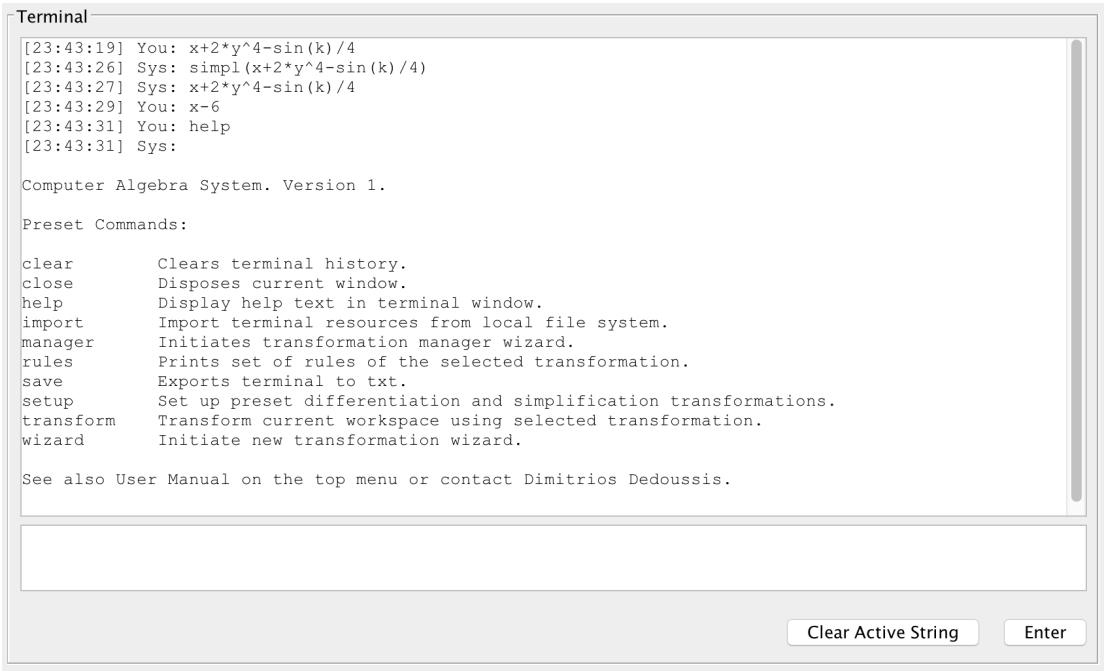

Fig. 3.25 System pre-set commands presented by registering the help command.

The **transformation panel** of the main GUI is implemented using a combo box. The combo box stores all the transformation objects constructed within this session.

GUI also includes:

- New transformation wizard
- Manage transformation facility
- Tree display facility
- About

Figures explaining the functionality of these frames are presented in the Appendix.

Chapter 4

Results and evaluation

4.1 Transformations

The system was tested and evaluated on the following custom transformations. All the transformation rewriting rules presented in this chapter are implemented using the user-programming language of this CAS.

4.1.1 Differentiation

Rules:

```
diff($a+$b,$v)=diff($a,$v)+diff($b,$v)
diff($a-$b,$v)=diff($a,$v)-diff($b,$v)
diff($a*$b,$v)=diff($a,$v)*$b+diff($b,$v)*$a
diff($a/$b,$v)=(diff($a,$v)*$b-diff($b,$v)*$a)/$b^2
diff($a^$b,$v)=$b*$a^($b-1)*diff($a,$v)
diff(sin($v),$v)=cos($v)
diff(cos($v),$v)=0-sin($v)
diff(tan($v),$v)=1+tan($v)^2
diff($u,$v)=0 if is_const($u) OR NOT(depends($u,$v))
diff($v,$v)=1
diff(-$v,$v)=-1
ndiff($e,1)=diff($e,x)
ndiff($e,$n)=ndiff(diff($e,x),$n-1)
```

