Dissertation

An intuitive and adaptive approach to Term Rewriting

By: Xiu Hong Kooi
BSc Computer Science

School of Computing
Supervisor: Dr Jason Steggles

Abstract

Term rewriting is a simple concept used to perform symbolic computation, theorem proving and more. Despite its simplicity, systems that implement term rewriting are often complicated or restrictive and does not allow novice users to explore the concepts of term rewriting. This dissertation documents the process of developing a tool that solves the problem through the use of algorithms and software design techniques, the end product is an intuitive and flexible term rewriting system which inexperienced users can use to gain an understanding of the concepts behind term rewriting.

Declaration

"I declare that this dissertation represents my own work, except where otherwise stated."

Acknowledgements

I would like to thank my supervisor, Dr Jason Steggles for his guidance throughout the project as well as introducing me to a new field in computer science.

My family and friends for their continous support throughout University.

Users of the system for their valuable time and feedback.

Contents

1 Introduction

1.1 Rewrite Systems

In computing, *rewriting* [35] is the act transforming objects such as strings, graphs, mathematical equations, etc from one form into another. A *rewrite system* [36] is a system that performs rewriting on a set of objects following a set of criterias on how to manipulate them, these criterias are called the *rewrite rules* [30].

There are many different types of rewrite systems ranging from *strings rewriting systems* [45], *graphs rewriting systems* [3] and more. The type of rewrite system this project will be focusing on is *term rewriting systems* [40] which focuses on rewriting mathematical terms. A more detailed explanation of term rewriting is covered in Section 2.1.

Term rewriting systems are widely used unknowingly by people, the act of simplifying an equation, one of the fundamental concepts in mathematics that is taught very early in mathematics education is in fact term rewriting, the set of objects here are the equations and the rules are the mathematical rules. Calculators are often used to perform these operations efficiently and these operations are implemented in calculators using term rewriting. A more complex yet important use case for a term rewriting system is *automated theorem proving* [22] [34]. In theorem proving systems, complex predicate logic is simplified using term rewriting until the proof is solved automatically or at a state that it can be easily proved manually. Theorem proving is very important for safety critical systems like aircrafts and medical devices where faults could mean life or death, this makes term rewriting a highly important technique in computer science.

1.2 Motivation

Term rewriting is a basic concept which has simple syntax and semantics [30]. This simplicity allows knowledged users to utilise it to perform mathematical analysis and abstract data type specification analysis, such as theorem proving. However, there is no system that allows inexperienced users to take advantage of this simplicity and perform their own analysis without going through a steep learning curve.

Existing rewrite systems suffer from two main problems, lacking flexibility for users to manipulate the system and lacking an intuitve way for users to interact with the system. Systems such as *Maude* [49] and *Isabelle* [55], are text based and lack a graphical user interface which makes it hard for new users to use [27]. They do provide flexibility by allowing users advanced features like defining rewrite rules, but those have a steep learning curve. Whereas systems like *Wolfram Alpha* [58] are more user friendly, but are limited in terms of flexibility as they do not allow custom rewrite rules. Another lacking feature in most term rewriting systems is ability to perform analysis such as looking at the trace of how the rewrite is performed, all the possible rules that can be applied to a term, etc.

1.3 Aim and objectives

1.3.1 Aim

To address the problems stated above, our aim is to build a tool for term rewriting that allows users to define their own rules, perform analysis, and visualise the rewrite process through a graphical user interface. To achieve the aim, the following objectives are defined.

1.3.2 Objectives

1. Research and evaluate existing rewrite system.

To identify the strengths and weaknesses of existing systems in order to compile a list requirements for our system that addresses their weaknesses.

2. Investigate existing parsing techniques and apply the most appropriate for the system.

To gain a better understanding of different parsing techniques and implement the one most suitable in our system.

3. Design and implement a rewrite algorithm that transforms inputs based on rewrite rules.

Design an algorithm that is able to efficiently rewrite user inputs correctly with regards to the rules that is defined by the user.

4. Investigate and implement advanced rewrite techniques to control rewriting.

To gain an understanding of advanced rewriting techniques that deal with edge cases and implement these techniques to handle errors.

5. Design and implement a Graphical User Interface that allows user to interact with the system.

Design and Implement a graphical user interface in our system that is intuitive and provides ease of use to the user.

6. Investigate and implement different analysis capabilities.

Investigate different types of analysis that can be done on rewriting and implement these feature in the system.

7. Implement textual and graphical visualisation capabilities

Implement visualisation capabilities that will allow users to view and understand different concepts of term rewriting.

8. Evaluate the final system to gauge its effectiveness.

Evaluate the system using a variety of methods and find out if we have met the original aim.

1.4 Project management

1.4.1 Project plan

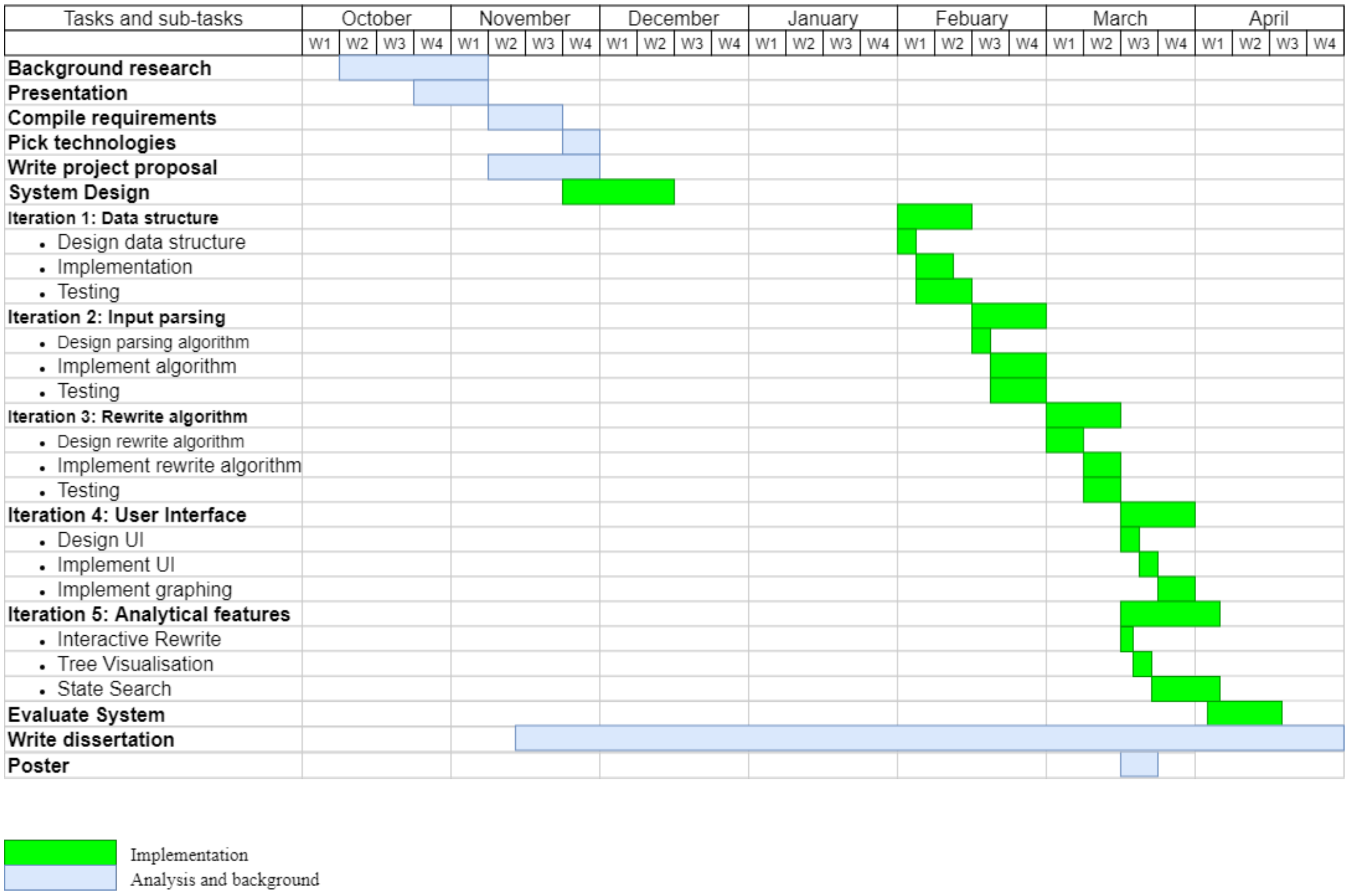


Figure 1.1: Gantt chart

Figure 1.1 shows a Gantt chart created for project management and outlines the major tasks for the project. The project is broken into two phases, research and analysis, blue in the Gantt chart and implementation, green in the Gantt chart.

Research and analysis which are done prior to implementation include looking into existing term rewrite systems in order to indentify their strengths and weaknesses as well as looking into parsing and rewriting techniques. Completion of these tasks achieve objective 1 and 2 which helped us compile a list of requirements for the system.

During the system design stage, a high-level system architecture is designed, outlining the essential components of the system. The implementation stage is then split into iterations, each iteration focuses on one feature of the system. In our system design stage we identified five main components, each of these components will take one iteration where they are desinged, implemented and tested. The list below are the five iterations and their focus.

Iteration 1: Data structure In this iteration, a data structure needed to store user inputs is designed and implemented. The data structure is tree based and designed based on the requirements of the system.

Iteration 2: Input parsing This iteration is focused on developing a parsing mechanism that will transform user inputs into the data structure developed in iteration 1.

Iteration 3: Rewrite algorithm This is a key iteration as it is focused on designing and implementing the algorithm that would perform the rewrite on terms in the form of a tree and following user defined rules.

Iteration 4: User interface In this iteration all the different components of the system are tied together into a graphical user interface.

Iteration 5: Analytical features We have identified three analysis features that can be included in our system, this iteration is to design and implement them. This is done in parellel with the GUI as there are no dependency between them.

Most of the iterations have dependency on the previous iteration and as such are schedule in way that the previous has to be completed before the next one can begin. Each iteration is allocated a timeframe of two weeks however if an iteration finished ahead of schedule, the next iteration shall begin ahead of schedule.

1.4.2 Risks assessment

There will always be risks in project that could potentially impact the timeline in a negative way. To minimise the possibility of failure, we performed risks assesment in order to identify and produce contingency plans.

Features being too hard to implement It is possible that certain features that are planned cannot be implemented due to constraints such as time. Our contingency plan is to pick a development approach that allows us to alter the requirements if needed.

Data loss due to hardware issues There could be unforseen circumstances that hardware failure could result in data loss. Although unlikely, the impact could be catastrophic and as such it is important to take contingency and keep timely backups.

2 Background research

2.1 Term Rewriting Systems

Term rewriting [40] is a simple computational paradigm that is based on repeated application of rules to rewrite a term until it cannot be rewritten further. The word term refers to mathematical terms which are made up of variables, constants and operations [50]. Examples of terms are: $6+3$, $x+y$, $sin(x)$, $A \vee B$, x. Term rewriting systems are suitable for solving problems such as *symbolic computation* [21] and mathematical analysis.

2.1.1 Formal definition

To capture the behavior of a term rewrite system, we can provide a formal mathematical definition of rewrite rule and rewrite system. We start by defining what is a signature and a term.

Definition 1 A *Signature* is a triple $\Sigma = (funcs, vars, ar)$ where $funcs$ is a set of functions symbols, $vars$ is a set of variables symbol and ar is a mapping from function symbol to natural number representing the arity of the function, i.e how many operands it takes. $funcs$ and $vars$ must be disjoint sets. A constant is defined as a function with 0 operands.

Definition 2 $Ter(\Sigma)$ is a set of all possible *terms* under the signature Σ and is defined recursively as follows:
1) If $x \in vars$ then $x \in Ter(\Sigma)$

2) If $c \in funcs$ is a constant then $c \in Ter(\Sigma)$

3) If $F \in funcs$ is a n-nary function $o_1, ..., o_n \in Ter(\Sigma)$, then $F(o_1, ..., o_n) \in Ter(\Sigma)$

Definition 3 A *substitution* is a map $\sigma : v \in vars \mapsto Ter(\Sigma)$. For any $t \in Ter(\Sigma)$, we can replace all occurence of variable v in t with $\sigma(v)$ for a given σ.

Definition 4 Two terms $t_1, t_2 \in Ter(\Sigma)$ *matches*, if there exists a substitution σ such that when applied to t_1 we get $t_1 = t_2$. The substitution σ is called the matching substitution.

Definition 5 A *term rewriting rule* is a pair $l \rightarrow r \in Ter(\Sigma)$.
There are 2 restrictions on a rewrite rule:
1) $l \notin vars$
2) All variables in r are in l as well.
For a valid rewrite rule, any term $t \in Ter(\Sigma)$ that matches l can be replaced by term r.

Definition 6 A *term rewriting system* is a pair $< \Sigma, R >$ given signature Σ and a set of term rewrite rules under signature Σ R. Any term under signature Σ that matches the l of a rule in R can be replaced with the r.

Definition 7 A term $x \in Ter(\Sigma)$ is in its *normal form* if $\nexists y \in Ter(\Sigma) \;\; s.t. \;\; x \to y \in R$ given term rewriting system $< \Sigma, R >$.

2.1.2 Example of term rewriting

Let us consider the following example to illustrate the different components of a term rewrite system. Suppose we have a boolean expression and we would like to evaluate it, using term rewriting we could simplify the expression using the *boolean algebra laws* [23] until we reach an answer.

$funcs = \{\vee, \neg, \wedge, True, False\}$
$vars = \{x\}$
$ar = \{\neg \mapsto 1, \vee \mapsto 2, \wedge \mapsto 2, True \mapsto 0, False \mapsto 0\}$
$\Sigma = \{funcs, vars, ar\}$

(a) Signature of the system

$\neg\neg x \to x$ (Double negation law)
$x \wedge x \to x$ (Idempotent law)
$x \vee True \to True$ (Identity law)

(b) The rewrite rules

$False \vee (True \wedge \neg\neg True)$
$\hookrightarrow False \vee (True \wedge True)$ (Double negation law)
$\hookrightarrow False \vee True$ (Idempotent law)
$\hookrightarrow True$ (Identity law)

(c) Boolean expression rewrite

Figure 2.1: Example of rewriting

In this example, we see the initial term being simplified repeatedly until reaching a state that it does not match any rule and thus cannot be simplified, at this state the term is in its normal form.

2.1.3 The rewrite algorithm

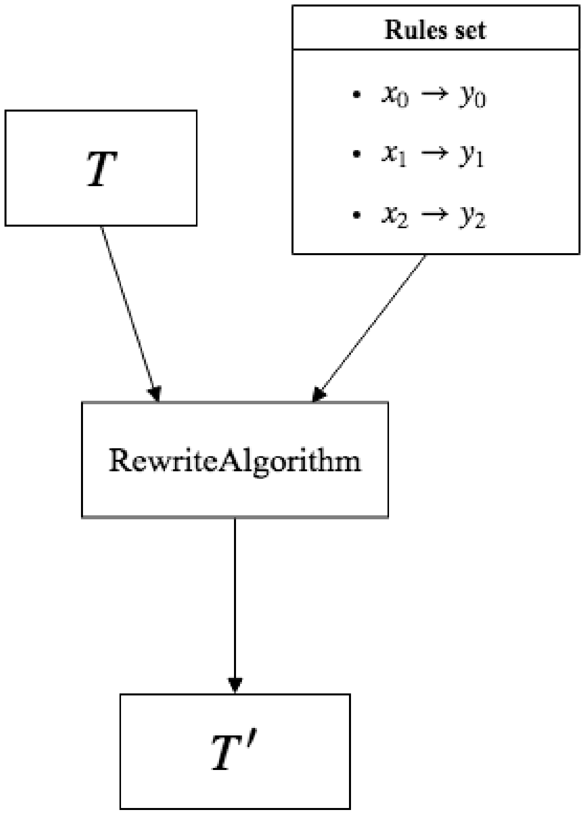


Figure 2.2: Overview of the rewrite process

Figure 2.2 shows a high-level overview of the rewriting process. An inital Term T and a set of rewrite rules are passed into a rewrite algorithm. The system then outputs a term T' that is the normal form of T. The rewrite algorithm is key to this process and should be able to correctly rewrite any term into its normal form [40].

The two main functionalities of a rewrite algorithm is to perform matching and substitution [53]. Given an initial term, the algorithm should try to match any sub-term with a rule and replace it until the term is in its normal form.

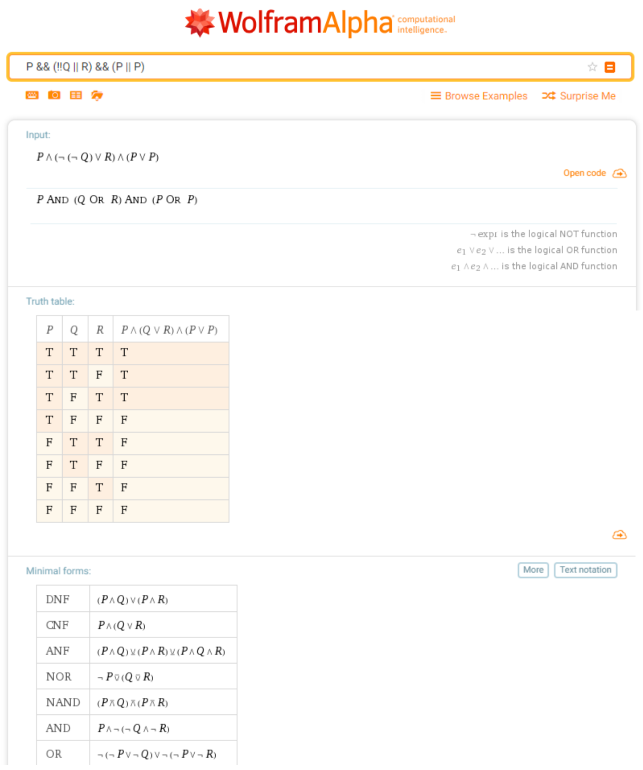
Althought these two functionalities are precisely defined, many aspects of the rewrite algorithm are unspecified. Mainly, how do we select a sub-term to be evaluated and which rule do we apply first. Different implementation of a rewrite algorithm have different methods of choosing the next sub-expression and rule, these are called *rewrite strategies* [13].

2.2 Existing systems

There are a number of existing systems available that uses term rewriting for different purposes. We will be discussing three systems that utilise term rewriting for symbolic computation, theorem proving and general programming.

2.2.1 Wolfram Alpha

Wolfram Alpha [58] is a computational knowledge engine that has many functionalities such as answering queries about factual questions from many different fields, including mathematics. It is able to perform various mathematical computation, some of which are done through term rewriting.



P && (!!Q || R) && (P || P)

Input:

$P \land (\neg(\neg Q) \lor R) \land (P \lor P)$

P AND $(Q$ OR $R)$ AND $(P$ OR $P)$

$\neg$ expr is the logical NOT function
$e_1 \lor e_2 \lor \ldots$ is the logical OR function
$e_1 \land e_2 \land \ldots$ is the logical AND function

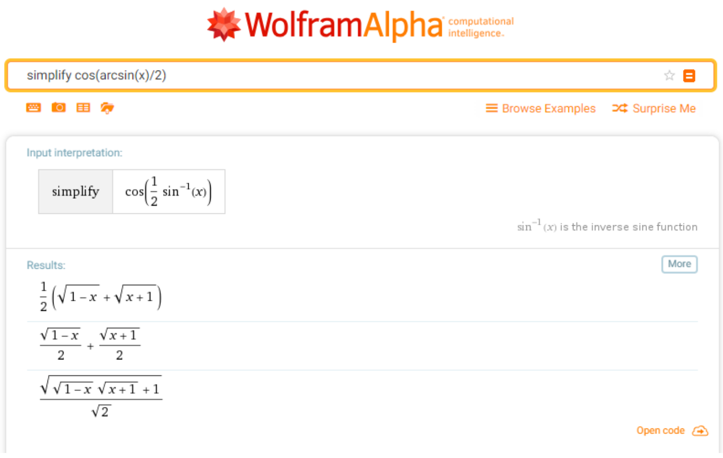
Truth table:

P	Q	R	$P \land (Q \lor R) \land (P \lor P)$
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

Minimal forms:

DNF	$(P \land Q) \lor (P \land R)$
CNF	$P \land (Q \lor R)$
ANF	$(P \land Q) \veebar (P \land R) \veebar (P \land Q \land R)$
NOR	$\neg P \overline{\lor} (Q \overline{\lor} R)$
NAND	$(P \overline{\land} Q) \overline{\land} (P \overline{\land} R)$
AND	$P \land \neg(\neg Q \land \neg R)$
OR	$\neg(\neg P \lor \neg Q) \lor \neg(\neg P \lor \neg R)$

(a) Evaluating boolean



simplify cos(arcsin(x)/2)

Input interpretation:

simplify	$\cos\left(\frac{1}{2}\sin^{-1}(x)\right)$

$\sin^{-1}(x)$ is the inverse sine function

Results:

$$\frac{1}{2}\left(\sqrt{1-x}+\sqrt{x+1}\right)$$

$$\frac{\sqrt{1-x}}{2}+\frac{\sqrt{x+1}}{2}$$

$$\frac{\sqrt{\sqrt{1-x}\,\sqrt{x+1}+1}}{\sqrt{2}}$$

(b) Simplifying Equations

Figure 2.3: Wolfram Alpha symbolic computation

Positve features

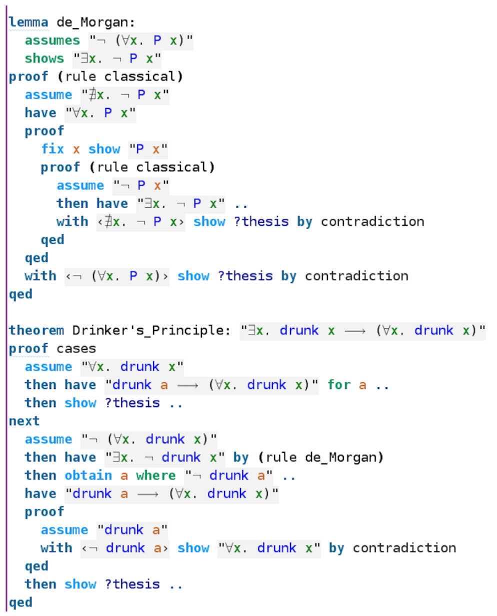
Figure 2.3 shows Wolfram Alpha performing boolean evaluation and simplification which are a form term rewriting. Wolfram Alpha is powerful and provides a lot of different functionalites such as the showing the different types of normal forms. Their main positive feature is that they have a significantly simpler user interface compared to other system due to being primarly targeted towards the general public rather for research purposes.

Limitations

The main limitation is that Wolfram Alpha does not allow custom rewrite rules, it will simplify an equation or logic using all the available rules and return the final result. It does not allow users to just apply a certain rule or define their own rules. Wolfram Alpha also does not provide an easy way for users to trace and analyse the rewrite process.

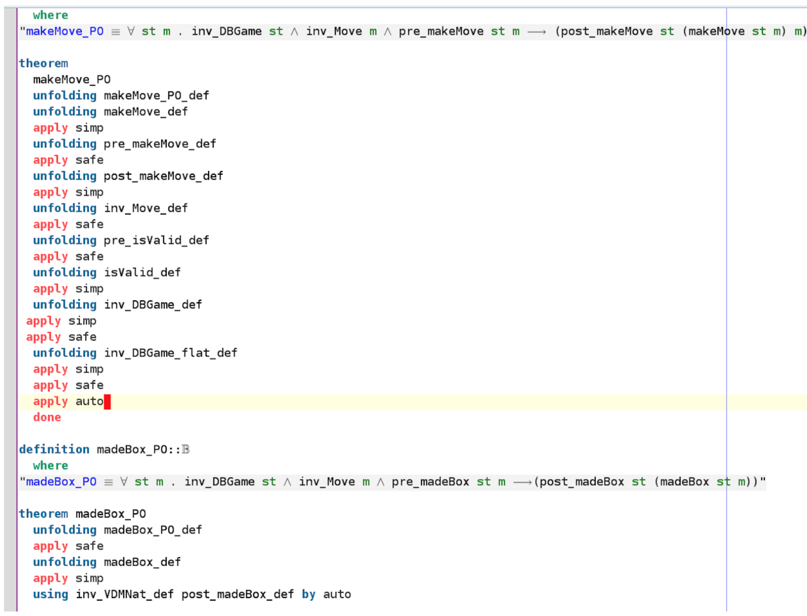
2.2.2 Isabelle

Isabelle [55] is a generic automatic theorem prover that allows users to enter propositional logic and provides tools to perform proofs. Isabelle is widely used in *Formal Methods* [12] for hardware and software specifications.



```
lemma de_Morgan:
  assumes "¬ (∀x. P x)"
  shows "∃x. ¬ P x"
proof (rule classical)
  assume "∄x. ¬ P x"
  have "∀x. P x"
  proof
    fix x show "P x"
    proof (rule classical)
      assume "¬ P x"
      then have "∃x. ¬ P x" ..
      with ‹∄x. ¬ P x› show ?thesis by contradiction
    qed
  qed
  with ‹¬ (∀x. P x)› show ?thesis by contradiction
qed

theorem Drinker's_Principle: "∃x. drunk x ⟶ (∀x. drunk x)"
proof cases
  assume "∀x. drunk x"
  then have "drunk a ⟶ (∀x. drunk x)" for a ..
  then show ?thesis ..
next
  assume "¬ (∀x. drunk x)"
  then have "∃x. ¬ drunk x" by (rule de_Morgan)
  then obtain a where "¬ drunk a" ..
  have "drunk a ⟶ (∀x. drunk x)"
  proof
    assume "drunk a"
    with ‹¬ drunk a› show "∀x. drunk x" by contradiction
  qed
  then show ?thesis ..
qed
```

(a) Isablle editor



```
  where
"makeMove_PO ≡ ∀ st m . inv_DBGame st ∧ inv_Move m ∧ pre_makeMove st m ⟶ (post_makeMove st (makeMove st m) m)

theorem
  makeMove_PO
  unfolding makeMove_PO_def
  unfolding makeMove_def
  apply simp
  unfolding pre_makeMove_def
  apply safe
  unfolding post_makeMove_def
  apply simp
  unfolding inv_Move_def
  apply safe
  unfolding pre_isValid_def
  apply safe
  unfolding isValid_def
  apply simp
  unfolding inv_DBGame_def
 apply simp
 apply safe
  unfolding inv_DBGame_flat_def
  apply simp
  apply safe
  apply auto
  done

definition madeBox_PO::𝔹
  where
"madeBox_PO ≡ ∀ st m . inv_DBGame st ∧ inv_Move m ∧ pre_madeBox st m ⟶(post_madeBox st (madeBox st m))"

theorem madeBox_PO
  unfolding madeBox_PO_def
  apply safe
  unfolding madeBox_def
  apply simp
  using inv_VDMNat_def post_madeBox_def by auto
```

(b) A proof in Isabelle

Figure 2.4: Interactive proofs in Isabelle

Positive features

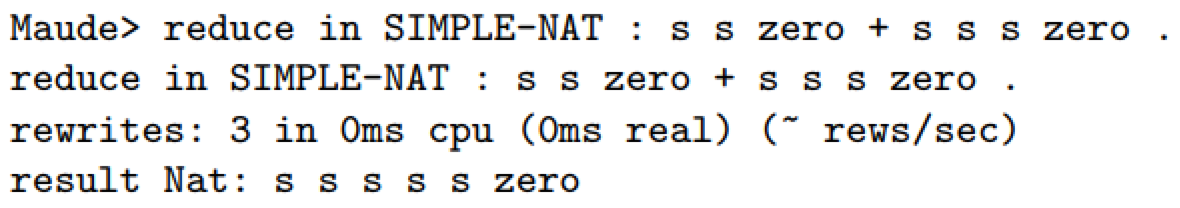
Isabelle uses term rewriting as one of its automatic reasoning tool to perform proofs by performing simplification on an expression until it is trivial to prove. Users interact with the system using an editor that uses the Isabelle syntax to perform these action. Isabelle is interactive and allows users to apply just certain rules on an expression and see the results. Furthermore, Isabelle allows the definition of rules and also data type making it significantly more flexible than Wolfram Alpha.

Limitations

Although Isabelle is powerful and flexible it does not provide a graphical user interface. Users are expected to know the Isabelle syntax and the different Isabelle libraries which can be intimidating to novice users. Furthermore Isabelle does not provide the capability to perform analysis on the rewrite process.

2.2.3 Maude

Maude [49] is a multiparadigm language that relies on rewrite logic. It is highly extensible and powerful and can be used for programming as well as system specifications.



```
Maude> reduce in SIMPLE-NAT : s s zero + s s s zero .
reduce in SIMPLE-NAT : s s zero + s s s zero .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Nat: s s s s s zero
```

Figure 2.5: The Maude language

Features

Maude is capable of doing much more compared to other systems. Using the Full Maude extension, it can be used for specification, concurrent calculi, logic representation, proofs and more, all of which are based on rewriting logic. It uses a module known as “META-LEVEL” to allow rewrite startegies that specifies the way rules are applied which are generally not available on other systems.

Limitations

While Maude is a powerful tool it is not easy to use, learning how to use Maude is comparable to learning a new programming language. Maude does not have an user interface at all and all actions are performed through writing code.

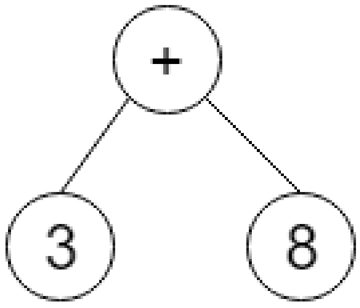
2.3 Parsing

Term rewriting systems need to interpret mathematical terms, because of this parsing plays a huge role in the system.

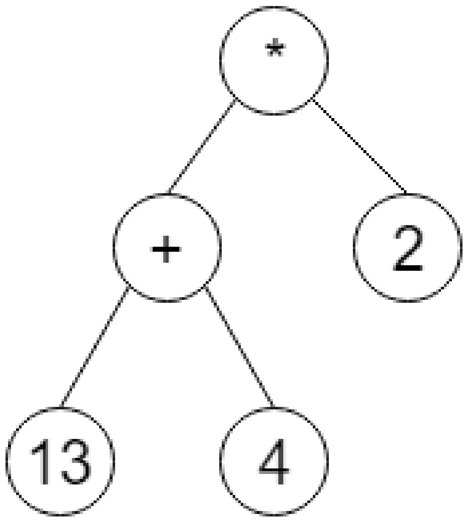
Parsing is the process of analysing stream of texts and turn it into a data structure, often tree based structure. The process of parsing takes two steps, lexical and syntactical analysis. Parsers need to generate a structure that condonned to a specified grammar, which are a set of rules used to recognise patterns in the text.

2.3.1 Abstract Syntax Trees

In this project we are particularly interested in parsing mathematical terms. A commonly used method to represent mathematical terms in a computer is to use an *Abstract Syntax Tree* [28] (AST). ASTs are widely used in compilers as a mean of representing programs however it can also be used to represent mathematical terms as shown in figure 2.6.



(a) AST for expression 3+8



(b) AST for expression (13+4)*2

Figure 2.6: Representing terms using abstract syntax trees

Each node represents either an operator or operand. For a binary operator, its left and right child nodes represent the first and second operands. The tree is evaluate from the bottom-up, higher precedence operations should be lower in the tree as does operations within parenthesis.

2.3.2 Lexical Analysis

Lexical Analysis (also known as tokenization), is the process of taking an input string and converting it into tokens. In the context of terms, the tokens can be operators or operands. The figure belows shows an example of lexical analaysis on a mathematical term.

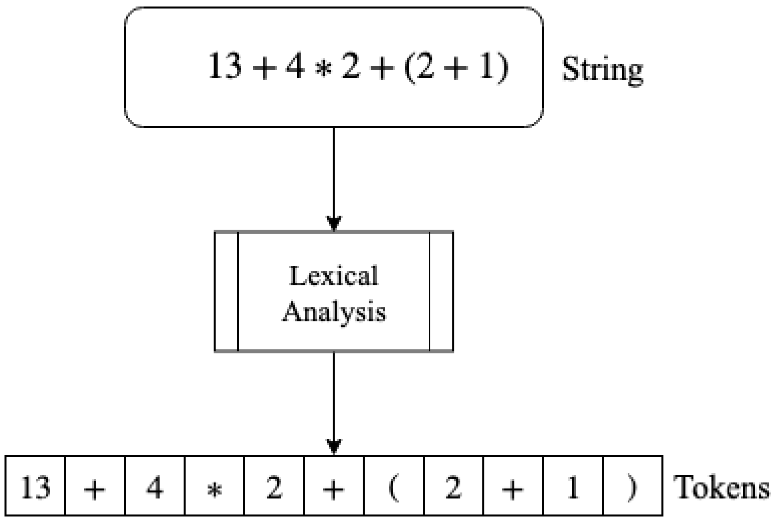


Figure 2.7: The lexcial analysis process

2.3.3 Syntactic Analysis

After the process of tokenizing the string, syntactic analysis is the process of checking the semantics of the tokens. The tokens are checked for any potential error such as mismatch parenthesis, type mismatch and so on. If no errors are present then we can build an abstract syntax tree using the tokens. Syntactic analysis is a complicated procedure and as such many tools exist to help minimise the complexity by automatically generating parser for users given the grammar.

3 Software engineering

3.1 Software development lifecycle

Before we can begin developing any system, we need to set out a plan for development, this is known as the *software development lifecycle* [48]. Two of the most popular software development models are the *Waterfall* [38] and *Agile* [47] development models.

3.1.1 Waterfall method

The waterfall model is regarded as the most classical and basic software engineering model [48]. It follows a linear process where each phase is carried out in sequence, the next phase cannot begin before the current stage finishes. Figure 3.1 below shows the stages of a waterfall model.

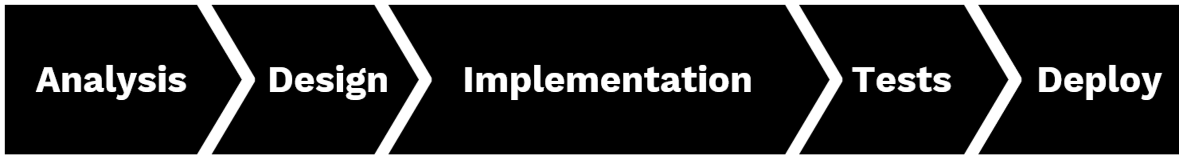


Figure 3.1: The waterfall model

The biggest advantage of using the waterfall model is its simplicity, it is easy to understand and execute. The waterfall method emphasises heavy documentation and validation which reduces the chances of error propagtion throughout development. Lastly, each phase has a clear goal and deliverable, therefore it is easier to measure progress.

However, due to the sequential nature of waterfall, changing requirements during development cycle is difficult as it may lead to having to redesign the entire system. Development time is often longer using the waterfall model and no intermediate product is produced, the system is only visible after the entire process.

3.1.2 Agile method

Unlike the waterfall model which follows a linear process, the Agile model follows an iterative and incremental process that focuses on having continuous delivery of software by having rapid development cycles. There are multiple Agile methods widely used, such as *Scrum* [46], *Lean-Kanban* [8] and more, each with their own pros and cons. Although an Agile process can be done in multiple ways, figure 3.2 shows the general idea of an Agile process.

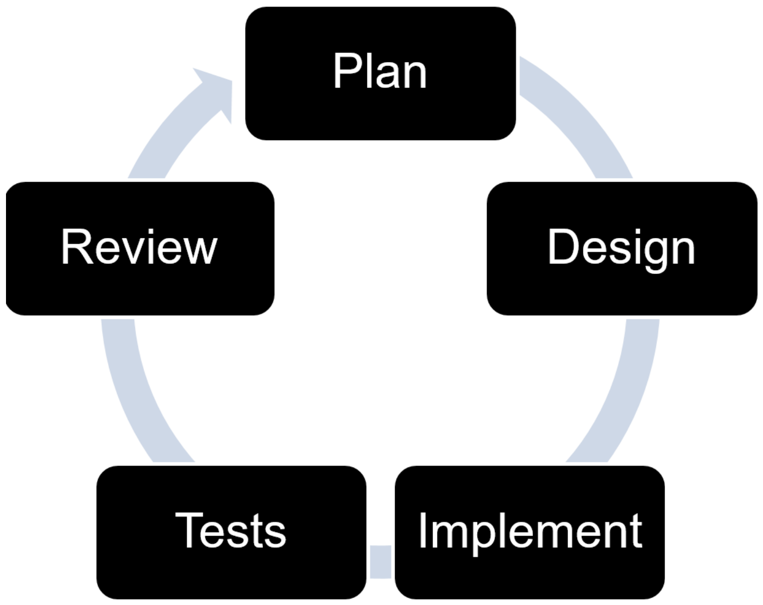


Figure 3.2: The Agile development model

By having quick releases of software, the Agile method is better at measuring real-time progress becuase there will be an intermediate product, this allows bad design to be identified quicker and resolved earlier. Most importantly, the Agile development model is able to adapt to changes in requirements.

The limitations of Agile is that it heavily relies on good communication among team members and as such it works better within small teams. Another disadvantage is that although the method allows us to cope with changing requirements, this is often seen as a waste of time and effort.

3.1.3 Chosen development model

After considering different aspects, the development model we have settled on for this project is the Agile model. We will follow an iterative build model with *Test-Driven Development* [10]. The project is split into multiple iterations where each feature of the system is designed, implemented and tested. Below is a figure showing the process of each iteration.

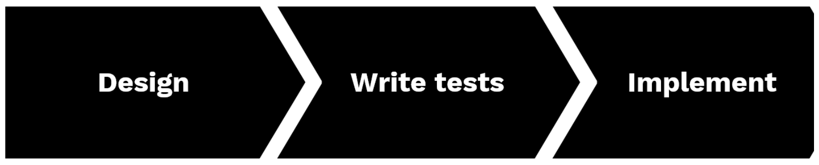


Figure 3.3: Iterative build model with TDD

Design In this stage, the details of the feature is designed, this includes any algorithms, interface, classes, etc. It is important to note that designs may be altered if needed.

Write tests Since we are following a Test-Driven Development approach, the unit tests are written before implementation. The tests are written during this phase and should cover different cases. This stage can be performed concurrently with implementation to maximise efficiency.

Implementation Finally we provide the implementation for the feature and ensure they pass all the test case. We will also perform tests using a driver program during implementation to further ensure the system is correct.

The reason we picked this approach is because it allows us to cope with changing requirements. Since this project is in an unfamiliar aspect of computer science, it is possible some requirements may not be feasible to achieve, by having this approach we are able to alter it. We will also be able to quickly identified any mistakes made easier and correct it without it propagating. Using Test-Driven Development further aid us by allowing us to swiftly run the tests and ensure any changes we have made are correct.

3.2 Tools and technologies

After choosing a suitable development model, we will need to pick the right stack of tools and technologies. There is a wide range of technogy available and we would have to evaluate all of them and pick the most appropriate.

3.2.1 Programming languages

Picking the correct programming language is crucial because the cost of switching a programming language in the middle of development is high. There are many consideration when deciding on a language [4] , we have narrowed down the selection to the modern Object-Oriented languages, *Java* [20], *C++* [25] and *C#* [5].

C++ is a powerful Object-Oriented language. Although being a high-level language, C++ provides low-level memory manipulation functionalities. This makes C++ very fast but comes at a cost of being harder to use and have the potential of being error prone.

Java is a programming language influenced by C++. Unlike C++, it does not have low-level features and lacks performance compared to C++. However, a Java program is compiled on to the Java Virtual Machine (JVM) instead of directly into machine code, making Java very portable. In addition, Java is also less error prone with its internal Garbage Collector.

C# is programming language heavily based on Java. It shares all the pros and cons of Java with the main difference being the runtime enviornment. C# runs on the Microsoft .NET framework which makes it slightly less portable than Java because it is not compatible with non Windows OS.

The language chosen for this project is Java. Since the system is not performance oriented, we can trade speed for an easier and less error prone development. Java has better portability compared to C# and it have multiple libraries we are able to utilise for the project. Another aspect we took into consideration is that we have a strong familiarity with Java and this reduces any possible issue that will arise when working in a foreign language.

3.2.2 GUI libraries

Our graphical user interface can be implemented using libraries. Java provides two well known GUI libraries, Java Swing and JavaFX.

Java Swing has been around for a long time. It was release in 1996 to replace the previous GUI library, AWT. Swing provides a framework to easily create components such as frame, buttons, scrollbar, etc. The main advantage of Swing is that it is lightweight and fast. Furthermore Swing requires a lower version of Java to use making it more accessible.

JavaFX is Java's modern GUI framework. In addition to all of Java Swing's functionality, JavaFX allows using CSS, which are typically used in websites, to style components allowing for much more nicer looking interface. Its main highlight is the abilty create animations and the ability to create richer GUI with a toolkit that is still being updated.

Despite its age, Swing is still a robust framework supported by Java. Our system would be running on the desktop, as such there is no need for JavaFX's web functionality. Our system is also not graphically intensive and would not need JavaFX's rich graphical toolkit, Swing will be sufficient for this project.

3.2.3 Graphics

One of the feature of the system is to generate a graphical view of a tree with all possible rewrite state. There are two Java libraries we can utilise to implement this feature, *JGraphX* [24] and *JUNG* [26].

JUNG (Java Universal Network/Graph Framework) is a Java framework for graph modelling, analysis and visualisation. It supports different representations such as directed, undirected graphs and more. Its standout feature is the inclusion of different algorithms for graph theory, data mining, etc.

JGraphX is a Java library for graph visualisation. Like JUNG, it provides various graph representation, it also provides basic graph algorithms but not as comprehensive as JUNG.

We will be using JGraphX to implement the graphical visualisation because we are only interested in the showing graphs and we do not need the additional functionalities that JUNG provides.

3.2.4 Parsing techniques

Parsing is a key process in this project and can be implemented in the system using two approaches, using a parsing tool like *ANTLR* [52] or implementing an algorithm in the system to perform the parsing.

Parsing tools Using a parsing tool like ANTLR we can provide it grammar and it will generate a syntax tree where the program can operate on. ANTLR is stable however it requires some form of learning before we can use it and it is a heavyweight solution. Another downside when using ANTLR is that we would not have total control of the syntax tree it generates.

Parsing algorithms Although ANTLR is conveniet, it is essentially applying an algorithm to perform the parsing. We might be able to design an algorithm to perform the parsing and we would not have to depend on external tool and have a parser designed precisely for our need. The downside is that our algorithm is might potentially be less stable and it would be harder to extend to other applications.

We have decided on implementing our own parsing algorithm for this project, since we are allowing user defined symbols it would be hard to generate a parser since there is no fixed set of symbols we can provide ANTLR. The cost of generating a new parser everytime the user add a new operator or variable is high. Using an algorithm will allow us to fine tune the parser based on user inputs and generate the appropriate data structure. The algorithm we will base our parsing on is the *Shunting Yard Algorithm* [6] by Dijkstra.

3.3 Documentation

This dissertation serves as a form of docuementation for the project where we recorded our research and any decisions made.

Alongside this dissertation we will be providing documentation of the technical aspect of the development process. All the packages and classes designs will be documented with *Unified Modelling Language* (UML) [44] diagrams which will provide a better understanding of the system to outsiders as well as any future developers not on the original team.

All source code in the project should be commented using JavaDocs comments to provide guidelines. Properly documented code helps developers better understand and use different components of the system which is important because of our software design model which is covered in the next chapter.

4 System design

The first step is to identify what we want the system to do, the system design stage is where we do that by drawing out the main features and the high-level design of the system.

4.1 Requirements

Before we can begin designing our system, we need to analyse the requirements in order to understand precisely what the system needs before diving into designing a solution. We have compiled a set of functional and non-functional requirements to describe what the system will do and how they will do it.

4.1.1 Functional requirements

No.	Requirement	Notes
1	The system will rewrite a term into its normal form following a set of rules.	Given an initial term and set of rules the system should output the normal form of the term.
2	The system will allow users to specify the signature of the system.	User will be able to specify the operators and variables symbol in the system, operators can be unary or binary while variables symbol can be any ASCII string.
3	The system will allow users to specify the rules used to perform rewrite.	Users can define the rewrite rules in the system by providing a left and right hand side term.
4	The system will provide information of the rewrite system.	Users will be able to view basic information of the rewrite system such as the signature of the system and the rules.
5	The system will check for syntax error in user inputs.	The system should not accept any user input that is malformed and should notify the user.
6	The system will guarantee termination for all user defined rules.	The system should be able to recover from any infinite rewrite loop and should terminate for every input.
7	The system will provide visualisation of the rewrite process.	The system will provide a textual visualisation of the rewrite process by providing a step by step trace.
8	The system will allow application of rewrite strategies.	The system will allows users to use rewrite strategies by letting them choose the rule to apply.
9	The system will provide analysis capabilities.	The system will provide different analysis functionalities such as searching and exploring all possible rewrite.

4.1.2 Non-functional requirements

No.	Requirement	Notes
1	The system will provide a graphical user interface.	Users will be able to interact with the system through a graphical user interface.
2	The system will have a window that let users input rewrite rules, operators and variables.	The system will have a dialog window that user can add rewrite rules, operators and variables by provinding the required information.
3	The system will allow users to delete rewrite rules, operators and variables.	Users will be able to remove information from the system which will take effect immidiately.
4	The system will allow saving and loading of rewrite systems.	The system will allow users to save the signature and rules of a rewrite system as a file that can be loaded into a system and reused at a different time.

4.2 High-level system design

The project uses an iterative build model where each iteration is focused on one component, the high-level system design is where we can identify the components needed.

4.2.1 System architecture

Figure 4.1 is an architecture diagram for our system, it uses a *component based architecture* [16] and breaks down the system into multiple components. This method emphasises *separation of concerns* [41], each component has a specific role and functionality which allows the system to be modular and reuseable [54]. As previously mentioned, there is potential that requirements could change throughout production, having modular components allow us to easily swap or alter existing components to deal with the change.

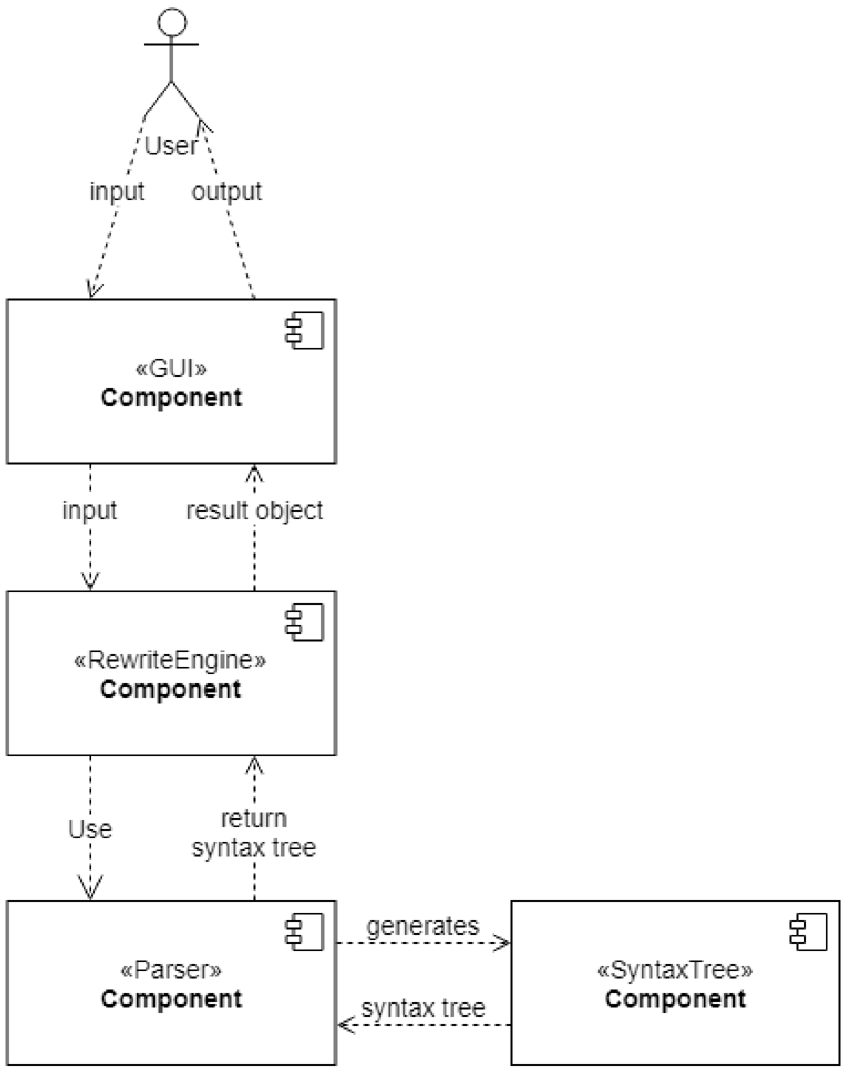


Figure 4.1: High level system architecture

Our software development model and technology stack work well with this architecture, our model allows us to fit each component into an iteration and focus on one component at a time. Futhermore, our chosen programming language Java is fully object-oriented and allows us to represent components as a collection of classes. Considering these and the other advantages of using a component based architecture, our choice of design pattern is obvious.