Below are examples of differentiating a symbolic expression in terms of x. Output listed is generated and exported from the CAS terminal:

```
[17:28:18] You: 5+2*x^4+6/x
[17:28:24] Sys: diff(5+2*x^4+6/x,x)
[17:28:27] Sys: 0+0*x^4+2*1*4*x^3+(0*x-6)/x^2
[17:28:29] Sys: 0+2*4*x^3+(-6)/x^2
[17:28:31] Sys: 2*4*x^3+(-6)/x^2
```

```
[17:33:02] You: setup
[17:33:19] You: sin(x)^3/x
[17:33:24] Sys: diff(sin(x)^3/x,x)
[17:33:26] Sys: (x*3*sin(x)^2*cos(x)-1*sin(x)^3)/x^2
[17:33:37] Sys: (x*3*sin(x)^2*cos(x)-sin(x)^3)/x^2
[17:33:43] Sys: (x*3*sin(x)^2*cos(x)-sin(x)^3)/x^2
```

```
[17:42:05] You: 5+3*x-y
[17:42:12] Sys: diff(5+3*x-y,x)
[17:42:13] Sys: 0+3+0*x-0
[17:42:20] Sys: 3
```

The ndiff operator defined in the rules, differentiates the given expression over a specified n number of times. Testing the `ndiff` operator using the trigonometric `tan(x)` function:

For `n=3`:

```
You: tan(x)
Sys: ndiff(tan(x),3)
Sys: ndiff(1+tan(x)^2,2)
Sys: ndiff(0+2*tan(x)^1*(1+tan(x)^2),1)
Sys: ndiff(2*tan(x)*(1+tan(x)^2),1)
Sys: diff(2*tan(x)*(1+tan(x)^2),x)
Sys: (0*tan(x)+2*(1+tan(x)^2))*(1+tan(x)^2)+...
         ...(0+2*tan(x)^1*(1+tan(x)^2))*2*tan(x)
Sys: (0+2*(1+tan(x)^2))*(1+tan(x)^2)+...
         ...2*tan(x)*(1+tan(x)^2)*2*tan(x)
Sys: 2*(1+tan(x)^2)*(1+tan(x)^2)+...
         ...2*tan(x)*(1+tan(x)^2)*2*tan(x)
```

As the differentiation rule of tan(x) produces expanding results, consecutive differentiations require high memory and processing resources. This CAS has tested to successfully compute the ndiff transformation of tan(x), with `n=10`.

4.1.2 Simplification

Rules:

```
simpl($s)=$s if is_const($s)
$s+0=$s
0+$s=$s
$s-0=$s
0-$s=-$s
$s*0=0
0*$s=0
$s*1=$s
1*$s=$s
0/$s=0
$s^0=1
$s^1=$s
0^$s=0
$s-$s=0
$s/$s=1
$s/1=$s
simpl($s+$k)=$s+$k
simpl($s-$k)=$s-$k
simpl($s*$k)=$s*$k
simpl($s/$k)=$s/$k
simpl($s^$k)=$s^$k
```

Simplification examples exported by the CAS terminal:

```
[17:58:01] You: 6+y*x^0+k/1
[17:58:05] Sys: simpl(6+y*x^0+k/1)
[17:58:06] Sys: 6+1*y+k/1
[17:59:07] Sys: k+6+y
```

```
[18:01:23] You: 4+3-5*(x/x)+0/y-0
[18:01:26] Sys: simpl(4+3-5*(x/x)+0/y-0)
[18:01:27] Sys: 2
```

4.1.3 Fibonacci

Rules:

```
f(0)=0
f(1)=1
f($a)=f($a-1)+f($a-2)
```

Examples:

```
[18:03:27] You: 1
[18:03:32] Sys: f(1)
[18:03:33] Sys: 1

[18:03:38] You: 4
[18:03:40] Sys: f(4)
[18:03:41] Sys: 3

[18:04:03] You: 8
[18:04:05] Sys: f(8)
[18:04:05] Sys: 21

[18:03:48] You: 15
[18:03:50] Sys: f(15)
[18:03:53] Sys: 610
```

Fibonacci is a recursive transformation. If the input parameter number increases (especially above 25), the system exhibits high resource demands and needs more time to compute the output, similarly with `ndiff`. The system has been tested to successfully compute f(30).

4.1.4 Factorisation

This custom transformation factorises the sum of two exponents, which are polynomials in the form of $x^n + x^k$, with n, k being constant integers.