A component is defined differently in different types of software, in our system a component is a package. Packages are be made up of classes with related funtionality and each package is responsible for one main task. The list below shows the packages that would be in our system and their main functionality.

Syntax Tree This is the internal tree data structure that is used to represent a term. Although it does not provide much in terms of functionality it is important to our system as it allows other components to perform actions on a term efficiently.

Parser The parser is reponsible for converting user inputs into the syntax tree. It uses an algorithm to perform the conversion that other component can utilise without knowing the technical details of the function. In addition, the parser is able to convert a syntax tree back into string so that it can be displayed back to the user.

RewriteEngine This component acts as the "Engine" of the system where most of the algorithms for rewriting and analysis are implemented. The engine will use the parser to generate a syntax tree that it can operate on however since it does not need to know the technical details, we can separate it from them and provide a level of abstraction and modularity.

GUI This component provides all the graphical side of the system. Each classes in the package is responsible for a graphical element such as a panel, frame, graphical visualisation, etc. This component serves as the entry point of the system. Saperating the component from the engine itself allows us to manage the system easier as everything related to graphics would be in one place.

5 Syntax tree and parsing

This chapter is about the process of building the first two components the syntax tree and the parser.

5.1 Design

The design phase is focused on designing the classes needed for the syntax tree and the parser. For the syntax tree, we need classed to represent tree nodes and operators. We need an *Operator* class because just representing operators as a string is not enough, there is a lot more information associated with an operator.

We started by designing the syntax tree package since it has to be completed before we can start development on the parser.

5.1.1 Designing the operators

The system supports two types of operators, unary and binary. Unary operators take one operand and is written in a *prefix* [56] manner. Examples of unary operators are $NOT\ \ True$, $succ\ \ 12$, $\neg x$. Binary opearators have two operands and are written in an *infix* [56] style. For example $x + y, A\ AND\ B$.

The operator classes would represent these two types of operator by storing the symbol, precedence and return type. We need the precedence as it is crucial when parsing and the return type is needed when performing rewriting. An enumeration type, DataType is created to represent the return type of operators. The data types included are *int*, *boolean*, *nat*, *string*, *const*.

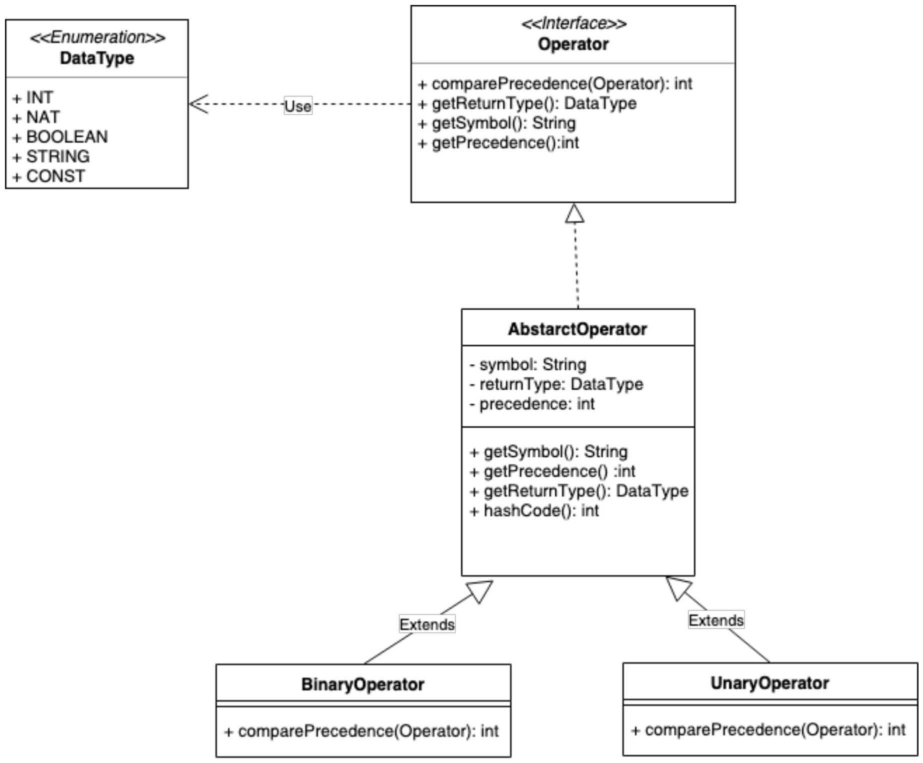


Figure 5.1: Operators' class diagram

Figure 5.1 is a UML class diagram for the operator classes which are designed using a *Facade design pattern* [7]. It has a simple interface to define functionalities every operators need, mainy comparing precedence with another operator. The *comparePrecedence* method should return 1 when compared to an operator with a lower precedence, 0 for equal precedence and -1 if compared to a higher precedence operator. The *AbstarctOperator* class implements all the accessor methods in the interface except the *comparePrecedence* method. We then split unary and binary operator into saperate classes which will implement the *comparePrecedence* method.

Comparing the precedence of operators is trivial, we compare if the precedence value is higher, lower or equal. We decided that all unary operators take precedence over binary operators. This decision is made as it makes functions and unary operators' syntax look more natural.

Having made that declaration, there are three possible cases for the*comparePrecedence* method.

Case 1: Comparing same type of operators If we are comparing the same type of operators, then we just simply compare their precedence.

Case 2: Comparing binary with unary In this case, we will always return -1 as we have established that binary always have a lower precedence when compared to unary.

Case 3: Comparing unary with binary In this case, we will always return 1 as it is just the reverse of the case 2.

5.1.2 Designing a tree node

The initial design for the nodes of the syntax tree follows a similar design pattern to the operators, having sub-classes representing different nodes (See inital design in appendix A). However, during development this has to be changed to accomodate with a limitation of our programming language (Further explaination in section 7.3.2). The final design of the syntax tree is shown below.

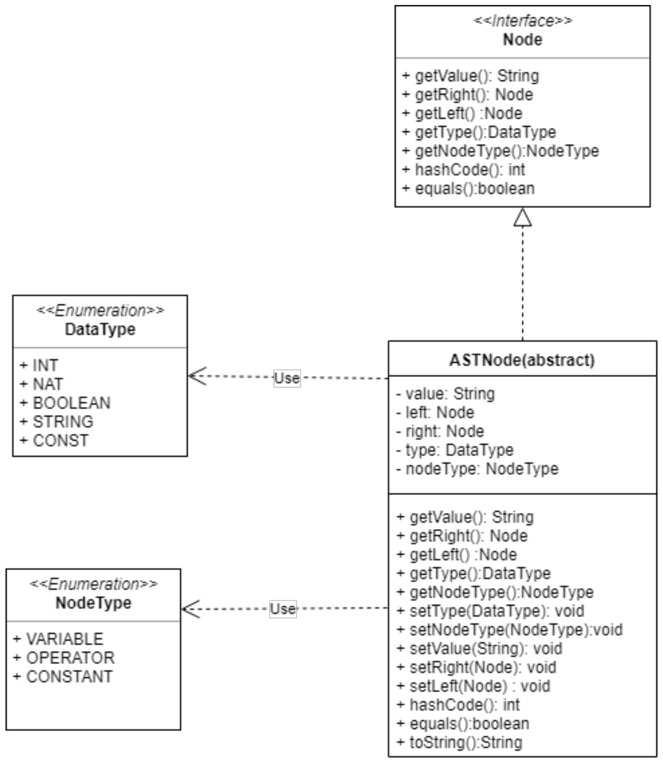


Figure 5.2: Syntax Tree Node Class Diagram

The enumeration type, *NodeType* is used to represent the type of value the node holds, whether it is a variable, operators or constant.

A Node also have a *type* field that holds the type of a variable if the node is holding a variable, or the return type of an operator if the node is holding an operator. A decision is made to keep constants untyped because it reduces the complexity of the systems and makes it much more flexible to the user.

In addition to the standard accessing and modifying the child nodes, we need to be able to compare two nodes and deem if they are equivalent. We decided that two nodes are equal if all fields are equal, we can recursively compare them with the algorithm below.

Algorithm 1 Comparing tree nodes

```
procedure EQUALS($n$)
    if $left = null$ then
        if $n.left \neq null$ then
            return $false$
        end if
    else if $left.equals(n.left) = false$ then
        return $false$
    end if
    if $right = null$ then
        if $n.right \neq null$ then
            return $false$
        end if
    else if $right.equals(n.right) = false$ then
        return $false$
    end if
    if $type \neq n.type$ then
        return $false$
    end if
    if $nodeType \neq n.nodeType$ then
        return $false$
    end if
    if $value \neq n.value$ then
        return $false$
    end if
end procedure
```

5.1.3 Designing the parser

Figure 5.3 shows the classes in the parser package, we need a class for the parser itself as well as a *Signature* class that represents the signature of the system.

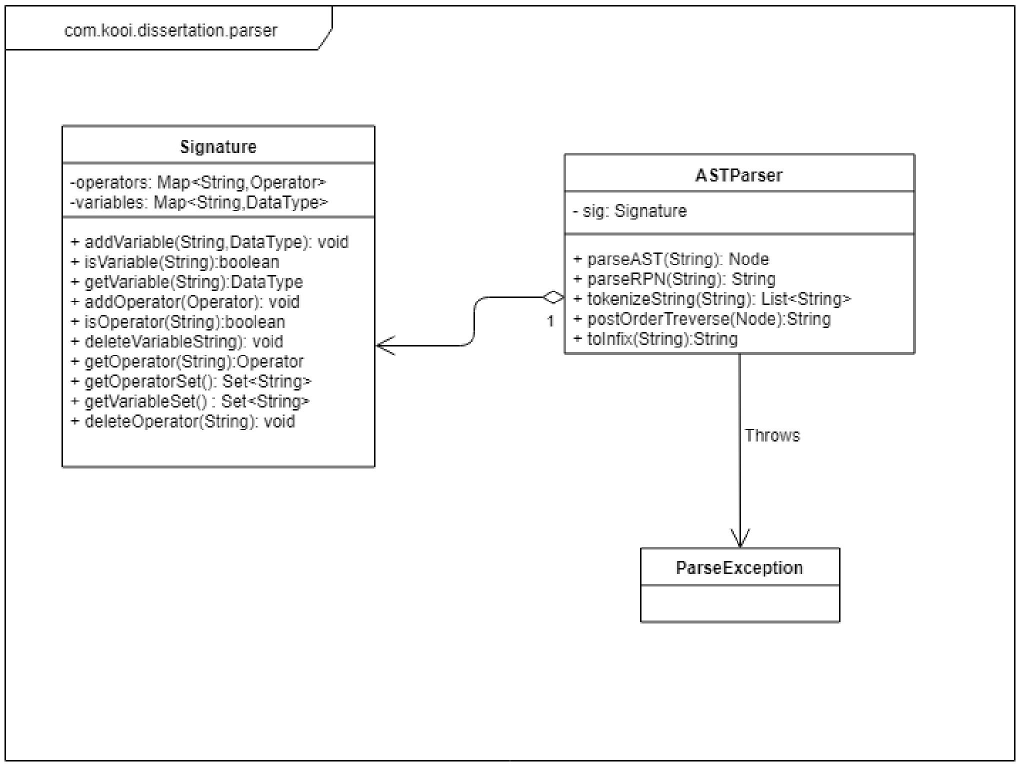

com.kooi.dissertation.parser
Signature
-operators: Map<String,Operator>
-variables: Map<String,DataType>
+ addVariable(String,DataType): void
+ isVariable(String):boolean
+ getVariable(String):DataType
+ addOperator(Operator): void
+ isOperator(String):boolean
+ deleteVariableString): void
+ getOperator(String):Operator
+ getOperatorSet(): Set<String>
+ getVariableSet() : Set<String>
+ deleteOperator(String): void
ASTParser
- sig: Signature
+ parseAST(String): Node
+ parseRPN(String): String
+ tokenizeString(String): List<String>
+ postOrderTreverse(Node):String
+ toInfix(String):String
1
Throws
ParseException

Figure 5.3: Parser package's class diagram

The signature class The signature of the system is represented by this class, recall from definition 1, a signature has a set of variable symbols and operator symbols, which are represented here using a map, variables map from string symbol to DataType and operators are mapped from string symbol to *Operator*. Mapping from the operator's symbol to the *Operator* objects will allow us to retrieve the required information such as precedence and arity.

The ASTParser class The ASTParser class provides functionalities to parse a string into either *postfix* (Also known as Reverse Polish Notation, RPN) [56] or a syntax tree. As previously mentioned, parsing has two stages, lexical analysis and syntactic analysis. The *tokenizeString* method performs the former by splitting the string given the operators. The *parseAST* and *parseRPN* methods perform syntactic analysis and parses the string into their respective forms, syntax tree and Reverse Polish Notation. Since infix is the notation displayed back to the users, the parser will also be able to convert from RPN to infix notation (See detailed implementation of conversion in appendix B).

5.2 Designing the parsing algorithm

Our parsing algorithm is based on Dijkstra's *Shunting-Yard Algorithm* [6]. The original algorithm describes converting from infix into postfix however it is possible to modify it to produce a syntax tree instead. The pseudocode below shows our modification and implementation.

Algorithm 2 parse infix expression into AST

```
procedure PARSEAST($s$)
    $opStack \leftarrow Stack$
    $nodes \leftarrow Stack$
    $tokens \leftarrow tokenize(s)$
    for $t$ in $tokens$ do
        if $t =$ "(" then
            $opStack.push(t)$
        else if $t =$ ")" then
            while $opStack$ not empty do
                $p \leftarrow opStack.pop()$
                if $p =$ "(" then
                    $break$
                else
                    $addNode(nodes, p)$
                end if
            end while
        else
            if $t$ is Operator then
                while $opStack$ not empty do
                    $last \leftarrow opStack.peek()$
                    if $last =$ "(" then
                        $break$
                    end if
                    if $current.comparePrecedence(last) \leq 0$ then
                        $addNode(nodes, opStack.pop())$
                    else
                        $break$
                    end if
                end while
                $opStack.push(t)$
            else
                $n \leftarrow Node(t)$
                $nodes.push(n)$
            end if
        end if
    end for
    while $opStack$ not empty do
        $addNode(nodes.opStack.pop())$
    end while
    return $nodes.peek()$
end procedure
```

Algorithm 3 Add node to the tree

```
procedure ADDNODE($stack, s$)
    $right \leftarrow stack.pop()$
    $left \leftarrow stack.pop()$
    if $s$ is $BinaryOperator$ then
        $stack.push(Node(s, left, right))$
    else if $s$ is $UnaryOperator$ then
        $stack.push(left)$
        $stack.push(Node(s, nil, right))$
    end if

end procedure
```

The algorithm uses two stacks, *nodes* to track the nodes that are currently in the tree and *opStack* to track the operators that are to be added. The algorithm then loops through all tokens where there can be four cases.

Case 1: token is an operand If the token is an operand then the algorithm just creates a new node with no child and add it to the stack. In our system, any token that is not a variable or operator is automatically treated as a constant. We chose to do this because requiring users to declare all constants is infeasible as certain types like natural number have infinite number of constants.

Case 2: token is an operator When the algorithm encounters an operator we cannot just add it to the tree because we have to consider the precedence rule and add higher precedence operators first. We go through *opStack* and add all previous operators with a higher precedence compared to the current one until we encounter one with a lower precedence, we then push the current operator onto the stack.

Case 3: token is an opening parenthesis We simply push the opening parenthesis onto the stack.

Case 4: token is a closing parenthesis If we hit a closing parenthesis, we add all the operators from *opStack* into the tree until we pop off the opening parenthesis, this ensures everything in the parentheses are lower in the tree and evaluated first. Note that case 2 will ensure we keep precedence rule within the parentheses

After the loop, the rest of the operators are added, at this stage poping off *opStack* will follow precedence rule. At the end of the algorithm, the top element of the *nodes* is the root of the tree.

Given the stack *nodes* and an operator to be added, we use *addNode* to add the operator to the tree by using top two elements of *nodes* as the right and left operands respectively. We need to perform a check before adding an operator, if we are adding a binary operator we create a new operator node with two child nodes and add it to the tree, otherwise we have an operator node with only one child.

5.3 Implementation

5.3.1 Comparing tree nodes

Implementation of the comparison of tree nodes is done by overriding the *equals* method in Java. This allows the classes to be compatible with the *Java Collections library* [39] and allow unit testing to be easier. The implementation is straightforward and can easily be done using the IDE's code generation feature as shown below.

```
@Override
public boolean equals(Object obj) {
   if (this == obj)
      return true;
   if (obj == null)
      return false;
   if (getClass() != obj.getClass())
      return false;
   ASTNode other = (ASTNode) obj;
   if (left == null) {
      if (other.left != null)
         return false;
   } else if (!left.equals(other.left))
      return false;
   if (nodeType != other.nodeType)
      return false;
   if (right == null) {
      if (other.right != null)
         return false;
   } else if (!right.equals(other.right))
      return false;
   if (type != other.type)
      return false;
   if (value == null) {
      if (other.value != null)
         return false;
   } else if (!value.equals(other.value))
      return false;
   return true;
}
```

5.3.2 Tokenizing the string

Our initial approach to tokenizing limits operators to be only a character, we would split the string using a linear algorithm that added the substring between the previous and the current operator. However, we were able to come up with a better approach that removes this limitation.

The final tokenizing approach utilises *Regular Expressions* [33] and Java's *split*() method from the String library. We would build a regular expression that would match all the user provided operations and parenthesis. This regular expression is used as the delimiter to split the string, *split*() returns an array of string that is split using anything that matches the delimiter.

Passing the regular expression to the method as the delimiter is not enough as this will not include them in the array. To keep the delimiters, we have to use a technique called *"lookaround"* [42]. This allows us to split the string into an array while preserving the delimiters. The code is shown below, note that *isMetaCharacter* is a helper function to check if the string is a reserved character for regular expression.

```java
public List<String> tokenizeString(String s){

   //remove white spaces from the input
   s = s.replaceAll("\\s+","");

   //build the regex
   StringBuilder regex = new StringBuilder();

   for(String c: sig.getOperatorSet()) {
      if(isMetaCharacter(c))
         regex.append("(\\"+c+")|"); //escape the special character
      else
         regex.append("("+c+")|");
   }
   regex.append("(\\()|");
   regex.append("(\\))");

   //using java string split and look ahead and look behind regex,
   //we can split the string while preserving the delim(operators)

   String[] arr =
       s.split("((?<=("+regex.toString()+"))|(?=("+regex.toString()+")))");

   return Arrays.asList(arr);
}
```

5.3.3 Implementation of the parsing algorithm

We were able to implement the parsing algorithm in Java easily using the Java Collections Framework which provides the required data structures such as the stack. We have also made sure to implement error checking to detect malform expression such as mismatch parentheses. The full implementation of the algorithm is included in appendix C.

During testing we discovered that if the user started a term with a binary operator, the algorithm will attempt to pop 2 elements off the stack when there is one, which led to an *EmptyStackException*. There are two possible fix, check that the first token is not a binary operator before parsing, alternately check if the stack has elements before popping. We chose the former and checked that the first token is not a binary operator as it is syntatically wrong to have a bianry operator with only one operand.

Building on the issue above, we discovered that it is possible to generate a syntax tree with syntax error. For example, if given term $2 + - * 2$ we can generate a syntax tree. Our fix is to check that all binary operator are not preceded and succeeded by binary operator, we have to also factor in parenthesis when validating the tokens. The validation method is shown below.

```
    private boolean verifyTerm(List<String> tokens) {

    for(int i=0;i<tokens.size();i++) {
       String t = tokens.get(i);

       if(sig.isOperator(t)) {
          Operator o = sig.getOperator(t);

          if(o instanceof BinaryOperator) {
             if(i == 0 || i == tokens.size()-1 ||
                 sig.isOperator(tokens.get(i-1)) ||tokens.get(i-1).equals("(")
                 || tokens.get(i+1).equals(")") ||
                 (sig.isOperator(tokens.get(i+1)) &&
                 sig.getOperator(tokens.get(i+1)) instanceof BinaryOperator) ) {
                return false;
             }
          }
       }
    }
    return true;
 }
```

5.4 Conclusion

Overall the two iterations were smooth and we did not encounter any major complication aside from having to alter the design, which our development model allowed us to do so without repercussions. Parsing proved to be a challenging problem however we were pleased with our solution, our algorithm is efficient with a linear time complexity and is able to generate a syntax tree correctly. Having a data structure and a parser for said data structure, we can move on to the third iteration that is designing the rewrite algorithm.

6 Developing the rewrite engine

The focus of this iteration is building the rewrite engine, which mainly involves designing the algorithm. This chapter will be discussing the algorithm in detail and how it is implemented.

6.1 Design

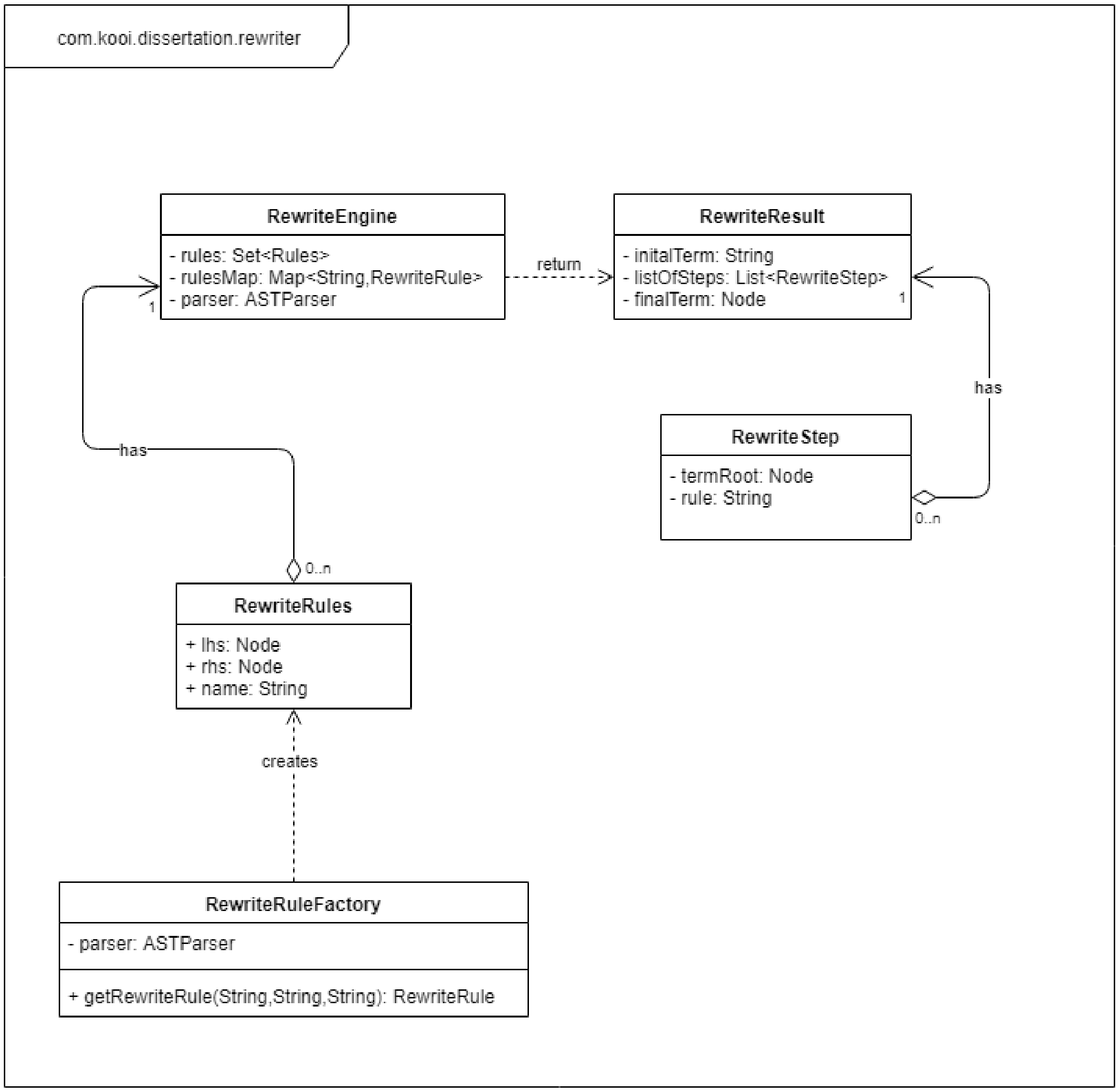


Figure 6.1: The Rewrite package diagram

Before we can implement the algorithm, we need classes to represent the rewrite rules and result. The UML diagram above shows the classes in this package, methods are ommited from the diagram to make improve legibility.

6.1.1 The RewriteRule class

Storing a rewrite rule is simple, we only need to provide a class with three fields, the left-hand side and the right-hand side, both as the root nodes of the syntax tree, another field is included to hold the name of the rule. There was a decision to make if we should store the left and right hand side as strings or Nodes, we chose to go with storing nodes as it meant we only have to perform parsing once and that is when we create them.

Since we would always need to parse input before creating a rewrite rule, we used a *Factory Design Method* [59]. The *RewriteRuleFactory* class has a *ASTParser* and contains a sole method to generate rewrite rule given the strings. This approach provides a level of abstraction and makes the code flexible [7] as if we decide to alter the parsing we only need to change it in one place.

6.1.2 Storing rewrite result

To store the result, we created two classes which are *RewriteStep* and *RewriteResult*. The *RewriteStep* stores each indivual step, it has a reference to a root node that represents the current state and a string representing the name of the rule applied on the previous state. We decided to store a reference to a node instead of as a string in infix form because it allows us to perform more actions on the term if needed.

The *RewriteResult* class is an immutable class storing all the data involved in a rewrite, the initial term provided by the user, a list of *RewriteStep* to store the intermediate stage and the final term as a root node. We decided that it is sensible for it to be immutable as allowing changes could potentially invalidate the result.

6.2 Designing the rewrite algorithm

This is the crux of the matter, designing an algorithm that would perform term rewriting over our syntax tree. We tackled the problem by designing the algorithm on paper before writing code as it is significantly easier to visualise a tree on paper than over the command line.

A total of 4 algorithms are needed to solve this problem, the 4 algorithms are:

Matching Determine if two root nodes match and if they match provide the matching substitution (Definition 4).

Substitution Given two root nodes m, n and a matching substitution, change m to be equal to n.

Swap Given a root node and a matching substitution, replace the any variable in the tree with the value in the matching substitution.

Search Traverse a tree and search for a match between a rule and a sub-tree.

6.2.1 Matching algorithm

The figure below shows an example of what we are trying to achieve. The tree on the left represents the term $succ(10+6)+5$ and the tree on the right is $succ(x)+y$. The colours represent the match, with $y \mapsto 5$ and $x \mapsto (10+6)$

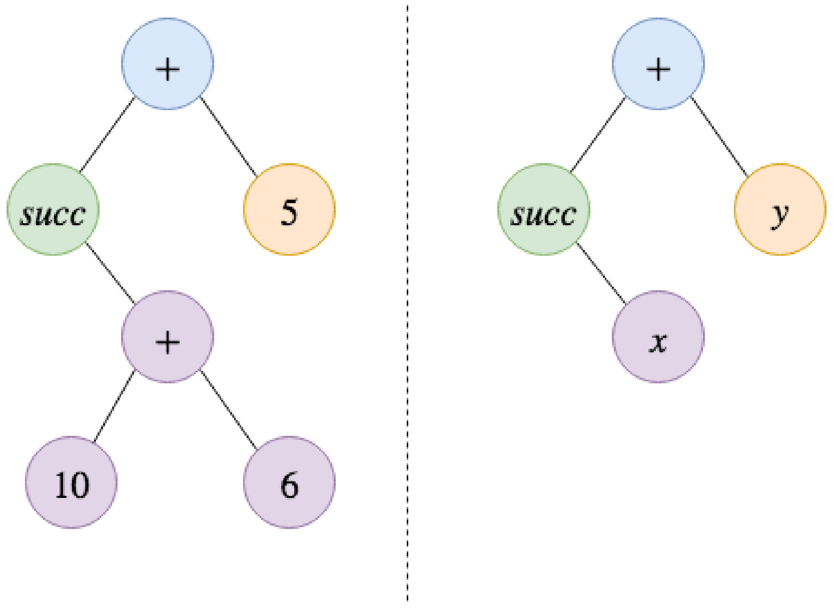


Figure 6.2: An example of matching

In order to match the two syntax trees, *term* and *rule*, we need to traverse and match all the nodes, keeping track of variables matched along the way. There are two cases when matching, matching with a variable and matching with a non-variable.

Case 1: Matching with a variable When we are come across a variable node in *rule*, we can match it if the *term* node is a constant or an operator with the same return type as the variable. It is important to check that if the variable has been matched before, the mapping is equal, otherwise it does not match. For example, if $x \mapsto True$ then we cannot match another $x \mapsto False$. We do not need to traverse further if we matched with a variable as it can be matched with the entire sub-tree.

Case 2: Matching with a non-variable If we are matching with a non-variable node, then it is trivial. It will only match if the nodes have the same symbol and both their sub-trees match.

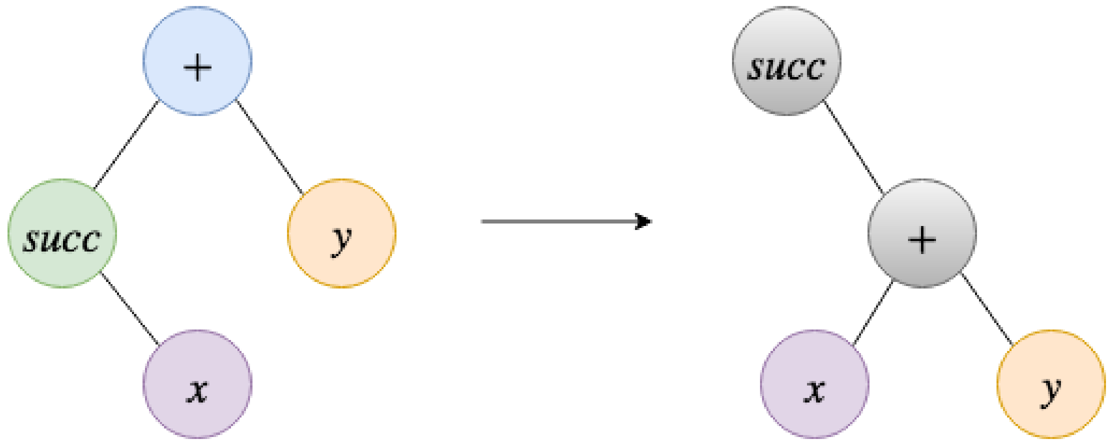
In addition to the two nodes, the algorithm also takes a map $varMap$ as parameter, this will be the matching substitution at the end of the algorithm. The algorithm returns a boolean signifying if the two nodes match, the pseudocode below shows the algorithm.

Algorithm 4 Matching term with rule

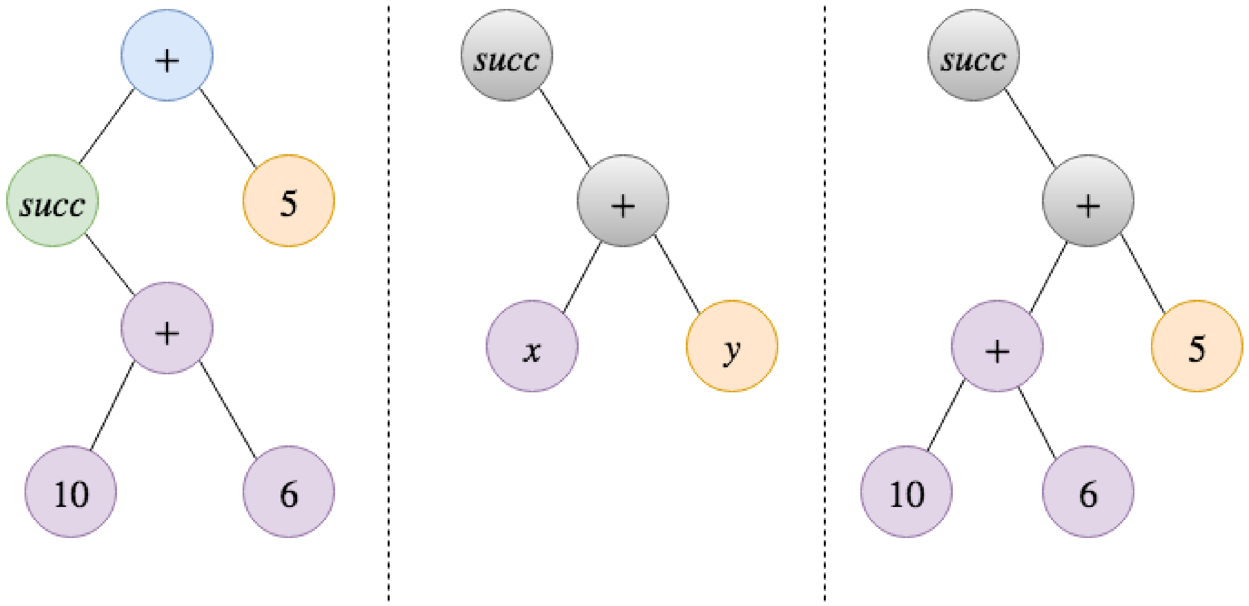
```
procedure MATCH($term, rule, varMap$)
    if $term = nil$ $or$ $rule = nil$ then
        return $term = rule$
    end if
    if $rule$ $is$ $variable$ then
        if $varMap$ $contains$ $rule.value$ then
            return $term = varMap(rule.value)$
        else
            if $term$ $is$ $operator$ $and$ $term.type = rule.type$ then
                $varMap(rule.value) \leftarrow term$
                return $true$
            else if $term$ $is$ $constant$ then
                $varMap(rule.value) \leftarrow term$
                return $true$
            else if $term$ $is$ $variable$ $and$ $term.type = rule.type$ then
                $varMap(rule.value) \leftarrow term$
                return $true$
            else
                return $false$
            end if
        end if
    else
        if $term.value = rule.value$ then
            return $match(term.left, rule.left, varMap)$ and
              $match(term.right, rule.right, varMap)$
        else
            return $false$
        end if
    end if
end procedure
```

6.2.2 Substitution algorithm

Once we determined that a term matches with a rule, we need to perform the substitution. Our approach is to change the term to be equal to the rule, then traverse it and replace all variables with the matching substitution. A helper function *copy* is used to make a copy of a node by creating a new reference. The figure below shows a high level overview of the transformation.



(a) The rule matched



(b) The swapping process

Figure 6.3: High level trace of the substitution algorithm

In figure 6.3(b), the initial term matches the left-hand side of the rule in figure 6.3(a). It is completely replaced by the rule's right-hand side. Similar to figure 6.2, the colour of the variables correspond to their mapping which are swapped into the final term. This is achieved by using the two algorithms below.

Algorithm 5 Substitute term with rule

```
procedure SUBSTITUTE($term, rule, varMap$)
    $term \leftarrow copy(rule)$
    $swap(term, varMap)$
end procedure
```

Algorithm 6 Swap variables with value

```
procedure SWAP($n, varMap$)
    if $n = nil$ then
        return
    end if
    if $n.value\ in\ varMap$ then
        $n \leftarrow copy(varMap(n.value))$
    end if
    swap(n.left,varMap)
    swap(n.right,varMap)
end procedure
```

6.2.3 Search algorithm

The search algorithm is responsible for traversing the syntax tree and find any potential matches. Since we are storing all intermediate stage, the algorithm will perform at most one match and substitution. The algorithm returns a boolean to show if a rewrite took place.

There are two ways to search a tree, *depth-first* [43] and *breadth-first* [2]. There is no clear advantage in picking one over the other here so we opted for a depth-first search as its recursive nature works well with the other algorithms.

In terms of ordering, we went with a post-order traversal as it has the potential of genarating more steps compared to an in-order traversal where the root could be matched first and eliminating the entire sub-tree. This would be an advantage if our goal is to perform rewriting as fast as possible but having more steps we can provide users a better understanding of rewriting. The pseudocode below shows the algorithm, *rules* is a list of rewrite rules for the system.

Algorithm 7 Search syntax tree for matches

```
procedure SEARCH($node, rules$)
    if $node = null$ then
        return $false$
    end if
    if $Search(node.left, rules) = true$ then
        return $true$
    end if
    if $Search(node.right, rules) = true$ then
        return $true$
    end if
    for $r$ in $rules$ do
        $vars \leftarrow Map(String \mapsto Node)$
        if $match(node, r.leftHandSide, vars)$ then
            $subtitute(node, r.rightHandSide, vars)$
            return $true$
        end if
    end for
    return $false$
end procedure
```

Lastly, we would need to repeatedly apply the algorithm on the inital term until the whole tree is traversed and no rule match, (i.e the tree is in its normal form). A simple loop with a flag to indicate whether a rewrite took place can accomplish this, if the whole tree is traversed with no match it means it is in its normal form.

Algorithm 8 Rewrite a term

```
procedure REWRITE($term, rules$)
    $node \leftarrow parseAST(term)$
    $flag \leftarrow false$
    repeat
        $flage \leftarrow search(node, rules)$
    until $flag = true$
    return node
end procedure
```

6.3 Implementation

Implmenting all the algorithms above is not straightforward, this section discusses changes made and problems encountered during implementation.

6.3.1 Differences in implementation

The algorithms are implemented in the *RewriteEngine* class, the final Java code is mostly similar compared to the pseudocode with the exception of some modifications made for efficiency and to provide additional functionalities.

Performance tweak In the search algorithm, the rules are member fields of the class and therefore does not need to be passed into the method. We have also used a *HashMap* to provide an effienct lookup by grouping all the rules with the same root node which allows us to discard all the rules that would not match because the root nodes between term and rule are different. Bearing that in mind, the code snippet below shows the implementation of the *search* method.

```java
private boolean search(Node n,StringBuilder rName) throws RewriteException {

      if(n == null)
         return false;

      //if something is rewritten don't rewrite further
      if(search(n.getLeft(),rName))
         return true;
      if(search(n.getRight(),rName))
         return true;

      //try all the possible rule that can be match
      if(ruleMap.get(n.getValue()) == null)
         return false;

      for(RewriteRule r : ruleMap.get(n.getValue())) {
         Map<String,Node> vars = new HashMap<>();

         if(match(n,r.getLhs(),vars)) {
            substitute(n,r.getRhs(),vars);
            if(rName != null)
               rName.append(r.getName());
            return true;
         }
      }
      return false;
}
```

Storing intermediate result To store the intermidiate result, the *search* method would need to provide the name of the rule last applied. The *rewrite* method would then build a *RewriteStep* after each loop. In the end, it returns a *RewriteResult*

Guaranteed termination Termination of a rewrite system is undecided just given the rules [32]. However we can guarantee termination using a bound, which is the maximum iterations a rewrite can take. If we do not reach a normal form after N iterations, we signal to the user there is potentially a problem with the system.

The code snippet below is the implementation of the *rewrite* method and shows how we implemented the last two extra features.

```
   public RewriteResult rewrite(String infix) throws
       ParseException,RewriteException{

      int c = 0; //total iteration, prevents infinite

      Node root = parser.parseAST(infix);

      if(!checkTerm(root))
         throw new RewriteException("Term to rewrite cannot contain declared
             variable.");
      boolean flag = false;
      StringBuilder lastRule;
      List<RewriteStep> steps = new ArrayList<>();

      do {
         lastRule = new StringBuilder();
         flag = search(root,lastRule);

         //skips the final step
         if(flag) {
            RewriteStep step = new RewriteStep(copy(root),lastRule.toString());
            steps.add(step);
         }
         c++;
      }while(flag && c<MAX_ITERATION);

      if(c == MAX_ITERATION)
         throw new RewriteException("Max Itertion reached, possible infinite
             rewrite");

      return new RewriteResult(infix,steps,root);

}
```

6.3.2 Problem during implementation

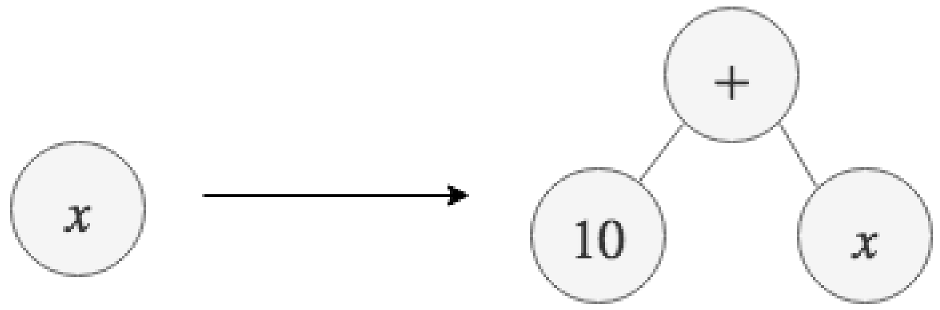
A problem we faced during implementation is the lack of *pass by reference* [17] and pointers in Java. In the *substitute* and *swap* algorithms, we would just set the node to be a copy of the rule however because Java does not support pass by reference or pointers, the method would only be changing the variable in the method's scope and not the node passed into the method, ultimately meaning that the node passed in would never change.

Our solution was to alter the design of the syntax tree, initially we had sub-classes representing a variable, operator or constant node (see appendix A). To make it work with our algorithm, we now only have a single node class and have a field to represent what type it is. We will now be able to use mutator methods to change the type of the node within the *substitute* and *swap* methods. This allowed us to alter the information of the node and not just change the reference. With the changes being made, this is what the methods look like.

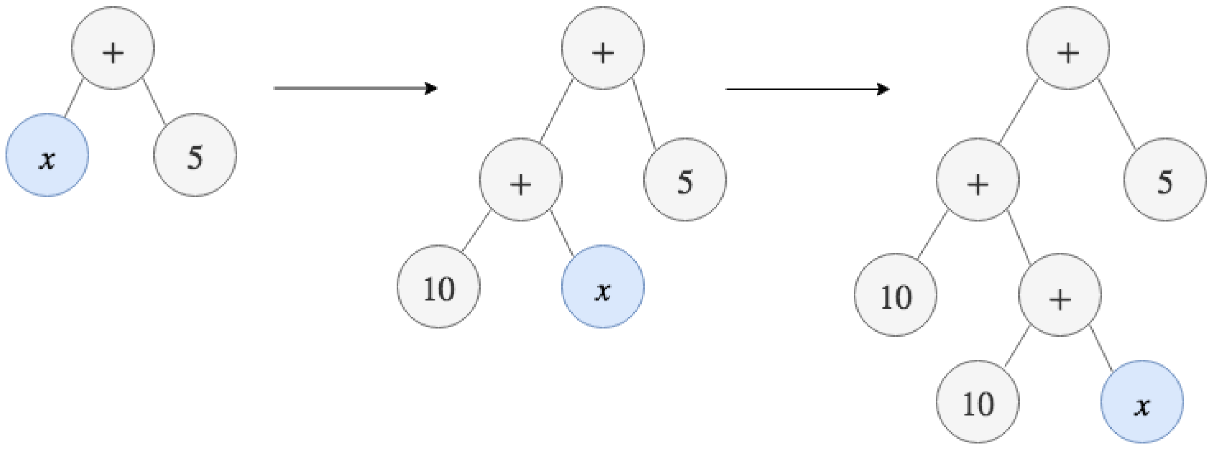
```
 private void substitute(Node term , Node rule, Map<String,Node> vars) {

    //swap the term to be the rule
    ((ASTNode)term).setValue(rule.getValue());
    ((ASTNode)term).setNodeType(rule.getNodeType());
    ((ASTNode)term).setType(rule.getType());
    ((ASTNode)term).setRight(copy(rule.getRight()));
    ((ASTNode)term).setLeft(copy(rule.getLeft()));
    //fill in variable
    swap(term,vars);

}

 private void swap(Node m,Map<String,Node> vars) {
    if(m == null)
       return;

    //check node value is a variable
    if(vars.containsKey(m.getValue())) {
       String initialValue = m.getValue();

       //set the respective node
       ((ASTNode)m).setValue(vars.get(initialValue).getValue());
       ((ASTNode)m).setNodeType(vars.get(initialValue).getNodeType());
       ((ASTNode)m).setType(vars.get(initialValue).getType());
       ((ASTNode)m).setRight(copy(vars.get(initialValue).getRight()));
       ((ASTNode)m).setLeft(copy(vars.get(initialValue).getLeft()));

    }
    swap(m.getLeft(),vars);
    swap(m.getRight(),vars);
 }
```

6.3.3 Faults discovered during testing

During testing we ran various scenarios and while doing so discovered a major problem. The problem was an infinite recursion that caused a *StackOverflowError* when traversing the tree in the *swap* method. It happens when the term provided to be rewritten contains a variable that is also declared in the signature, it could potentially replace itself with a bigger sub-tree then when traversing further it replaces itself again. Figure 6.4 below shows how the problem occur when $x \mapsto 10 + x$ and we are swapping the term $x + 5$.



(a) The mapping for variable x



(b) The process of swapping x

Figure 6.4: Problem when having variable in term to be rewritten

Our fix for this problem is to not allow any variables in the term to be rewritten, it can only contain operators and constants. Before performing the rewrite, the syntax tree will be checked by the method below. This fix imposes some limitations however due to our system's ablity to accept any symbol as constants we can easily workaround the limitation. For example if we want to rewrite the term $x+y$ but x, y are already variables in the system, we can simply replace x, y in the term with i, j and perform the rewrite.

```
private boolean checkTerm(Node n) {

  if(n == null)
    return true;
  if(n.getNodeType() == NodeType.VARIABLE)
    return false;
  if(!checkTerm(n.getLeft()))
    return false;
    return checkTerm(n.getRight());
}
```

6.4 Conclusion

We are pleased with our rewrite algorithm becuase it is completely designed from the scratch. We can use this algorithm to rewrite a term to its normal form as well as making slight modifcation to solve different problems and provide more functionalities. Although we ran into issues during implementation, we were able to come up with solutions.

7 Implementing analytical functionalities

Completion of the rewrite algorithm and being able to correctly rewrite a term allowed us to extend the functionalities by implementing various analysis capabilities. This chapter discusses the functionalities we will be implementing.

7.1 Interactive rewriting

It is often the case that multiple rewrite rules can be applied to a term, applying different rules could generate different outputs. The interactive rewriting feature would allow user to choose and apply one rule at the time and monitor the output.