Rules:

```
nfact($x^$b+$x^$c,$n)=nfact($x^0*($x^$b+$x^$c),$n)  ...
        ... if is_const($n)
nfact($x^$k*($x^$b+$x^$c),1)=$x^($k+1)*($x^($b-1)+$x^($c-1))
nfact($x^$k*($x^$b+$x^$c),$n)=nfact($x^($k+1)*($x^($b-1)+...
        ...$x^($c-1)),$n-1) if is_const($n)
```

Examples:

```
[18:14:22] You: x^8+x^4
[18:14:27] Sys: nfact(x^8+x^4,4)
[18:14:28] Sys: nfact(x^0*(x^8+x^4),4)
[18:14:29] Sys: nfact(x^1*(x^7+x^3),3)
[18:14:31] Sys: nfact(x^2*(x^6+x^2),2)
[18:14:31] Sys: nfact(x^3*(x^5+x^1),1)
[18:14:32] Sys: x^4*(x^4+x^0)
[18:14:37] Sys: x^4*(1+x^4)

[18:23:23] You: y^2+y^9
[18:23:32] Sys: nfact(y^2+y^9,2)
[18:23:34] Sys: nfact(y^0*(y^2+y^9),2)
[18:23:36] Sys: nfact(y^1*(y^1+y^8),1)
[18:23:36] Sys: y^2*(y^0+y^7)
[18:23:40] Sys: y^2*(1+y^7)
```

4.1.5 Trigonometric identities

This custom transformation example implements several trigonometric identities.

Rules:

```
trgid(sin($v)^2+cos($v)^2)=1
trgid(sin(-$v))=-sin($v)
trgid(cos(-$v))=cos($v)
trgid(tan($v))=sin($v)/cos($v)
trgid(sin($v)/cos($v))=tan($v)
```

Examples:

```
[18:33:20] You: sin(x)^2+cos(u)^2
[18:33:22] Sys: trgid(sin(x)^2+cos(u)^2)
[18:33:24] Sys: sin(x)^2+cos(u)^2

[18:33:33] You: sin(x)^2+cos(x)^2
[18:33:34] Sys: trgid(sin(x)^2+cos(x)^2)
[18:33:35] Sys: 1

[18:36:33] You: sin(-x)
```

```
[18:36:36] Sys: trgid(sin(-x))
[18:36:37] Sys: -sin(x)

[18:37:03] You: sin(x)/cos(x)
[18:37:04] Sys: trgid(sin(x)/cos(x))
[18:37:05] Sys: tan(x)
```

4.1.6 Short-circuit testing

This short-circuit testing example is a custom transformation that confirms the correct implementation of short-circuit evaluation within the boolean conditions.

Rules:

```
short($x,$y)=1 if NOT(($y==0)) AND ($x/$y==2)
short($x,$y)=2
```

Examples:

Case 1:

```
[18:43:12] You: 6
[18:43:19] Sys: short(6,3)
[18:43:24] Sys: 1
```

Case 2:

```
[18:44:02] You: 4
[18:44:05] Sys: short(4,0)
[18:44:06] Sys: 2
```

```
[18:44:16] You: 4
[18:44:18] Sys: short(4,1)
[18:44:19] Sys: 2
```

From the above results, it is evident that in Case 1, where x=6, y=3 and thus x/y=2, the output is 1. In Case 2, where y=0, there is no divide-by-zero exception caused. The second boolean condition of the AND operator was never evaluated, as the first one was false. The rewriting proceeded with matching the next rule, and returning a result of 2.

4.2 Evaluation

The system is successful in rewriting input symbolic expressions by applying a set of specified transformation rules. All of the transformations tested in the above sections return correct and accurate results. For each of the transformation tested, boundary and limiting values were used, in order to trick the system and detect possible defects.

The system was also examined within resource demanding circumstances, where matching and rewriting was to be performed on lengthy and complex expressions. Again, the results were particularly positive. The system manages to successfully compute up to the 10th derivative of tan(x). Early in the process of setting objectives, a number of 6 consecutive differentiations of the tan(x) expression was considered a desired result. Hence the achievement of computing 10^{th} derivative is beyond the original expectation and aim set for this CAS. Moreover, the Fibonacci computation of 30 was also an important success in terms of performance, as the Fibonacci transformation, being recursive, requires substantial system resources.

The CAS was also successful in terms of storing and representing symbolic expressions. The tree data structure designed proved to be very efficient in handling such mathematical objects. The parsing component of the system is precise in terms of interpreting human input of various forms, lengths and complexities. The CAS interacts with the user in an effective manner, through its well deigned GUI terminal, which took a few design iterations to reach its final shape. Expression printing maintains precedence, and generates precise output.

Finally, in terms of exceptions, the system examines them and does not allow null field registration or mathematical exceptions such as divide-by-zero. Tips and help messages prevent the user from creating exceptions.

Overall, the CAS designed within the context of this dissertation produces correct and precise results, meeting (or in certain aspects exceeding) the requirements set at the beginning of the project. Implementation of symbolic custom transformations through rewriting is successfully achieved, and this paves the way for utilizing this tool in a variety of applications.

Chapter 5

Conclusions

5.1 Meeting objectives

The Computer Algebra System developed within the scope of this dissertation successfully meets the objectives that were defined during the requirements and planning stages of the project. The outcome is a system that understands and manipulates symbolic expressions.

Interpreting infix mathematical input was a demanding task that was tackled by designing and implementing the appropriate tree data structure. Furthermore, the correct parameterisation of ANTLR4 generated a parser that allows the system to accurately understand algebraic expressions.

Manipulation involves not only the evaluation but also the transformation of the mathematical object. The rewriting system of the CAS was designed and implemented successfully, enabling the custom transformation feature. By supporting custom transformations, the system becomes dynamic, significantly extending its flexibility and overall potential.

In terms of user-interaction, the GUI application offers a friendly and effective experience and was a product of a few design iterations. The terminal component proved to be highly usable, and particularly efficient for the needs of the project. The parsing system successively implements a simple and accessible user-programming language. The interface is also very accurate in converting and presenting tree expressions into human-readable output.

5.2 Challenges faced

The materialisation of a software package such as this Computer Algebra System was an involved and non-trivial process, as it was designed and implemented entirely from scratch. Looking back at the project, there are a few challenges that were faced.

As this system is closely related to mathematics and computer algebra, its results should be analytically accurate. Therefore, every aspect of the back-end system had to be meticulously designed and thoroughly tested in order to achieve a successful final version. The bug and mistake intolerance that this project exhibited was a constraining factor. On a positive note however, this provided an opportunity for continuous testing after every step in the implementation, and thereby allowed errors to be spotted fairly quickly.

Furthermore, some of the concepts involved in this project, such as the rewriting methodology, are rather complex. They therefore required a considerable amount of research, and multiple prototype iterations. Eventually though, this challenge was overcome, leading to the implementation of a successful rewriting system within this CAS.

5.3 Future recommendations

The project undertaken was successful in implemented a user-friendly and accurate CAS. There are a few limitations to the current prototype, which can provide room for future improvement. Such recommendations are the following:

- Adding support of floating point values. The current prototype supports integers only.
- Enabling a variety of transformation strategies. Currently transformations are performed using a top-to-bottom approach. Even though this is efficient for a lot of transormations, there are transformations for which a botton-to-top approach would be more efficient. The implementation of a system that dynamically adapts the strategy based on the nature of transformation, would enhance the current capabilities of the CAS.
- Further improving the GUI, so that the system can provide syntax feedback for the user-programming language. For example, when entering inaccurate or invalid transformation rules, the user could be accordingly prompted.

References

[1] M. J. G. Veltman and D. N. Williams, "a Program for Symbol Handling," 1991.

[2] S. Wolfram, "Symbolic Mathematical Computation," *Communications of the ACM*, vol. 28, no. 4, pp. 390–394, 1985.

[3] A. Visser, "On the ambiguation of Polish notation," *Theoretical Computer Science*, vol. 412, pp. 3404–3411, 2011.

[4] M. Matooane, "Parallel systems in symbolic and algebraic computation," Tech. Rep. 5, 2002.

[5] B. Buchberger, G. E. G. E. Collins, and R. R. Loos, *Computer algebra : symbolic and algebraic computation*. Springer-Verlag, 1982.

[6] R. Piskac, "First-Order Logic - Syntax, Semantics, Resolution,"

[7] D. Richardson, "Some undecidable problems involving elementary functions of a real variable," *The Journal of Symbolic Logic*, vol. 33, no. 04, pp. 514–520, 1969.

[8] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. 2010.