To provide this function we added an additional method, *singleRewrite* to the *RewriteEngine* class. It takes a root of a syntax tree and the rule to apply. It is similar to the *search* algorithm (algorithm 7) in that it traverses the term and apply the rule on the first match as shown in the code below

```java
public boolean singleRewrite(Node t, RewriteRule r) throws RewriteException {

   if(t == null)
      return false;
   if(singleRewrite(t.getLeft(),r))
      return true;
   if(singleRewrite(t.getRight(),r))
      return true;
   Map<String,Node> vars = new HashMap<>();
   if(match(t,r.getLhs(),vars)) {
      substitute(t,r.getRhs(),vars);
      return true;
   }
   return false;
}
```

This lets the user select the rule to apply to a term however, it still have some non-determinism. For example, if a rule can be matched twice, the algorithm will not let user pick which sub-term it is applied to.

Given we are reusing most of the algorithm designed earlier on, everything works as expected and we did not encounter any issue when implementing this feature. Having the capability to apply a single rule enables us to provide much more advanced features.

7.2 Bulding a tree of all possible terms

As mentioned previously, there are many possible forms a term can be rewritten into, another analytical feature we want to provide is the ability to view all the possible forms a term can take. The way to achieve this is by building a tree of all possible forms as shown in figure 7.1. In our project we would be refering to the tree as a *search tree*.

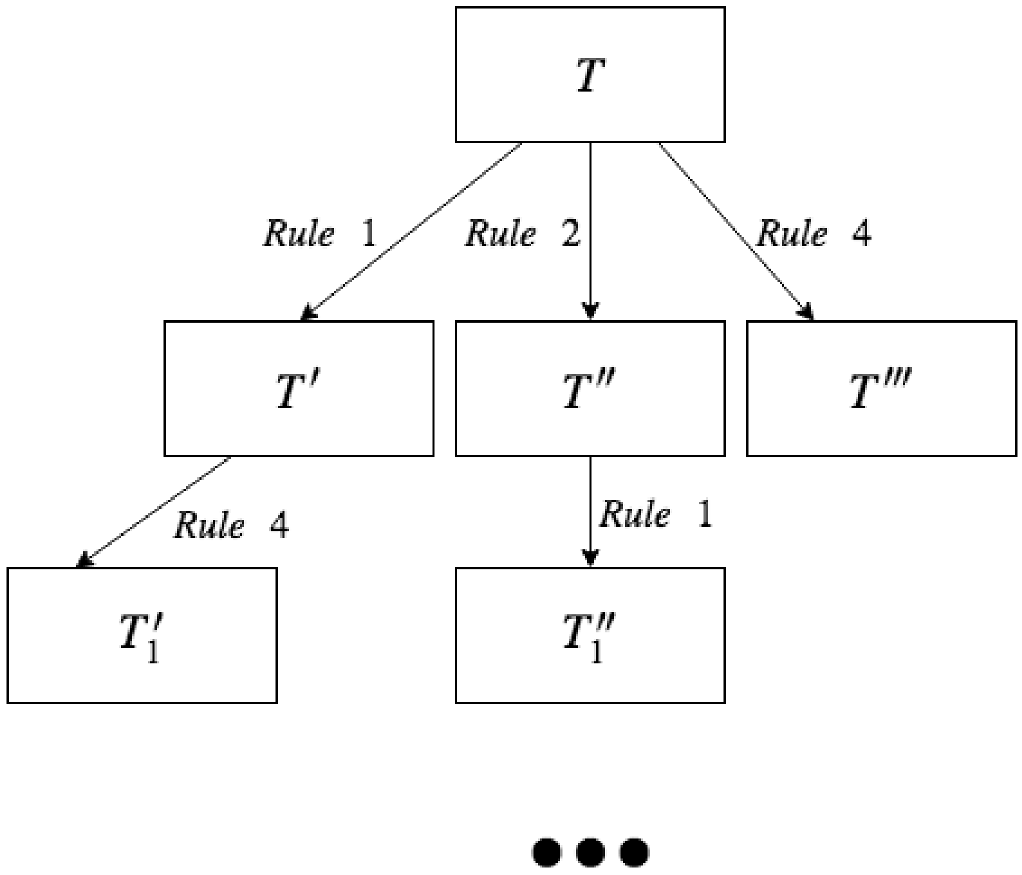


Figure 7.1: A tree containing all possible rewrite state

7.2.1 Design

Three classes are added to the rewriter package to solve this problem as demonstrated in figure 7.2. The *SearchEngine* class contains the algorithms for building the tree and searching the tree while the *SearchNode* and *SearchResult* hold the required information.

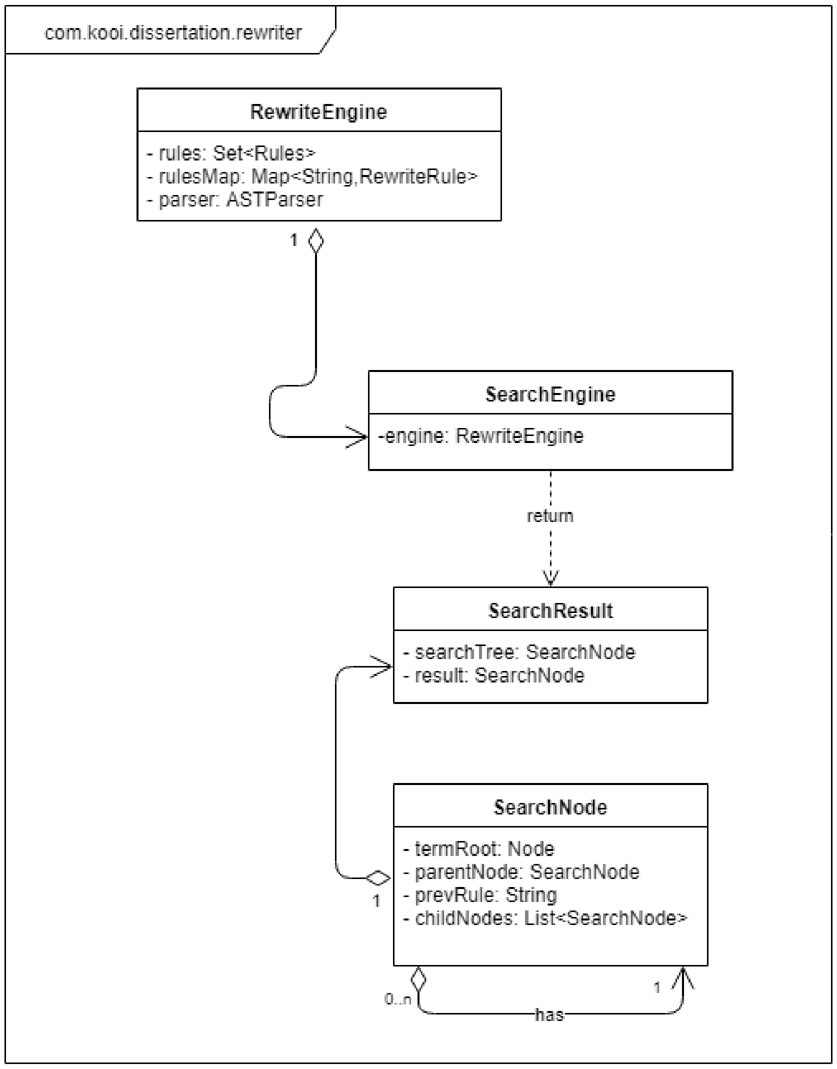


Figure 7.2: Additional classes added to the rewriter package

SearchNode represents a node in the search tree. The value it holds is the root node of the syntax tree of the term at that state. Unlike the node used in the syntax tree for a term, a *SearchNode* is not a binary tree node and can contain multiple children. It also keeps a reference to its parent and the name of the rule applied on its parent so that we can provide a trace starting from the root of the search tree, this is needed to provide a further feature.

7.2.2 Tree building algorithm

The two algorithm below, *buildUtil* and *buildSearchTree* builds a tree of all possible states given an initial term.

Algorithm 9 Building the tree

```
procedure BUILDUTIL($searchNode, rewriteEngine$)
    for $rule$ in $rewriteEngine.rules$ do
        $n \leftarrow copy(searchNode.termNode)$
        if $rewriteEngine.singleRewrite(n, rule)$ then
            $n.addChild(SearchNode(n, searchNode, rule.name))$
        end if
    end for
    for $child$ in $searchNode.childNodes$ do
        $buildUtil(child, rewriteEngine)$
    end for
end procedure
```

Algorithm 10 Build a tree given an initial term

```
procedure BUILDSEARCHTREE($initialTerm, rewriteEngine$)
    $searchRoot \leftarrow SearchNode(initialTerm, nil, \text{“”})$
    $buildUtil(searchRoot, rewriteEngine)$
    return $searchRoot$
end procedure
```

The *buildUtil* is the main algorithm for building the tree, however we need the *build-SearchTree* method to track the root. The tree is built from the top down, we create a child node to the current *SearchNode* for every rule we match starting from the root. It then recursively call itself on each child node. At the end of the algorithm, we would have a complete tree with all possible states.

7.2.3 Implementation

Similar to the implementation of the rewrite algorithm, we need to guarantee the algorithm terminates. This can be achieved by using a *bounded* search which limits the maximum depth the tree can reach.

The bound can easily be added by including extra parameters to specify the current bound and the maximum bound. When we make to recursive call down the tree we would increment the current bound by 1. Once the current bound reaches the maximum bound the method should return.

```java
public SearchNode buildSearchTree(Node initialTerm,int bound) throws
    RewriteException {
   SearchNode root = new SearchNode(engine.copy(initialTerm),null,"");
   buildUtil(root,bound,0);
   return root;
}
```

```
 private void buildUtil(SearchNode sr, int bound,int curB) throws
    RewriteException {
   if(curB > bound)
      return;
   for(RewriteRule r : engine.getRules()) {
      Node copyOfTerm = engine.copy(sr.getTermNode());
      if(engine.singleSearch(copyOfTerm, r))
         sr.addChild(new SearchNode(copyOfTerm,sr,r.getName()));
   }

   for(SearchNode c : sr.getChildNodes()) {
      buildUtil(c,bound,curB+1);
   }
}
```

7.3 Searching for state

Once we have a tree of all possible forms, we can perform a search for a particular state. Our last analytical feature is the ability to do that, it allows user to specify an initial term and a goal term to search for which the system would provide a trace to reach it if it is reachable within a bound.

SearchResult is a class created to hold the entire search tree and the reference to the *SearchNode* the user wants to search for. Since we included the reference to its parent in *SearchNode*, we do not need to store the entire trace of the search. In the scenario where the state is not reachable, the *result* field will be given *null*.

7.3.1 Tree searching algorithm

Searching for a specific term in the search tree is straightforward task, since we have a tree containing all possible value, we can perform an exhaustive search. We used a depth-first search to traverse the tree and as soon as we find a *SearchNode* whose value equals the term we are searching for we can return it. If we exceeded the bound or the entire tree is traversed without a result, the algorithm will return *null*.

Similar to the tree building algorithm, we have two methods *searchTerm* and *searchUtil*. *searchTerm* builds the term tree and call the *searchUtil* which traverse the tree. The pseudocode below shows the two methods.

Algorithm 11 Searching the tree

```
procedure SEARCHUTIL($searchNode, term, bound, currentBound$)
    if $currentBound > bound$ then
        return $nil$
    end if
    if $searchNode.value = term$ then
        return $searchNode$
    end if
    for $child$ in $searchNode.childNodes$ do
        $next \leftarrow searchUtil(child, term, bound, currentBound + 1)$
        if $next \neq nil$ then
            return $next$
        end if
    end for
    return $nil$
end procedure
```

Algorithm 12 Search a state given initial term

```
procedure SEARCHTERM($initialTerm, term, bound, rewriteEngine$)
    $searchRoot \leftarrow buildSearchTree(initialTerm, bound, rewriteEngine)$
    $result \leftarrow searchUtil(searchRoot, term, bound, 0)$
    return $searchResut(searchRoot, result)$
end procedure
```

7.3.2 Building the trace

Having the target node we can easily build a textual trace by going up to the parent nodes until we reach the root. We implemented it using Java's *StringBuilder* and Stack, we pushed all the nodes onto the stack then pop an element one at a time and append the information to the StringBuilder.

```java
private String buildResultText(SearchResult result) throws ParseException {

    StringBuilder trace = new StringBuilder();

    Stack<SearchNode> nodes = new Stack<>();
    SearchNode res = result.getResult();

    while(res.getParentNode() != null) {
      nodes.push(res);
      res = res.getParentNode();
    }

    while(!nodes.isEmpty()) {
        SearchNode curr = nodes.pop();
        trace.append("\=>Apply "+curr.getPrevRule()+" to get
            "+home.getParser().toInfix(
              home.getParser().postOrderTreverse(curr.getTermNode())));
        trace.append("\n");
    }
    return trace.toString();
}
```

7.4 Conclusion

Overall we are satisfied with the three analytical funtionalities, they are simple to use while also being able to provide more insight on term rewriting to the user.

8 Implementing the graphical user interface

This chapter focuses on designing and implementation of the graphical interface which is critical in our goal of providing ease of use to the user.

8.1 Design

The UI design process includes identifying what the interface needs and providing a sketch of the the graphical interface [1]. A good design of the user interface has to provide all the required functionalities while also being intuitive. We start the design process by drawing up a user flow diagram.

8.1.1 User Flow Diagram

The user flow diagram helps us undertand two things, all the actions that can be performed at each state of the system and the series of steps needed to achieve a specific goal.

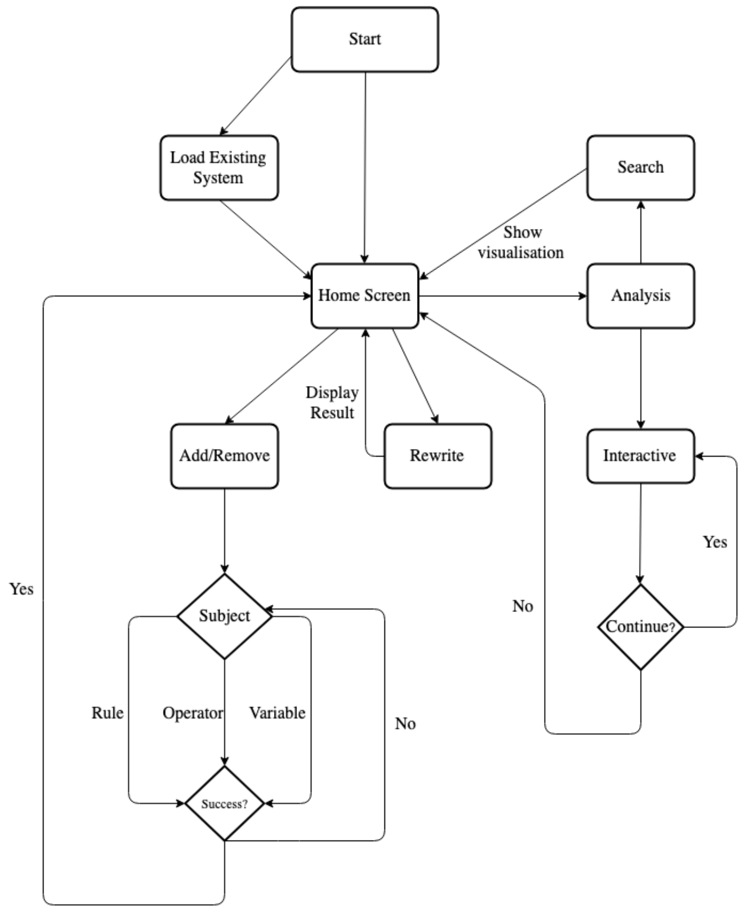


Figure 8.1: User flow diagram

As shown in the diagram above, our system has a main point of interaction which is the ***home screen***. Various actions can be performed by the user from the home screen and upon completion the user is taken back to the home screen.

Having all actions being accessable from the home screen minimise the need for users to remember how to access each feature of the system. This design rationale is one of *10 usability heuristics for user interface design* [19], recognition is better than recall.

8.1.2 UI mockups

Using the user flow diagram that identifies the actions the users can perform, we can start sketching visual representation of the windows/dialog that performs those actions. We used a technique called *Wireframe* [9], it lays out the content and functionalities of every key window/dialog needed. This approach was chosen as it is the quickest way of providing visual mockup of an interface, although wireframing lacks graphical details such as colours or typesetting it provides enough information where we can begin implementation.

Homescreen mockup The homescreen should provide relevant information of the system such as the signature and the rules. This allows users to make better decision, such as making any changes to the system if needed [19]. The figure below is the wireframe mockup of the homescreen.

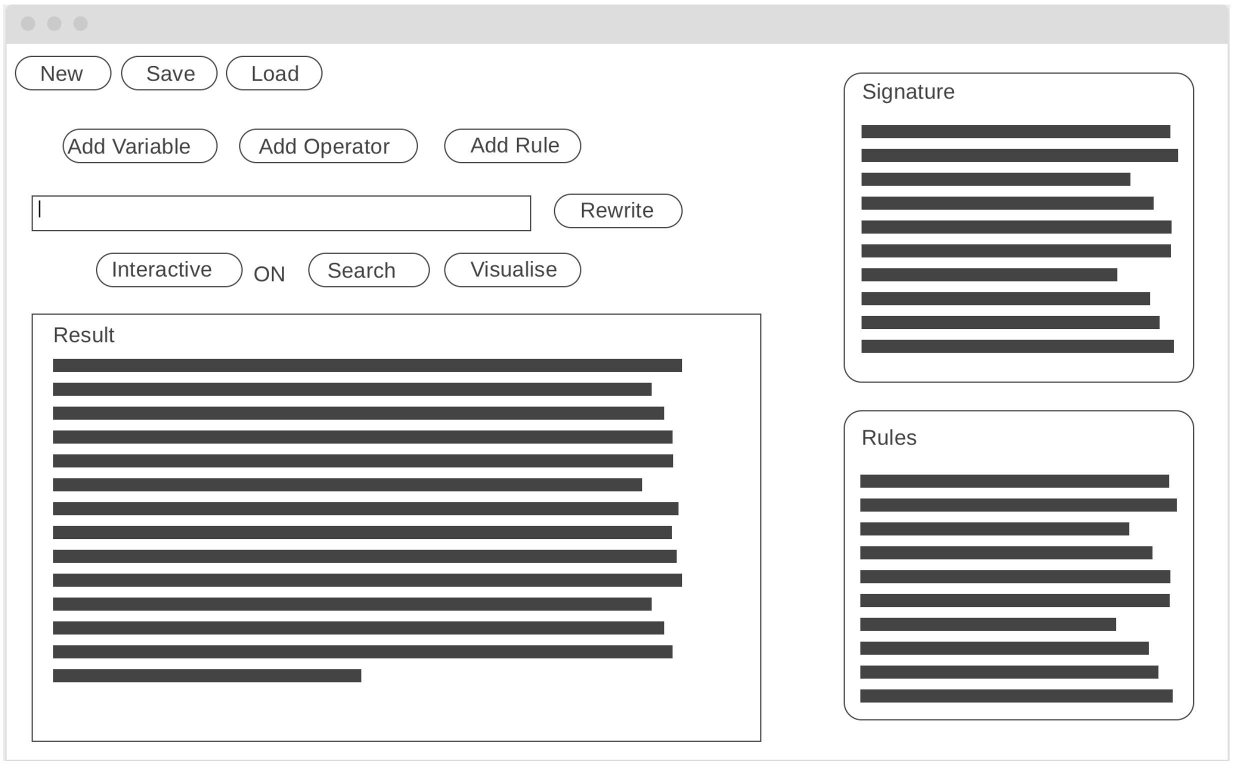


Figure 8.2: Home screen design

Pop-up window As we want all actions to be performed on the home screen, user would mainly interact with the system via pop-up windows. For example, when the user presses the button to add a rule, a pop-up would appear asking users for the required information.

The pop up window is minimalist and does not contain irrelevant information to improve the visibility. The mockup below shows the design of the “Add Rule window”. We only provide one example below as the other pop-up follows a similar design to provide consistency.

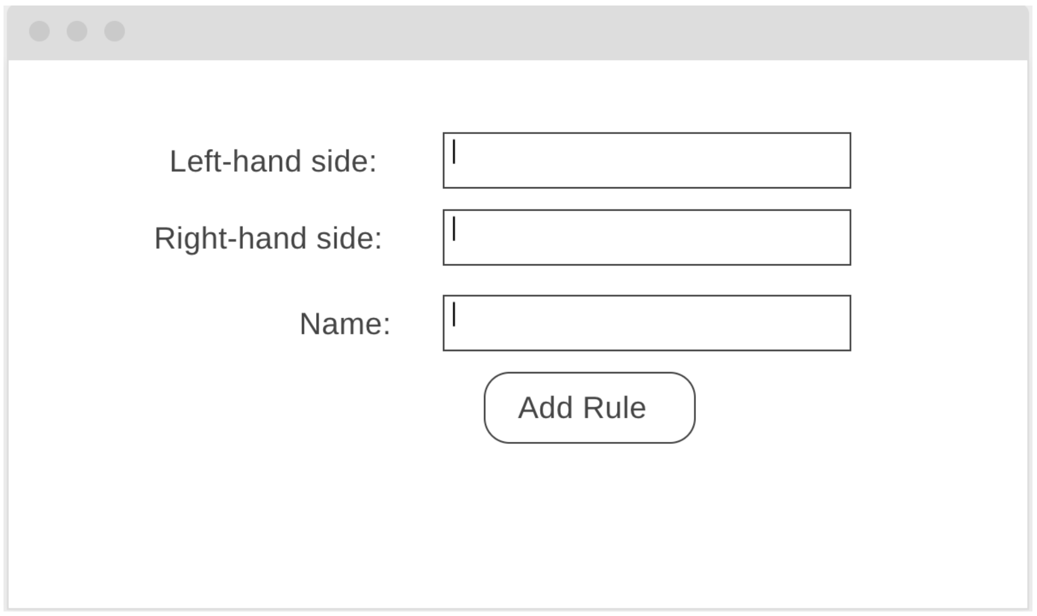


Figure 8.3: Pop-up window design

8.1.3 Usability of the design

We mentioned in earlier that we followed the *10 usability heuristics for user interface design* [19] in our design, this section will talk about a few more decisions we have made to adhere to the guidelines.

Flexibility and efficiency of use We made sure to include the ability to save and reload frequently used signatures and rules to provide regular users the efficiency.

User control and freedom Users have the abilty to remove operators and variables from the signature as well as rewrite rules. This allows user to quickly recover from any mistakes without having to go through the entire process again.

Error prevention and recovery Our interface performs error checking on user inputs before handing the data off to the system, this saves our system from failing. Users would be notified of any errors in non-technical terms so that they can perform the required diagnostic.

8.2 Implementation

Our implementation is carried out using pure Java Swing layout and did not use any interface building tool. The Java Swing framework operates using an *Event-driven architecture* [15] which is an important concept when implementing GUI.

8.2.1 Implementing the home screen

The home screen is the main screen of the system and as such holds all the important data, like the *ASTParser* and *RewriteEngine*. The home screen is responsible for handling the events of performing rewrite and displaying the results as well as opening any pop-up window while passing them the required objects.