[9] S. Autexier, E. Calculemus (Symposium) (15th : 2008 : Birmingham, and E. MKM (Conference) (7th : 2008 : Birmingham, "Intelligent Computer Mathematics, 9th International Conference, AISC 2008, 15th Symposium, Calculemus 2008, 7th International Conference, MKM 2008, Birmingham, UK, July 28 - August 1, 2008. Proceedings," in *AISC/MKM/Calculemus*, vol. 5144, p. 600, Springer, 2008.

[10] L. Bernardin, "How Maple Compares to Mathematica," p. 22, 2014.

[11] Wolfram.com, "Symbolic Computation—Wolfram Language Documentation."

[12] J. Hsiang and P. Lescanne, "THE TERM REWRITING APPROACH TO AUTOMATED THEOREM PROVING," no. June 1991, pp. 71–99, 1992.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, "Maude: Specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002.

Appendix A

Appendix

New transformation wizard

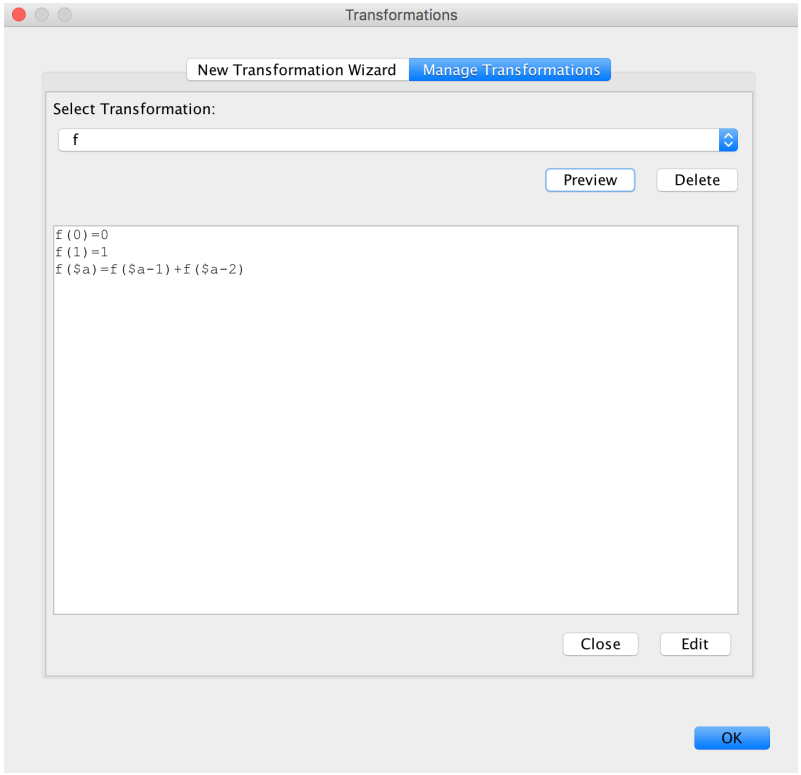

Fig. A.1 New transformation wizard.

Manage transformation facility

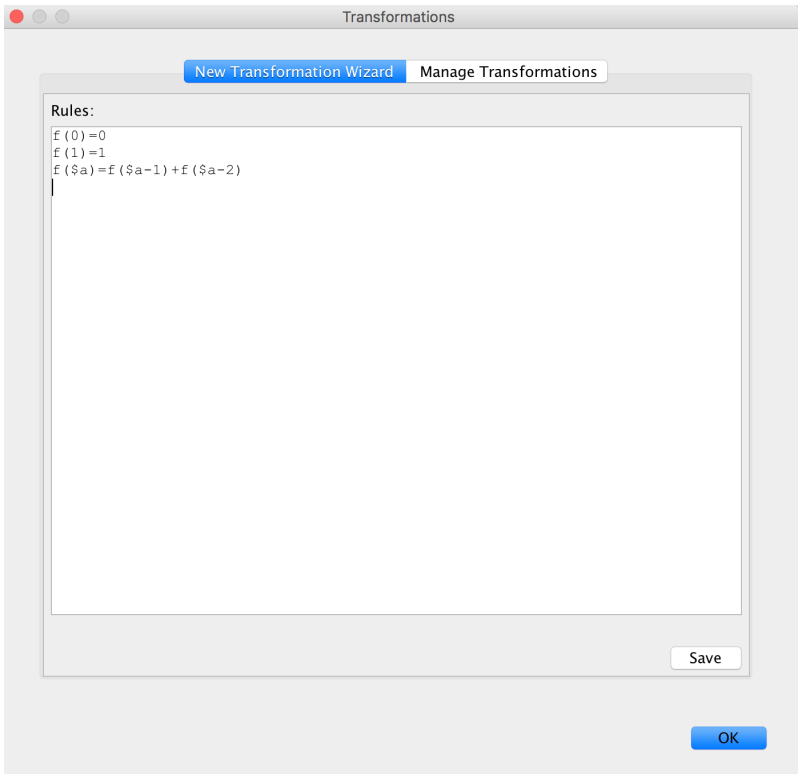

Fig. A.2 Manage transformation facility.

Tree display facility

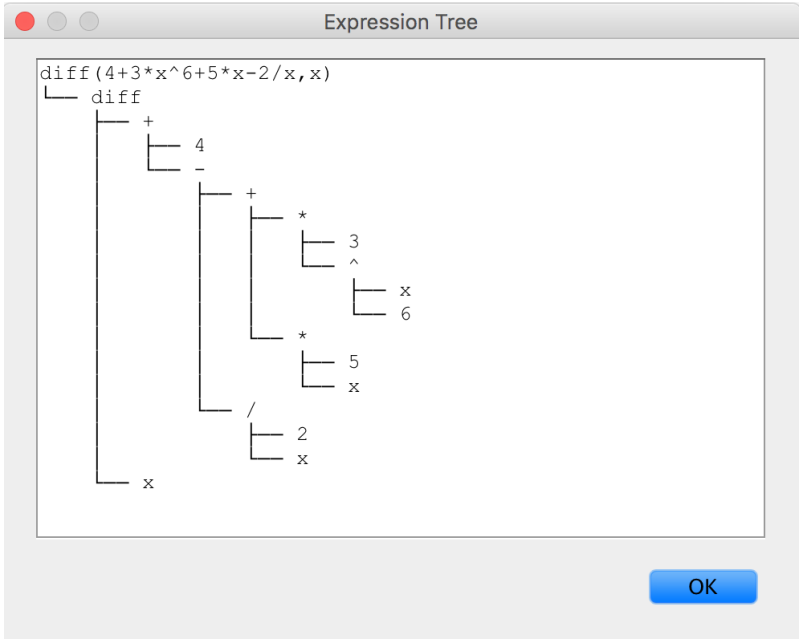

Fig. A.3 Tree display facility.

About panel

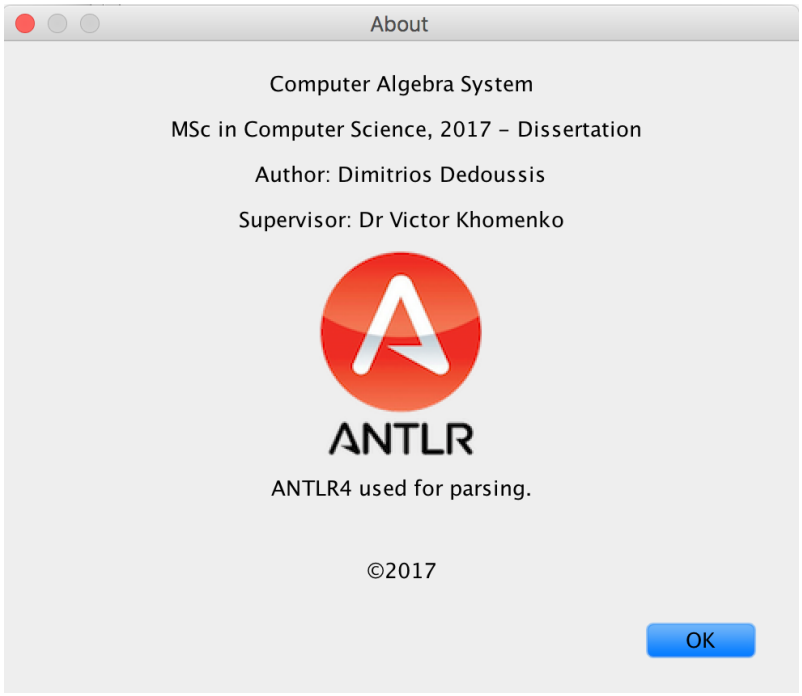

Fig. A.4 About panel.