Adding data into the system As previously mentioned, the home screen will call the respective pop-up window to handle adding new information to the system. The home screen will need to provide access to its *ASTParser* and *RewriteEngine* so other classes can modify it. Once the pop-up window have access to it, adding to it is straightforward. The following code is taken from the *AddRule* window which shows the process of taking inputs from the *JTextFields* to make rewrite rule and adding it into the system.

```
addBtn.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {

       //error handling
      if(lhsTextField.getText().equals("") ||
         rhsTextField.getText().equals("") ||
         nameField.getText().equals("")) {
         JOptionPane.showMessageDialog(AddRule.this, "Fields cannot be
            empty.","Missing values",JOptionPane.ERROR_MESSAGE);
      }else {

         RewriteRule rule =
            ruleFactory.getRewriteRule(lhsTextField.getText(),
            rhsTextField.getText(), nameField.getText());

         if(rule == null) {
            JOptionPane.showMessageDialog(AddRule.this, "An error is
               encountered during parsing, check for mismatch
               parenthesis.","Parsing Exception",JOptionPane.ERROR_MESSAGE);
         }else {
            home.getEngine().addRule(rule);
            home.updateUI();
            AddRule.this.dispose();
         }
      }
   }
});
```

Updating UI with changes in system It is important for the home screen to update when the user makes any changes to the system. We achieved this by splitting the components of the home screen into panels, when a change is made, we call the *updateUI* method which clears the panel and repaint it with the new data. The code snippet below shows our method (some details are ommited to improve legibility).

```
  public void updateUI() {
     opPanel.removeAll();
     opPanel.revalidate();
     opPanel.repaint();
     varPanel.removeAll();
     varPanel.revalidate();
     varPanel.repaint();
     rulePane.removeAll();
     rulePane.revalidate();
     rulePane.repaint();

     //adding the operators
     for(String op : sig.getOperatorSet()) {
       JLabel l = new JLabel(op);
       opPanel.add(op);
     }

     //adding the variables
     JLabel varLabel = new JLabel("Variables:");
     varPanel.add(varLabel);
     for(String id: sig.getVariableSet()) {
      JLabel l = new JLabel(id+":"+sig.getVariable(id));
      varPanel.add(l);
     }

     //adding the rules
     for(RewriteRule r : engine.getRules()) {
       JLabel l = new JLabel(toInfix(r.getLhs())+"->"+toInfix(r.getRhs()));
       rulePane.add(l);
     }
  }
```

Rewriting and displaying textual result The main text field *rwField* handles all rewriting funtionalities including rewriting, interactive rewriting and visualisation. When the rewrite button is pressed, the system will parse the term in this text field and perform the rewrite, the result is displayed in *resArea*, a non-editable *JTextArea*. Other components can set the text in *resArea* by using the *setTextArea* method in the home screen class.

The textual trace is built by iterating through the *RewriteSteps* and extract the relevant information as a string and appended to a *StringBuilder* which will be displayed in *resArea*.

```
    rwButton.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        try {
              if(rwField.getText().equals("")) {
```

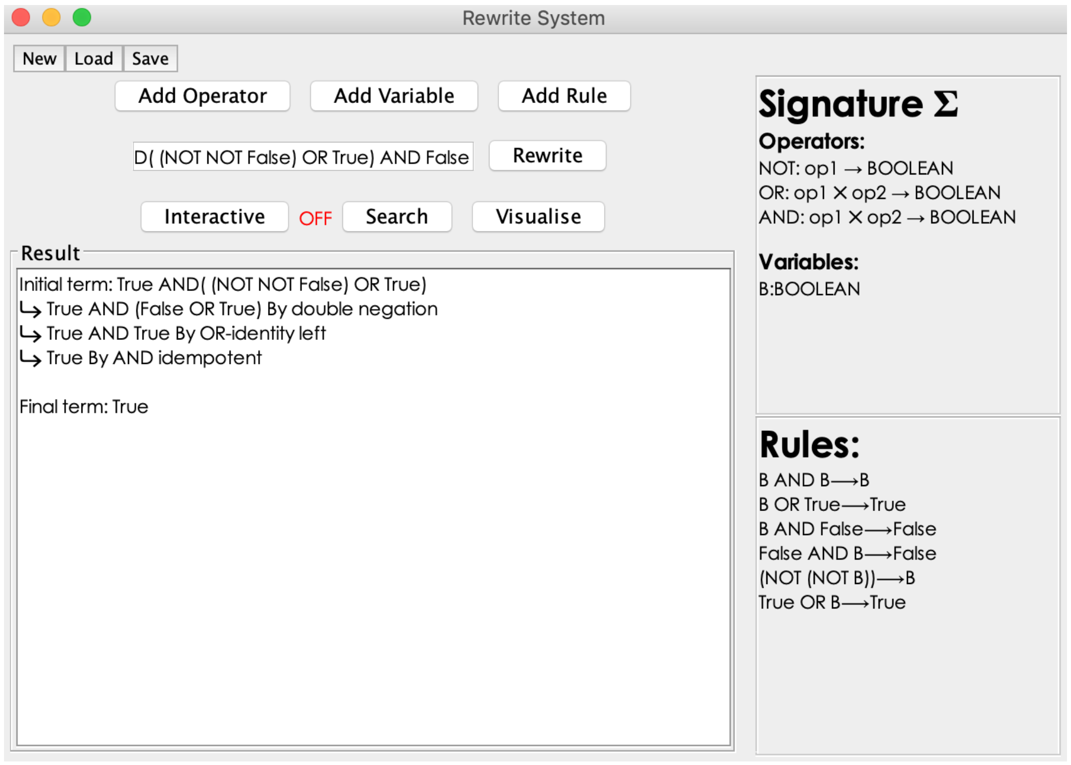
```
              //display error message
             }else {
                RewriteResult o = rw.rewrite(rwField.getText());
                currResult = new StringBuilder();
                currResult.append("Initial term: "+rwField.getText()+"\n");

                for(RewriteStep step : o.getListOfSteps()) {

                 currResult.append("=> "+
                        home.getParser().toInfix(
                        home.getParser().postOrderTreverse(step.getTermRoot())));

                 currResult.append(" By "+step.getRule());
                 currResult.append("\n");
                }
               //the final form of the term
               currResult.append("\nFinal term: "+
                       home.getParser().toInfix(
                       home.getParser().postOrderTreverse(o.getFinalTerm())));
               resArea.setText(currResult.toString());
         }catch(Exception e){
            //display error message
         }
   }
}
```

Implementing the interactive rewrite Implementing the interactive rewrite feature is slightly more involved as we would need a way to keep track of the current state. Our solution is to hold a *Node* in the *HomeScreen* class to represent the state, if interactive mode is off, then this node is *null*, this node is named the *interactiveRoot*.

When the user presses the button to initiate interactive rewrite, the system first check if it is in interactive mode, if it is then it will carry on using the *interactiveRoot*, if it is not then it goes into interactive mode by clearing the result area and setting the *interactiveRoot* with the term in *rwField*. A pop-up window will appear to let users select the rule they want to apply and the output will be shown in the result area. When the user exits interactive mode, the *interactiveRoot* is set to *null*.

```
    public void actionPerformed(ActionEvent e) {
      //create a selector of all possible rules
      Object[] rules = rw.getRules().toArray();
      RewriteRule r = (RewriteRule)JOptionPane.showInputDialog(home,
                                  "Choose a rule to apply\n",
                                  "Choose rule to apply",
                                  JOptionPane.PLAIN_MESSAGE,
                                   null,
                                rules,
                                "");

    if(r != null) {
      //if first interactive run, clear resArea and set on
      if(interactiveRoot == null) {
        try {
          interactiveRoot = home.getParser().parseAST(rwField.getText());
          intStat.setText("ON");
          intStat.setForeground(Color.GREEN);
          resArea.setText("");
        } catch (ParseException e1) {
          //display error message
        }
      }
      try{
        if(rw.singleSearch(interactiveRoot,r))
          resArea.append("=> "+
                  home.getParser().toInfix(
                  home.getParser().postOrderTreverse(interactiveRoot))+
                  " By "+r.getName()+"\n");
        else
          resArea.append("=> "+
                  home.getParser().toInfix(
                  home.getParser().postOrderTreverse(interactiveRoot))+
                  " Rule does not apply!!"+"\n");
      } catch (Exception e1) {
        //display error message
      }
    }
  }
```

In addition to the implementation above, we provided a few aesthetic enhancements to the system such as changing the fonts and using Unicode strings for the mathematical symbol. The screenshot below is the graphical user interface fully implemented (All screenshots are in appendix D).



D((NOT NOT False) OR True) AND False

Result

```
Initial term: True AND( (NOT NOT False) OR True)
↳ True AND (False OR True) By double negation
↳ True AND True By OR-identity left
↳ True By AND idempotent

Final term: True
```

Signature Σ

Operators:
NOT: op1 → BOOLEAN
OR: op1 ✕ op2 → BOOLEAN
AND: op1 ✕ op2 → BOOLEAN

Variables:
B:BOOLEAN

Rules:
B AND B⟶B
B OR True⟶True
B AND False⟶False
False AND B⟶False
(NOT (NOT B))⟶B
True OR B⟶True

Figure 8.4: Home screen implemented

8.2.2 Implementing the graphical visualisation

The graphical tree visualiser was implemented using the JGraph library, which is based on Java Swing. The JGraph's *mxGraphComponent* extends Java Swing's *JScrollPane* and is responsible for drawing the graph.

Components in the graph Before we can dive into the details of the implementation we need to understand some of JGraph's components. *mxGraph* is the most important component as it represents the graph that will be displayed by the *mxGraphComponent*. It contains *mxICell* which are the vertices of the graph and also edges between vertices which can be added using methods provided by *mxGraph*. In our implementation, our *mxGraph* component is named **graph** and we would be using this name in the code snippets in later sections. *mxGraph* can have different *layouts* which are different ways to arrange the *mxICell* in the graph. Since we want to build a tree we will be using JGraph's *mxHierarchicalLayout* which arranges the *mxICell* from top to bottom.

Building the graph We build our graph by making a one to one copy of our search tree using an algorithm very similar to the algorithm we used to build our search tree (section 7.2.2). We start from the root and insert a *mxICell* to represent it, we would then start adding a cell and edge for each of the child nodes in the search tree then recursively do the same for all the children until we reached the leaf nodes. The code below is the implementation of the algorithm.

```
private void buildGraph() throws ParseException {
   mxICell graphRoot = (mxICell) graph.insertVertex(parent, null,
       home.getParser().toInfix(home.getParser().postOrderTreverse(root.getTermNode())),
       250, 250, 80, 30);

   graph.updateCellSize(graphRoot);

   buildGraphUtil(root,graphRoot,1);

}

private void buildGraphUtil(SearchNode sNode, mxICell currentCell,int depth)
    throws ParseException {
   for(SearchNode n : sNode.getChildNodes()) {

      mxICell child = (mxICell) graph.insertVertex(parent, null,
         home.getParser().toInfix(home.getParser().postOrderTreverse(n.getTermNode())),
         0, 0, 80, 30);
      mapping.put(child, n);


      graph.insertEdge(parent, "", n.getPrevRule(), currentCell, child);
      graph.updateCellSize(child);
      buildGraphUtil(n,child,depth+1);
   }
}
```

After running the *buildGraph* method, ***graph*** will be filled with the vertices and edges that make up the search tree, it can then wrap in a *JFrame* and displayed in a separate window.

8.3 Conclusion

Implementation of the graphical user interface marks the end of the developement for our system. We were satisfied with our ability to implement a user interface with good usability by having a simple and intuitive yet good looking user interface.

9 Testing

In this section we would be discussing the testing process during and after development. Whilst most of the testing are done within each iteration we have summarised the results into one chapter.

9.1 Unit testing

As we are following a Test-Driven Development approach unit tests are key. These are tests that are written before any implementation are done and they are designed to test a small part of the system.

9.1.1 Test cases

There are three particular components that are extensively unit tested, the syntax tree, parser and rewrite engine, all test cases are included in appendix G.

In the syntax tree package we need to test that we are able to determine if tree nodes are equal and compare precedence of operators. We tested the tree nodes' comparison by creating multiple tree nodes objects with same values and different values. We compared them and ensure that they yielded the expected result. A similar approach is used when comapring operators' precedence, which is creating multiple operators, both unary and binary then comapared them against each other and validate the results.

As for the parser we need to test the tokenization of a string and the parsing, we created two different parsers with different signatures to generate a total of 53 test cases which involve testing the tokenizing method, parsing to RPN and AST. Checking that our syntax contains the correct values is achieved by doing a post-order traversal and comparing the values with the RPN values, a correct syntax tree should produce a RPN if it is traverse in post-order traversal [6]. We made sure to include multiple cases with syntax errors to make sure the parsers are able to detect them.

Finally for the rewrite engine we are looking at testing the full and interactive rewrite method. Similar to the parser's unit tests, we performed our tests on two different signatures to get a wider coverage and also included error cases. The output is checked by comparing the syntax tree after rewriting with a syntax tree we expect.

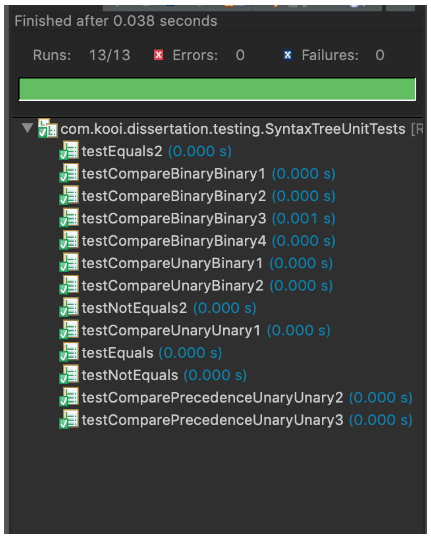
9.1.2 Test data

We used a variety of test data for our unit tests, we separated our data into two category, simple and complex. Having these two category we would randomly generate inputs that we fit into the respective category. The random test data generation method is the simplest way to generate test data hence the reason we chose it.

As the functions we are testing can easily be computed manually, it allowed us to easily generate expected result using pen and paper. The calculated result can then be compared with the output generated by the system in our unit tests which is going to tell us if it is working correctly.

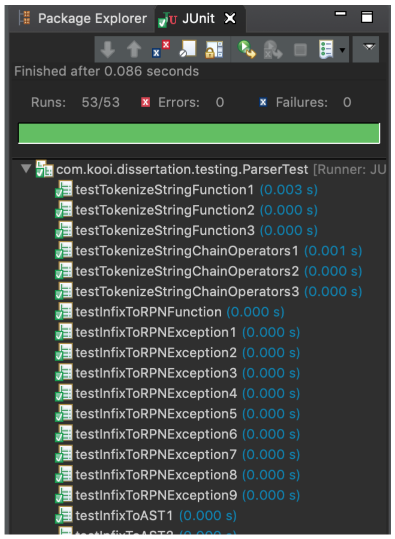
9.1.3 Test result

The figure belows shows the test result of all three unit test suite and shows all that all test passed with no issues. Although unit test is just the first step in testing, it gives us confidence in our system going into the other tests.



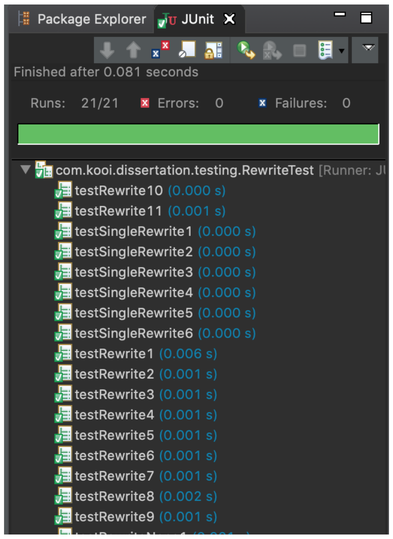
```
Finished after 0.038 seconds
Runs: 13/13   Errors: 0   Failures: 0
▼ com.kooi.dissertation.testing.SyntaxTreeUnitTests [R
    testEquals2 (0.000 s)
    testCompareBinaryBinary1 (0.000 s)
    testCompareBinaryBinary2 (0.000 s)
    testCompareBinaryBinary3 (0.001 s)
    testCompareBinaryBinary4 (0.000 s)
    testCompareUnaryBinary1 (0.000 s)
    testCompareUnaryBinary2 (0.000 s)
    testNotEquals2 (0.000 s)
    testCompareUnaryUnary1 (0.000 s)
    testEquals (0.000 s)
    testNotEquals (0.000 s)
    testComparePrecedenceUnaryUnary2 (0.000 s)
    testComparePrecedenceUnaryUnary3 (0.000 s)
```

(a) Syntax Tree package's unit tests



```
Package Explorer   JUnit
Finished after 0.086 seconds
Runs: 53/53   Errors: 0   Failures: 0
▼ com.kooi.dissertation.testing.ParserTest [Runner: JU
    testTokenizeStringFunction1 (0.003 s)
    testTokenizeStringFunction2 (0.000 s)
    testTokenizeStringFunction3 (0.000 s)
    testTokenizeStringChainOperators1 (0.001 s)
    testTokenizeStringChainOperators2 (0.000 s)
    testTokenizeStringChainOperators3 (0.000 s)
    testInfixToRPNFunction (0.000 s)
    testInfixToRPNException1 (0.000 s)
    testInfixToRPNException2 (0.000 s)
    testInfixToRPNException3 (0.000 s)
    testInfixToRPNException4 (0.000 s)
    testInfixToRPNException5 (0.000 s)
    testInfixToRPNException6 (0.000 s)
    testInfixToRPNException7 (0.000 s)
    testInfixToRPNException8 (0.000 s)
    testInfixToRPNException9 (0.000 s)
    testInfixToAST1 (0.000 s)
```

(b) Parser package's unit tests



```
Package Explorer   JUnit
Finished after 0.081 seconds
Runs: 21/21   Errors: 0   Failures: 0
▼ com.kooi.dissertation.testing.RewriteTest [Runner: JU
    testRewrite10 (0.000 s)
    testRewrite11 (0.001 s)
    testSingleRewrite1 (0.000 s)
    testSingleRewrite2 (0.000 s)
    testSingleRewrite3 (0.000 s)
    testSingleRewrite4 (0.000 s)
    testSingleRewrite5 (0.000 s)
    testSingleRewrite6 (0.000 s)
    testRewrite1 (0.006 s)
    testRewrite2 (0.001 s)
    testRewrite3 (0.001 s)
    testRewrite4 (0.001 s)
    testRewrite5 (0.001 s)
    testRewrite6 (0.001 s)
    testRewrite7 (0.001 s)
    testRewrite8 (0.002 s)
    testRewrite9 (0.001 s)
```

(c) Rewrite engine package's unit tests

Figure 9.1: Unit tests result

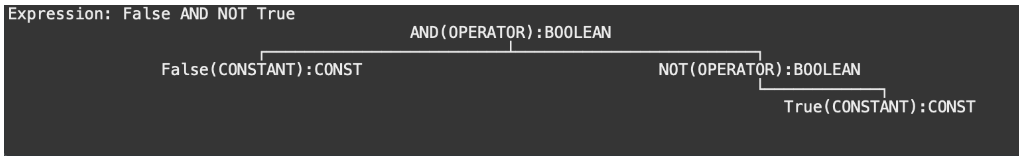
9.2 Integration testing

Unit testing is efficient as we can run them easily when we make any changes and although they are often reliable when testing individual part of the system, we have to make sure that these parts are able to correctly work with each other and we do that using *integration testing* [31]. Our integration testing is ran using a driver program which interacts with the components and prints out the output on the command line for us to analyse.

9.2.1 Integration testing the parser

We used a method that can print a binary tree out on the command line which we would then try parsing different terms and visualise the output. We carried out a number of test runs until we gain a high level of confidence on our parser. The code below shows how we set the test and the screenshot shows the output for a test run.

```java
static void parsingTest() {
      Set<Operator> opsB = new HashSet<>();
      opsB.add(new BinaryOperator("AND",2,DataType.BOOLEAN));
      opsB.add(new BinaryOperator("OR",2,DataType.BOOLEAN));
      opsB.add(new UnaryOperator("NOT",1,DataType.BOOLEAN));

      HashMap<String,DataType> variablesB = new HashMap<>();
      variablesB.put("B",DataType.BOOLEAN);

      Signature cBool = new Signature(opsB,variablesB);
      ASTParser boolP = new ASTParser(cBool);

      try {
         Node n = boolP.parseAST("False AND NOT True");
         prettyPrint(n); //method to print tree
      } catch (ParseException e) {
         e.printStackTrace();
      }
}
```



```
Expression: False AND NOT True
                                AND(OPERATOR):BOOLEAN
             ┌──────────────────────┴──────────────────────┐
     False(CONSTANT):CONST                        NOT(OPERATOR):BOOLEAN
                                                          └─────────┐
                                                          True(CONSTANT):CONST
```

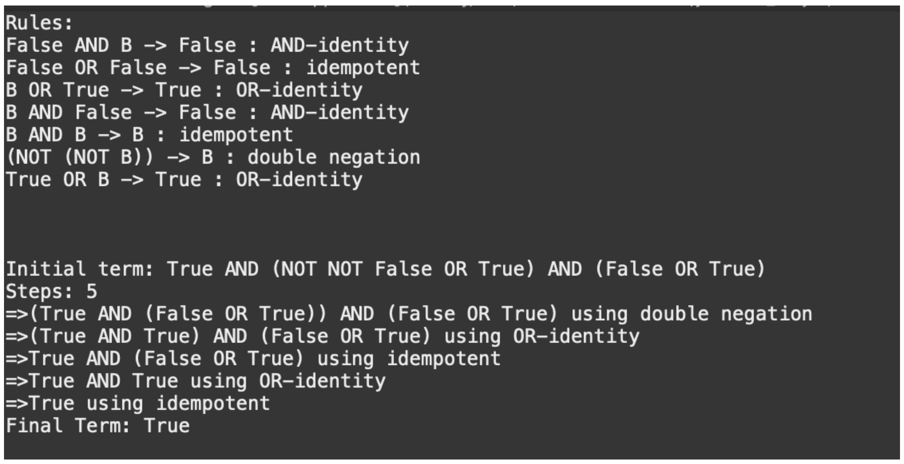
Figure 9.2: AST for expression False AND NOT True

9.2.2 Integration testing the rewrite engine

Integration testing becomes more useful when testing our rewrite engine as we can verify that all intermidiate stages are correct which would be hard and tedious to perform using unit tests. The code snippet below shows how we set up the test and screenshot below is an example of a test run.

```java
static void booleanTest() {
      //...set up parser, see section 9.2.1
      RewriteRuleFactory fB = new RewriteRuleFactory(boolP);
      Set<RewriteRule> rulesB = new HashSet<>();
```

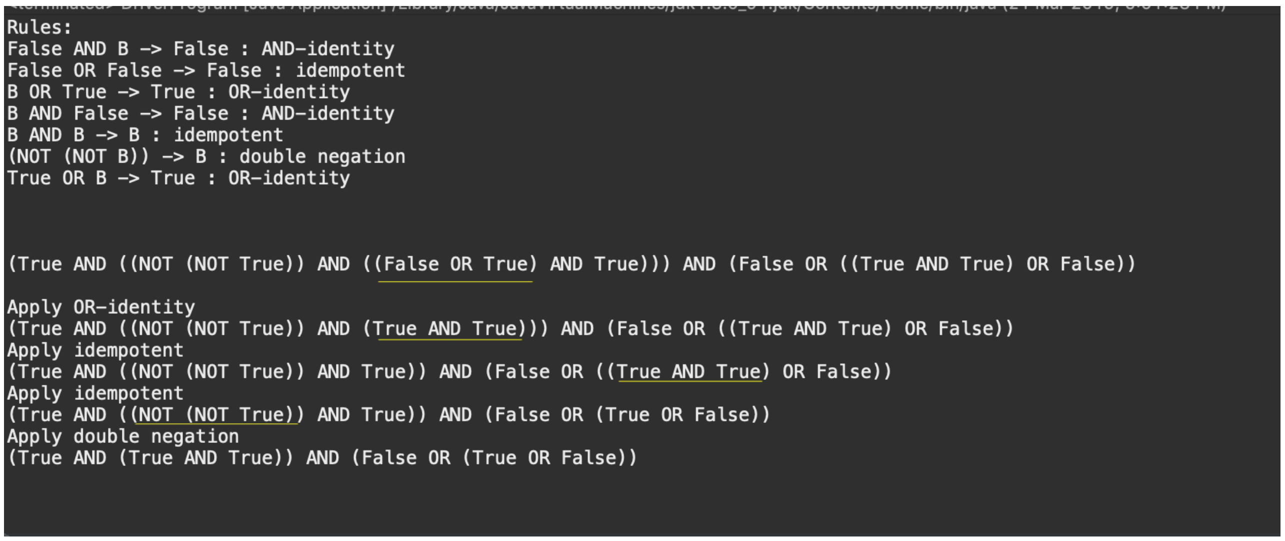
```
    rulesB.add(fB.getRewriteRule("NOT NOT B", "B", "double negation"));
    rulesB.add(fB.getRewriteRule("B AND B", "B", "idempotent"));
    rulesB.add(fB.getRewriteRule("True OR B", "True", "OR-identity"));
    rulesB.add(fB.getRewriteRule("B OR True", "True", "OR-identity"));
    rulesB.add(fB.getRewriteRule("False OR False", "False", "idempotent"));
    rulesB.add(fB.getRewriteRule("B AND False", "False", "AND-identity"));
    rulesB.add(fB.getRewriteRule("False AND B", "False", "AND-identity"));

    RewriteEngine r = new RewriteEngine(rulesB,boolP);
    try {
       System.out.println("Rules: ");
       for(RewriteRule rule : rulesB) {
          System.out.println(boolP.toInfix(
                      boolP.postOrderTreverse(
                      rule.getLhs()))+" -> "+boolP.toInfix(
                      boolP.postOrderTreverse(rule.getRhs()))+" :
                         "+rule.getName());
       }
       System.out.println("\n\n");


       RewriteResult res = r.rewrite("True AND (NOT NOT False OR True) AND
           (False OR True)");

       System.out.println("Initial term: "+res.getInitialTerm());
       System.out.println("Steps: "+res.getListOfSteps().size());
       for(RewriteStep step : res.getListOfSteps()) {
          System.out.println("=>"+boolP.toInfix(
                      boolP.postOrderTreverse(step.getTermRoot()))+" using
                         "+step.getRule());
       }
       System.out.println("Final Term:
           "+boolP.toInfix(boolP.postOrderTreverse(res.getFinalTerm())));

    } catch (ParseException | RewriteException e) {
       e.printStackTrace();
    }
}
```



```
Rules:
False AND B -> False : AND-identity
False OR False -> False : idempotent
B OR True -> True : OR-identity
B AND False -> False : AND-identity
B AND B -> B : idempotent
(NOT (NOT B)) -> B : double negation
True OR B -> True : OR-identity



Initial term: True AND (NOT NOT False OR True) AND (False OR True)
Steps: 5
=>(True AND (False OR True)) AND (False OR True) using double negation
=>(True AND True) AND (False OR True) using OR-identity
=>True AND (False OR True) using idempotent
=>True AND True using OR-identity
=>True using idempotent
Final Term: True
```

Figure 9.3: A test run for rewriter

In addition to the regular rewrite we also tested the interactive rewrite funtionality using the driver program. The screenshot below is a test run we performed with the term underlined being the one we attempt to rewrite.



```
Rules:
False AND B -> False : AND-identity
False OR False -> False : idempotent
B OR True -> True : OR-identity
B AND False -> False : AND-identity
B AND B -> B : idempotent
(NOT (NOT B)) -> B : double negation
True OR B -> True : OR-identity



(True AND ((NOT (NOT True)) AND ((False OR True) AND True))) AND (False OR ((True AND True) OR False))

Apply OR-identity
(True AND ((NOT (NOT True)) AND (True AND True))) AND (False OR ((True AND True) OR False))
Apply idempotent
(True AND ((NOT (NOT True)) AND True)) AND (False OR ((True AND True) OR False))
Apply idempotent
(True AND ((NOT (NOT True)) AND True)) AND (False OR (True OR False))
Apply double negation
(True AND (True AND True)) AND (False OR (True OR False))
```

Figure 9.4: Test run applying individual rewrite rule

9.3 Graphical User Interface testing

One of the down side of our integration testing approach is that it is hard to visualise the system especially when it comes to trees. As such some components of the system such as the visualisation can only be done through the graphical user interface.

9.3.1 Testing building search tree

Our algorithm for building the search tree is tested after we implemented the visualisation tool. We tried a variety of data and analysed the tree by going through every node and checked that the rules are applied correctly. The screenshot below is an example of a search tree viewed using the visualisation tool.

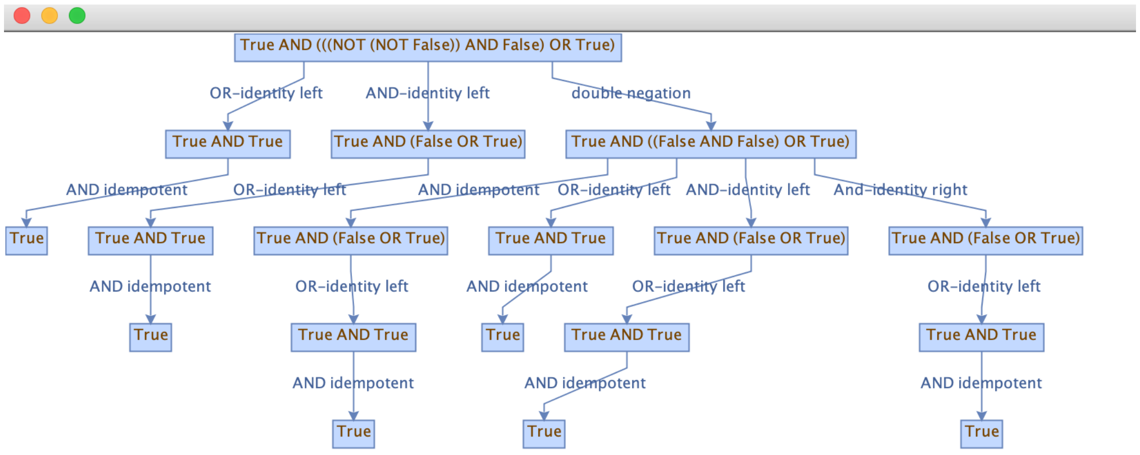
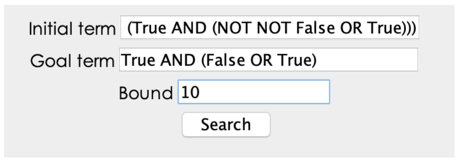


Figure 9.5: Search tree visualisation

9.3.2 Testing the search functionality

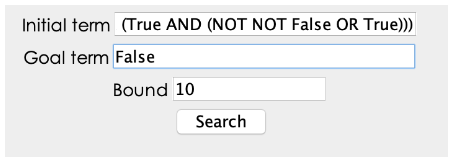
We tested the search functionality through the graphical user interface as it was dependant on completion of the search tree which meant at this stage the GUI was completed. This was an advantage as it minimises the number of extra code we would need to write if we were doing it through the command line.

As usual, we generated we variety of random data across different signatures and analysed the output. We tested a variety of cases where the term is reachable and also not reachable. Figure 9.6 and 9.7 shows two test runs for a reachable and unreachable term.



Initial term (True AND (NOT NOT False OR True))
Goal term True AND (False OR True)
Bound 10

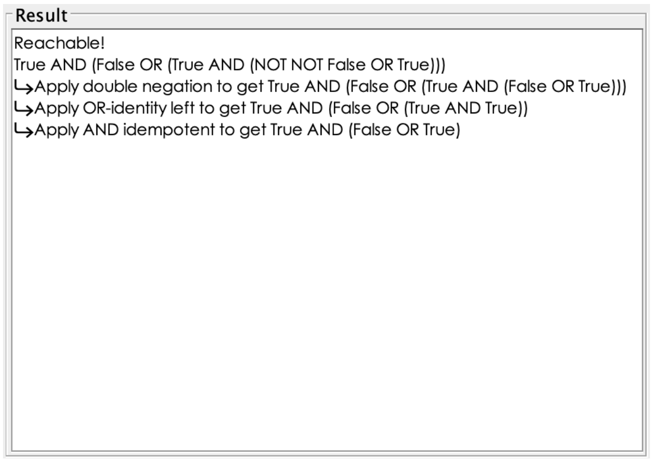
(a) Search test case 1



Initial term (True AND (NOT NOT False OR True))
Goal term False
Bound 10

(b) Search test case 2

Figure 9.6: Test cases for search functionality including reachable and not reachable



```
Reachable!
True AND (False OR (True AND (NOT NOT False OR True)))
↳Apply double negation to get True AND (False OR (True AND (False OR True)))
↳Apply OR-identity left to get True AND (False OR (True AND True))
↳Apply AND idempotent to get True AND (False OR True)
```

(a) Result for test case 1

```
Term is not reachable
```

(b) Result for test case 2

Figure 9.7: Test results for the search functionality

9.4 Analysis of testing

The goal of testing is to locate any faults in the system [11] and given the complexity of our system we are expected to have bugs. The list below are some of the bugs that we found during testing and how we fixed them.

Tokenizing including whitespace When we first implemented the tokenizing method we did not account for user putting whitespace in the term. This lead to "$x + \quad y$" to be tokenize into ["x", "$+$", " y"] which would led to the generation of an invalid sytax tree. The fix was to strip all whitespaces out of the string before we start tokenizing.

Binary operator only having one operand In our parsing algorithm we overlooked the case where users entered a malformed term where a binary operator only have one operand. This lead to either the system crashing as there are no operand to pop off the stack or the parser generating a meaningless syntax tree. The detailed fix was covered in section 5.3.3.

Infinite recursion when swapping variables There was a major bug in our rewrite algorithm that was caught at a very late stage. This bug occured when the term to rewrite has a variable and opens the door to cases where a variable in a rule would map to term that contains itself and potentially go into an infinite recursion when swapping. An in depth explanation and fix was covered in section 6.3.3.

Graphical Visualiser not showing the full term We had a problem in our graphical visualiser where the term would get cut off in the node. The problem is that the node is built first and there would be cases where the string for the term is bigger than the node hence it won't show the entire term. The fix for this is straightforward as the *mxGraph* component has a method call *updateCellSize* that would change the size of a cell to accomodate its content.

The bugs above are only a subset of what we caught during testing, they are all fixed and eventually reached a point where all our test passed and we are satisfied with the final system. It is important to note that “testing can only show the presence of bugs, not its absence” as stated by Dijkstra [29] however the success of our tests gave us positive signs that our system is performing as intended.

10 Evaluation

Evaluation is to determine if we have achieved the initial project aim. In this chapter we will be discussing our evaluation methods which includes conducting a user study, comparing the system to the requirements and comparing the system with other systems. The results from these evaluation will let us know if we have built what we set out to build and if we have done it correctly.

10.1 User study

We conducted our user study by using a questionanaire, this method is chosen as it is fast to distribute and collect results. In addition, using a questionanaire has the advantage of being easy to analyse the results as it is standarised. The downside however is that there could be ambiguity in the questions which may lead to inaccurate answers.

The idea behind our user study is to have users follow a written tutorial which provides a step by step guide on rewriting using our system, in the end of the tutorial users will answer the questionnaire. The questionnaire is designed to measure users' ability to understand term rewriting using our system and how intuitive was the GUI, the answers were measured using a *Likert Scale* [51]. The full tutorial and questionnaire are included in appendix E and F.

10.1.1 Demographic

Our target demographic for the user study are university students studying computer science or releted fields, we picked students that are not familiar with the concept of term rewriting as this will allow us to gauge how much understanding were they able to gain by using our system. Furthermore, having participants with minimum knowledge in term rewriting give us more accurate result as there will be no advantage between participants. The two charts below provide a summary of our demographic.

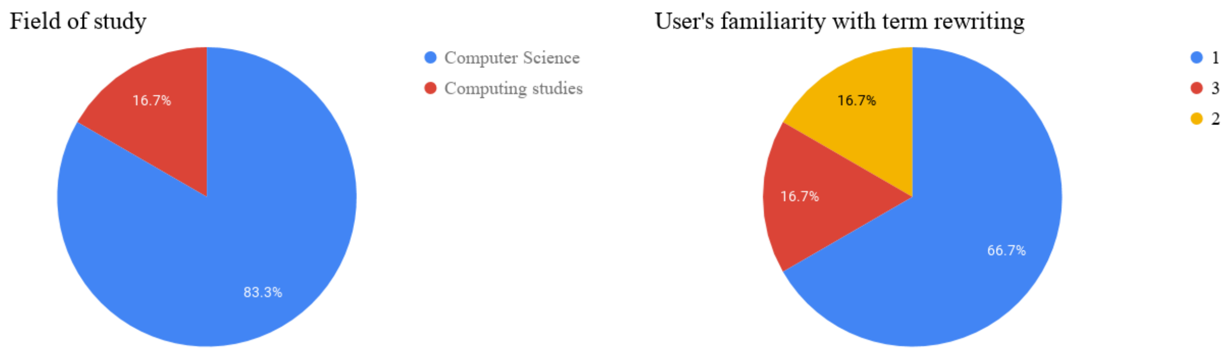
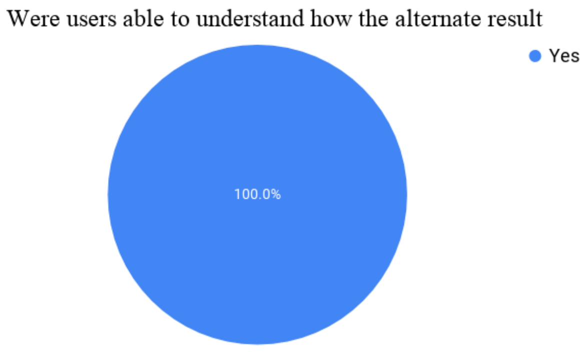


Figure 10.1: Demographic of the user study

10.1.2 Results

In terms of understanding, all users stated that they are able to understand the result derived from the tutorial, furthermore when asked about the understanding about term rewriting that were gained through using the system, 66.7% responded with a 5 on the likert scale while the other 33.3% responded with a 4 as shown in the charts below.



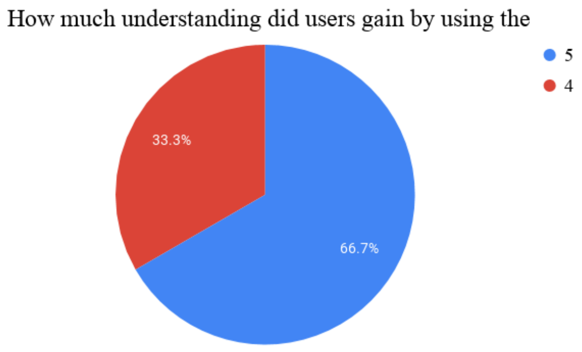
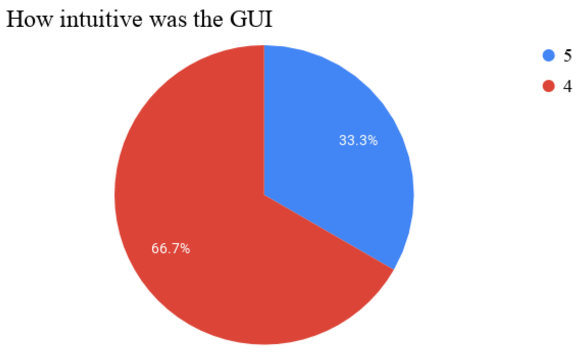


Figure 10.2: Charts showing users ability to understand term rewriting using the system

Building on the questions above, we asked participants how much more of the system were they able to explore. Only a third of users responded with a 5 while the rest responded with a score of 3 or below. We think this might be caused by a lack of time in the users end or the user not being able to find a use case for it. Given more time and a proper use case, they might be able to explore more features of the system.

Finally users are questioned on the graphical user interface of the system, the results are above average with all responses on being 4 or above, these results tell us that participants find the GUI easy to use and has the required features. The charts below summarises the responses.



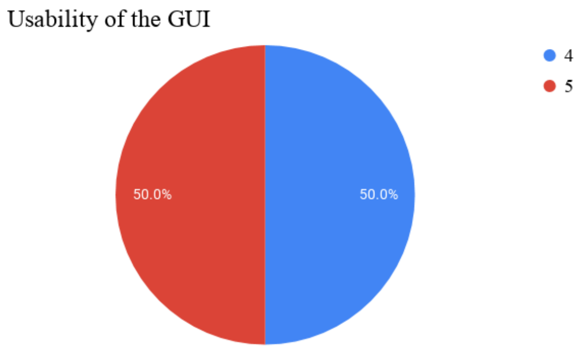


Figure 10.3: Charts summarising responses to GUI questions

There was some issues with the system that are brought up by participants, a number of participants reported that some of the unicode symbols in the system are not showing on their machines. We looked into the issue and found out that it was because the font we used was not in the user's machine, the user's machine will then use the default font which does not have unicode support. Although this isn't a programming error it should be noted and fix in the future as we cannot expect all our users will have the font we needed. A participant that was able to explore the system further told us that loading that graphical visualisation is slow, through investigation we found out that it was because the term provided generated a large tree and hence needed time to load. There isn't an obvious fix for this issue at the moment however we think tweaking the tree building algorithm might be able to provide slight improvement, this should be looked at in the future.

Users have also suggested improvements to the GUI such as having more graphical elements such as icons, having the ability to change the size of text and having the button glow when the mouse hovers across. These suggestions are all taken into consideration and can be added to the system in the future.

10.2 Fulfilment of requirements

We have defined a set of functional and non-functional requirements in section 4.1, the tables below shows each of the requirments and discuss how we have met them.

10.2.1 Functional requirements fulfilment

No.	Requirement	Fulfilment
1	The system will rewrite a term into its normal form following a set of rules.	The system will rewrite a term non-deterministically by applying the first rule that it matches. This process is repeated until the term does not match any rule in the system in which case it is by definition in its normal form.
2	The system will allow users to specify the signature of the system.	The final system provides a way to add unary and binary operators to the system. Users can also add variables of the following type NAT, STRING, BOOLEAN, CONSTANT, INT. All values that are not declared are treated as constants.
3	The system will allow users to specify the rules used to perform rewrite.	The final system allows user to create rewrite rules with regards to a specified signature via a dialog window in the graphical user interface.
4	The system will provide information of the rewrite system.	The final system's home sceen provides an overview of the rewrite system declared by showing the signature and rules. At a glance users can get all the needed information such as the arity of the operators and their return type, type of a variable and rewrite rules with both left and right hand side written out.
5	The system will check for syntax error in user inputs.	Our parsing algorithm is able to detect inputs that are malformed such as having mismatch parenthesis or binary operators with missing operands. Users will be prompted to correct the input before they and advance further.

6	The system will guarantee termination for all user defined rules.	A bound is used to guarantee that all rewrite will terminate. The current bound is set to 100, any rewrite that takes more than 100 steps will be stopped and signaled to the user.
7	The system will provide visualisation of the rewrite process.	The final system provides a textual visualisation of the rewrite where users and analyse the rules applied. A graphical visualisation is also present where user can see all possible state the term can take.
8	The system will allow application of rewrite strategies.	The system allows user to select which rule they want to apply. This way users can perform strategies such as *inner-most* [13] or *outer-most* [13] by selecting the appropraite rules. The system by default performs rewriting using an inner-most strategy.
9	The system will provide analysis capabilities.	The final system includes a search functionality that allows user to search for a particular state given an initial term. The search is performed using a bounded depth-first search on the initial term by applying all possible rules for every state

10.2.2 Non-functional requirements fulfilment

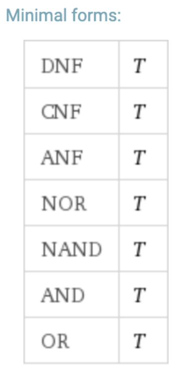
No.	Requirement	Notes
1	The system will provide a graphical user interface.	The final system has a graphical user interface implemented using Java Swing where users are able to interact with and perform all the needed actions.
2	The system will have a window that let users input rewrite rules, operators and variables.	On the home screen users and click the appropriate button to show a pop-up windows where they can add variables, operators and rules by providing the necessary information.
3	The system will allow users to delete rewrite rules, operators and variables.	The panel on the home screen showing the signature and rules allows user to click on an individual rule, variable and operator. Clicking on it allows users to delete the it from the system. The operation is performed on the fly and does not need the system to be restarted.
4	The system will allow saving and loading of rewrite systems.	The final system includes a button that can save the current state of the system as a *.rwr* file which can then be loaded into the system. The system is able to deal with errors and it will notify the user if there was a problem saving or loading file.

10.3 Features comparison with other systems

Previously in section 2.2 we looked at existing system that utilised term rewriting, the final stage of our evaluation is to review the limitations of those systems and how our system solved them.

Wolfram Alpha's lack of customisation One of the limitation of *Wolfram Alpha* is its lack of customisation as Wolfram Alpha's users have no control over the rules and signature. Our system solved the problem by giving users full control over the signature and rules.

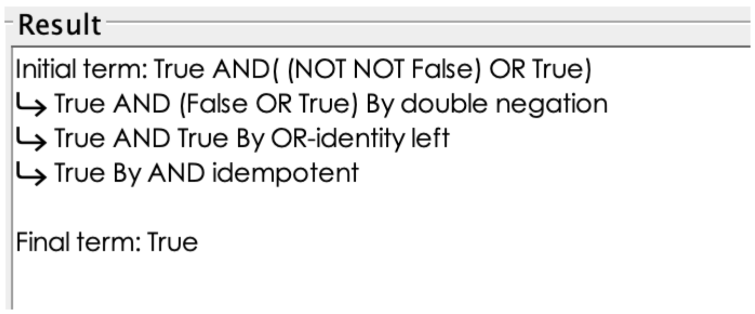
Wolfram Alpha's lack of visualisation Our system's capability to provide an in depth visualisation of the rewrite process is something Wolfram Alpha lack. In Wolfram Alpha, users only get the final result and not each individual step. The figure below shows the comparison between Wolfram Alpha and our system when given the input $True \wedge (\neg\neg False \vee True)$ and the *boolean algebra laws* [23] as the rules.



Minimal forms:

DNF	*T*
CNF	*T*
ANF	*T*
NOR	*T*
NAND	*T*
AND	*T*
OR	*T*

(a) Wolfram Alpha's result



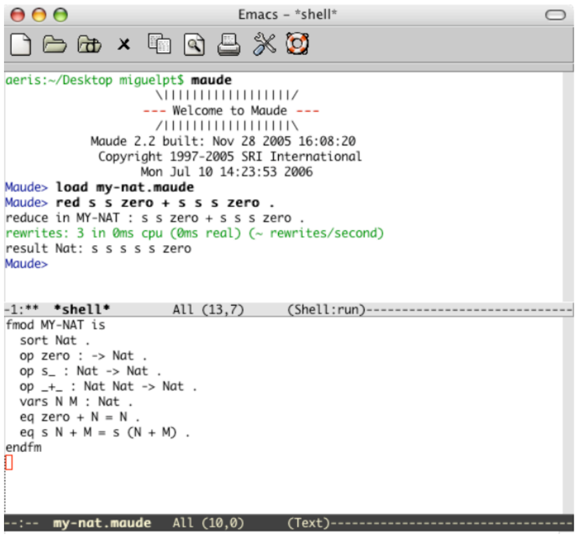
Result

```
Initial term: True AND( (NOT NOT False) OR True)
↳ True AND (False OR True) By double negation
↳ True AND True By OR-identity left
↳ True By AND idempotent

Final term: True
```

(b) Our system's result

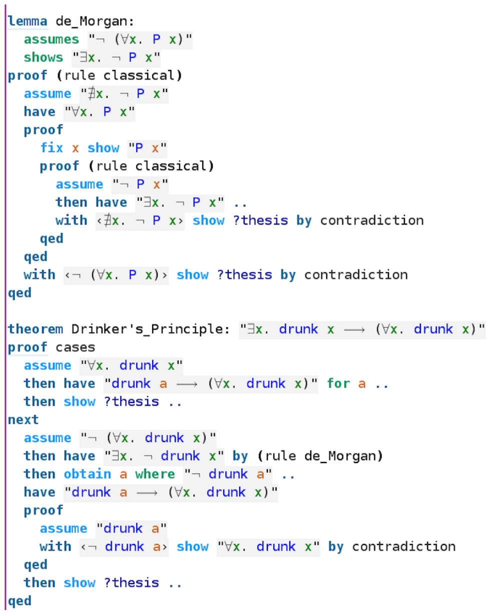
Figure 10.4: The result provided by the systems after performing rewrite

Isabelle and Maude's lack of graphical user interface These two systems does not provide graphical user interfaces and instead relies on textual interface. Although textual user interface is not better than graphical user interface in terms of functionality, they make a big difference for new users [27]. Our system includes a GUI which allows users to interact with the system in an easier way.



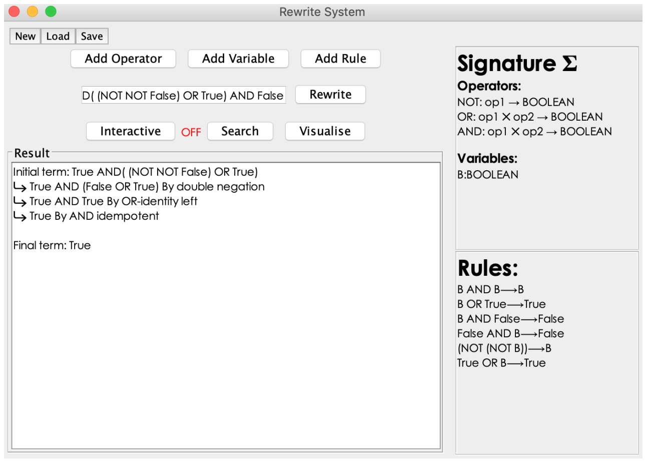
```
aeris:~/Desktop miguelpt$ maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.2 built: Nov 28 2005 16:08:20
             Copyright 1997-2005 SRI International
                   Mon Jul 10 14:23:53 2006
Maude> load my-nat.maude
Maude> red s s zero + s s s zero .
reduce in MY-NAT : s s zero + s s s zero .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s s s s s zero
Maude>

-1:**  *shell*         All (13,7)       (Shell:run)----------------------------
fmod MY-NAT is
  sort Nat .
  op zero : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq zero + N = N .
  eq s N + M = s (N + M) .
endfm

--:--  my-nat.maude    All (10,0)       (Text)--------------------------------
```

(a) Maude's interface



```
lemma de_Morgan:
  assumes "¬ (∀x. P x)"
  shows "∃x. ¬ P x"
proof (rule classical)
  assume "∄x. ¬ P x"
  have "∀x. P x"
  proof
    fix x show "P x"
    proof (rule classical)
      assume "¬ P x"
      then have "∃x. ¬ P x" ..
      with ‹∄x. ¬ P x› show ?thesis by contradiction
    qed
  qed
  with ‹¬ (∀x. P x)› show ?thesis by contradiction
qed

theorem Drinker's_Principle: "∃x. drunk x ⟶ (∀x. drunk x)"
proof cases
  assume "∀x. drunk x"
  then have "drunk a ⟶ (∀x. drunk x)" for a ..
  then show ?thesis ..
next
  assume "¬ (∀x. drunk x)"
  then have "∃x. ¬ drunk x" by (rule de_Morgan)
  then obtain a where "¬ drunk a" ..
  have "drunk a ⟶ (∀x. drunk x)"
  proof
    assume "drunk a"
    with ‹¬ drunk a› show "∀x. drunk x" by contradiction
  qed
  then show ?thesis ..
qed
```

(b) Isabelle's interface



D((NOT NOT False) OR True) AND False

Result

Initial term: True AND((NOT NOT False) OR True)
↳ True AND (False OR True) By double negation
↳ True AND True By OR-identity left
↳ True By AND idempotent

Final term: True

Signature Σ

Operators:
NOT: op1 → BOOLEAN
OR: op1 × op2 → BOOLEAN
AND: op1 × op2 → BOOLEAN

Variables:
B:BOOLEAN

Rules:
B AND B⟶B
B OR True⟶True
B AND False⟶False
False AND B⟶False
(NOT (NOT B))⟶B
True OR B⟶True

(c) Our system's interface

Figure 10.5: The interface of Isabelle and Maude compared to our system

11 Conclusion

11.1 Solution to the problem

The goal of the project was to create a tool that performs term rewriting while providing users flexibility and simplicity. It has to be flexible by providing the ability to let users define the signature and rewrite rules in the system. Furthermore, it has to be simple to understand such that even users with no experience in term rewriting can understand the system and the concept of term rewriting in a short amount of time. The purpose of having a system like this is for users to be able to understand the basic concept of term rewriting.

Our solution to having a flexible rewriting system was to create our own engine which will use an algorithm to find any matches and perform the rewrite using the current set of rules. The algorithm also utilise a parser that we have built that would use the current signature and parse the term into a syntax tree. Using this approach allowed us to manipulate the signature and rules of the system while have the result change according as the parser and algorithm would the working on the latest set of data.

The engine was then implemented into a graphical user interface which provides users an intuitive way to interact with the system. This approach is prefered as a GUI is easier for novice users to understand compared to a textual or command line interface [27]. A GUI is not enough to achieve simplicity, an important feature to include is visualisation which is attained by providing a step by step trace of the rewrite in textual form so that user can get a deeper understanding of the process behind the scene.

We were able to go further by including analysis capabilities to the system which would further help users understand the concept of term rewriting. The first analytical feature is the ability to perform interactive rewrite where users are able to choose and apply rule one at a time instead of having the system doing it automatically. By having that capability in the engine allowed us to build a tree of all possible states that can be achieved from an initial term by applying different rules. Users are then able to use the system to visualise all the possible state and perform a search to find a if a state is reachable from an initial term.

11.2 Achievement of aim and objectives

The aim of the project was "To build a tool for term rewriting that allows users to define their own rules, perform analysis, and visualise the rewrite process through a graphical user interface", the aim would be achieved by achieving the following objectives.

1. Research and evaluate existing rewrite systems.

We researched three popular systems that utilise term rewriting to solve different problems, *Maude* [49], *Isabelle* [55] and *Wolfram Alpha* [58]. We identified the strength and weaknesses of these systems and recorded our findings in section 2.2.

2. Investigate existing parsing techniques and apply the most appropriate for the system.

Our investigation concluded there are two main parsing techniques we can use, a parser generator or writing a parsing algorithm. We evaluated these approaches and concluded that writing an algorithm is the appropriate method (section 3.2.4).

3. Design and implement a rewrite algorithm that transforms inputs based on rewrite rules.

A rewrite algorithm was designed that is able to transform user inputs based on the current rewrite rules. This algorithm was designed completely from scratch and then implemented into the system. Chapter 6 covered this process in detail.

4. Investigate and implement advanced rewrite techniques to control rewriting.

To cope with edge cases we have implemented mechanisms into our algorithm that would control the rewrite so that the system would not run into errors. We covered the techniques we used in section 6.3.

5. Design and implement a Graphical User Interface that allows user to interact with the system.

We designed the graphical user interface using *wireframing* [9] which is then implmeneted using Java Swing. Chapter 8 covers the details of the design and implementation.

6. Investigate and implement different analysis capabilities.

After investigation we decided to include three analytical features in our system, interactive rewriting, viewing all possible states and searching for a state. We were then able to implemented these features into the system as shown in chapter 7.

7. Implement textual and graphical visualisation capabilities

To achieve this objective we have design our rewrite engine to be able to hold intermediate stages of the rewrite process (section 6.1.2). The system was then able to provide the textual visulisation by going through these stages as demonstrated in section 8.2.1 where we included the trace in the GUI. As stated in objective 6 the system is able to search for all possible state by building a tree, the tree can be viewed using a graphical visualisation tool in the system (section 8.2.2).

8. Evaluate the final system to gauge its effectiveness.

The effectiveness of the system was evaluated through a variety of methods including users study, fulfilment of requirements and comparison with other systems. The results of the evaluation is recorded in chapter 10.

The achievement of the objectives above means that we have achieved our aim of developing a term rewriting tool that is simple and flexible.

11.3 Challenges

Undertaking a project with such complexity is challenging, throughtout the projects we were faced with many challenges, the list below is some of the key challenges we faced.

Researching and understanding term rewriting. Term rewriting was not a familiar concept prior to this project, we found that many papers that cover term rewriting are often highly technical and does not give an easy to understand definition of term rewriting. Fortunately we eventually able to find a less technical paper and get the basic understanding of term rewriting. As the project went on and with the help of our supervisor, we were able to go revisit the more technical papers and complete our understanding.

Designing algorithms Although we have vast experience in programming and software development there rarely have been cases where we have to design algorithm from the ground up. Designing the rewrite algorithm was very challenging as it is very involved and have many aspect we need to consider. However we were able to overcome the challenge and we are very pleased with the end result.

Building the graphical visualisation The graphical visualisation is built using a popular Java library however despite its popularity, documentation and guides are limited. We had problems making the graph look the way we have intended in our design, through trial and error we were able to make it work though given more time it can still be improved further.

Large scale software development Managing the development of a system this size is not easy. The system has a lot of features and are dependant on one another, making it crucial that we planned our development well and ensure all deadlines are met. In addition to development there is also design and testing, adding up to a huge workload. However with our proper planning we were able to stay on track throughout the development and complete our system within time.

11.4 Further work

Although our tool works correctly and meets the requirements there is always room for improvements. Below is a list of possible features that can be added to the system in the future.

Support for more operators The system currently supports two types of operator, unary and binary. This is very limited as there are many contexts where operators with more than two operands are needed. Further work can be done to extends the current design to accomodate so that the system accepts *n-nary* opeators for $n > 2$.

Support for different rewrite strategies Currently our algorithm performs term rewriting using an inner-most strategy. However there may be cases where an outer-most strategy is preferable [18]. Users can use the interactive rewrite to apply an outer-most strategy one at a time but that could be a slow process. In future versions we can add a feature that let users choose the strategy they want the system to use when performing the automatic rewrite.

Improved graphical visualisation Currently the graphical visualisation only shows the search tree in one colour and when the user search for a term that is in a tree only a textual trace is produced. We plan to improve it in the future by having the path leading to the target in the tree be a different colour.

Improved GUI design The current design of the graphical user interface is very simple and rely on mostly text for information. A possible improvement is to utilise more graphical assets such as icons/images as it can improve the aesthetic of the system and possibly provide visual aids to the users.

References

[1] Alan Dix, Janet Finlay, Gregory D. Abowd, Russell Beale. Human Computer Interaction. Pearson/Prentice-Hall, Harlow, England New York, 2004.

[2] Avinash Kaur, Purva Sharma, Apurva Verma. A appraisal paper on Breadth-first search, Depth-first search and Red black tree. In: International Journal of Scientific and Research Publications, Volume 4, Issue 3. March 2014.

[3] Barendregt H.P., van Eekelen M.C.J.D., Glauert J.R.W., Kennaway J.R., Plasmeijer M.J., Sleep M.R. . Term graph rewriting. In: PARLE Parallel Architectures and Languages Europe. PARLE 1987. Lecture Notes in Computer Science, vol 259. Springer, Berlin, Heidelberg.

[4] Chris Britton. Choosing a Programming Language. Microsoft. January 2008. `https://docs.microsoft.com/en-us/previous-versions/cc168615(v=msdn.10)`. (accessed on Febuary 4th, 2019)

[5] Ecma International. C Language Specification 5th edition. December 2017. `https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf` (accessed on Febuary 4th, 2019)

[6] Edsger W. Dijkstra. Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60. Stichting Mathematisch Centrum. November 1961.

[7] Erich Gamma, John Vlissides, Richard Helm, Ralph Johnson. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. October 1994.

[8] Erika Corona,Filipino Eros Pani. A Review of Lean-Kanban Approaches in the Software Development. in: Wseas Transactions on Information Science and Applications Issue 1, Volume 10. January 2013.

[9] Experienceux. What is wireframing? `https://www.experienceux.co.uk/faqs/what-is-wireframing/`. (accessed April 2nd, 2019)

[10] Farcic V. Test Driven Development (TDD): Example Walkthrough. Technology Conversations. 2019 `https://technologyconversations.com/2013/12/20/test-driven-development-tdd-example-walkthrough/` (accessed January 28th, 2019)

[11] Glenford J. Myers. The Art of Software Testing, Second Edition. John Wiley Sons, Inc., Hoboken, New Jersey. 2004.

[12] Hinchey, Michael Bowen, Jonathan Rouff, Christopher. Introduction to Formal Methods. in: Agent Technology from a Formal Perspective. Springer. January 2006. Pages 25-64

[13] Horatiu Cirstea, Claude Kirchner, Luigi Liquori, Benjamin Wack. Rewrite Strategies in the Rewriting Calculus. In: Electronic Notes in Theoretical Computer Science 86 No. 4. December 2003.

[14] Hua Li. Binary Tree's Recursion Traversal Algorithm and Its Improvement. in: Journal of Computer and Communications. 2016. Pages 42-47.

[15] Ian Sommerville. Software Engineering 9th. Addison-Wesley Publishing Company , USA. 2010. Pages 205-206

[16] Ian Gorton, George T. Heinemann, Ivica Crnkovic, Heinz Werner Schmidt, Judith A. Stafford, Clemens Szyperski, Kurt Wallnau. Component-Based Software Engineering. in: 9th International Symposium. Springer. June 2006.

[17] IBM Knowledge Center. Pass by reference (C++ only). `https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.cbclx01/cplr233.htm`. (accessed 20th April 2019)

[18] Jaco van de Pol and Hans Zantema. Generalized innermost rewriting. In: Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan. April 2005.

[19] Jakob Nielsen. 10 Usability Heuristics for User Interface Design. Nielsen Normal Group. April 1994. `https://www.nngroup.com/articles/ten-usability-heuristics/` (accessed on April 2nd, 2019)

[20] James Gosling. The Java® Language Specification Java SE 7 Edition. July 2011.

[21] Jan Verschelde. Introduction to Symbolic Computation. Release 0.9.3. January 18 2019. `http://homepages.math.uic.edu/~jan/mcs320/mcs320.pdf` (accessed on January 29th, 2019)

[22] Jean H. Gallier. Logic For Computer Science Foundations of Automatic Theorem Proving. University of Pennsylvania. June 2003.

[23] Jens-Peter Kaps. Laws and Rules of Boolean Algebra. George Mason University. `https://ece.gmu.edu/~jkaps/courses/ece331-s07/resources/boolean.pdf` (accessed on January 29th, 2019)

[24] JGraph Ltd. JGraphX (JGraph 6) User Manual. December 2018. `https://jgraph.github.io/mxgraph/docs/manual_javavis.html` (accessed on Febuary 5th, 2019)

[25] Juan Soulié. C++ Language Tutorial. June 2007. `http://www.cplusplus.com/files/tutorial.pdf` (accessed on Febuary 4th, 2019)

[26] JUNG development team. JUNG Java Universal Network/Graph Framework. `http://jung.sourceforge.net/` (accessed on Febuary 5th, 2019)

[27] Jung-Wei Chen, Jiajie Zhang. Comparing Text-based and Graphic User Interfaces for Novice and Expert Users. In: AMIA Annu Symp Proc. 200.Pages 125–129.

[28] Joel Jones. Abstract Syntax Tree Implementation Idioms.Department of Computer Science, University of Alabama.

[29] J.N. Buxton and B. Randell, eds, Software Engineering Techniques. In: Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969. April 1970. Page 16.

[30] J.W Klop. Term Rewriting System. in: Handbook of logic in Computer Science volume 2. Oxford University Press. Oxford. March 1st 1993. Pages 1-116.

[31] Martin A. Ould, Charles Unwin. Testing in Software Development. Cambridge University Press. 1986. Pages 71-74.

[32] Max Dauchet. Simulation of Turing machines by a left-linear rewrite rule. Rewriting Techniques and Applications. Springer Berlin Heidelberg. 1989. Pages 109-120.

[33] Martin Erwig, Rahul Gopinath. Explanations for Regular Expressions. School of EECS Oregon State University.

[34] Nachum Dershowitz Computing with Rewrite Systems. in: Information and Control, Volume 65, Issues 2–3. Academic Press Professional, Inc. San Diego, CA, USA 1985. Pages 122-157

[35] Nachum Dershowitz, Laurent Vigneron. Rewriting Home Page. Rewriting.loria.fr. 2019 `http://rewriting.loria.fr/` (accessed on January 29th, 2019)

[36] Neil Sculthrope, Nicolas Frisby and Andy Gill. The Kansas University Rewrite Engine A Haskell-Embedded Strategic Programming Language with Custom Closed Universes. Journal of Functional Programming, vol. 24. July 2014. Pages 434–473.

[37] Overbey, Jeffrey L, Johnson, Ralph E. Generating Rewritable Abstract Syntax Trees. In: Gašević D., Lämmel R., Van Wyk E. (eds) Software Language Engineering. SLE 2008. Lecture Notes in Computer Science, vol 5452. Springer, Berlin, Heidelberg. 2008. pages 114-133

[38] Oxagile. Waterfall Software Development Model. 5 February 2014. `https://www.oxagile.com/company/blog/the-waterfall-model/` (accessed on January 31st, 2019)

[39] Oracle. Collections Framework Overview. Java Documentation. `https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html` (accessed on February 19th, 2019)

[40] Paul Kint. Quick Introduction to Term Rewriting. `https://homepages.cwi.nl/~daybuild/daily-books/extraction-transformation/term-rewriting/term-rewriting.pdf` (accessed on January 29th, 2019)

[41] Phillip A. Laplante. What Every Software Engineer Should Know. CRC Press. 2007. Pages 85-87

[42] Regular-expressions.info. Lookahead and Lookbehind Zero-Length Assertions. `https://www.regular-expressions.info/lookaround.html` (accessed on February 20th, 2019)

[43] Robert Tarjan. Depth-First Search and linear graph algorithms. In: Siam.J.Comput Volume 1, No. 2. June 1972. Pages 146-160.

[44] Rob Pooley and Perdita Stevens. Component Based Software Engineering with UML. Addison-Wesley, November 1998.

[45] Ronald V. Book and Friedrich Otto. String-Rewriting Systems. Springer, New York. 1993.

[46] Scrum, Inc. The Scrum Papers: Nut, Bolts, and Origins of an Agile Framework Version 1.1. 2 April 2012

[47] Sheetal Sharma, Darothi Sarkar , Divya Gupta. Agile Processes and Methodologies: A Conceptual Study. in: International Journal on Computer Science and Engineering (IJCSE) Volume. 4 No. 05. May 2012.

[48] Shubhmeet Kaur. A Review of Software Development Life Cycle Models. in: International Journal of Advanced Research in Computer Science and Software Engineering Volume 5, Issue 11. November 2015.

[49] SRI International. The Maude System. `http://maude.cs.illinois.edu/w/index.php/The_Maude_System` (access January 29th 2019).

[50] Steven Schwartzman. Explanation of terms and symbols. In The Words of Mathematics: An Etymological Dictionary of Mathematical Terms used in English. Mathematical Association of America 1994. Pages 11-16.

[51] SurveyMonkey. What is a Likert scale?. `https://www.surveymonkey.com/mp/likert-scale/` (accessed 24th April 2019).

[52] Terence Parr, Sam Harwell, Kathleen Fisher. Adaptive LL(*) parsing: the power of dynamic analysis In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages Applications (OOPSLA '14). ACM, New York, NY, USA. October 15th, 2014. pages 579-598.

[53] Teresa. Term Rewriting Systems. Cambridge University Press. 2003.

[54] The University of Texas at San Antonio. Principles of Object-Oriented Design. `http://www.cs.utsa.edu/~cs3443/notes/designPrinciples/designPrinciples.html` (accessed March 5th 2019)

[55] University of Cambridge, Technische Universität München. Isabelle `https://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html`

[56] University of Manchester. Infix, Postfix and Prefix. `http://www.cs.man.ac.uk/~pjj/cs212/fix.html` accessed November 23rd 2018).

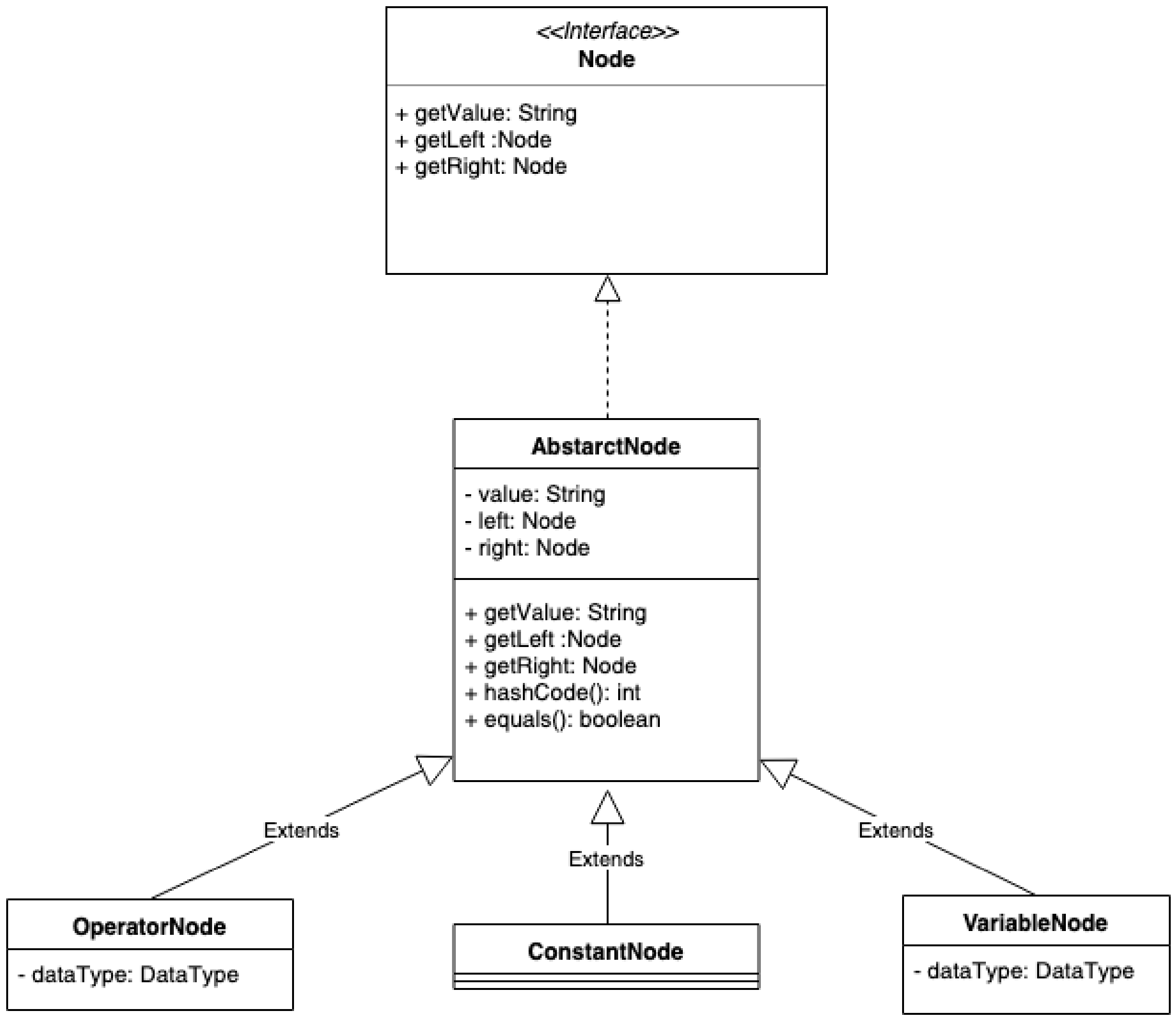
[57] Vassili Kaplan. A New Algorithm to Parse a Mathematical Expression and its Application to Create a Customizable Programming Language. In: ICSEA 2016 : The Eleventh International Conference on Software Engineering Advances. Rome, Italy. August 21st 2016. pages 272-277.

[58] Wolfram Alpha LLC. 2009. Wolfram—Alpha. `http://www.wolframalpha.com/input/?i=2%2B2` (access January 29th 2019).

[59] w3sDesign.com. The Factory Method design pattern - Structure and Collaboration. `http://w3sdesign.com/?gr=c03&ugr=proble#gf` (access March 4th 2019).

Appendix

A Initial syntax tree design



B Different notations parsing methods

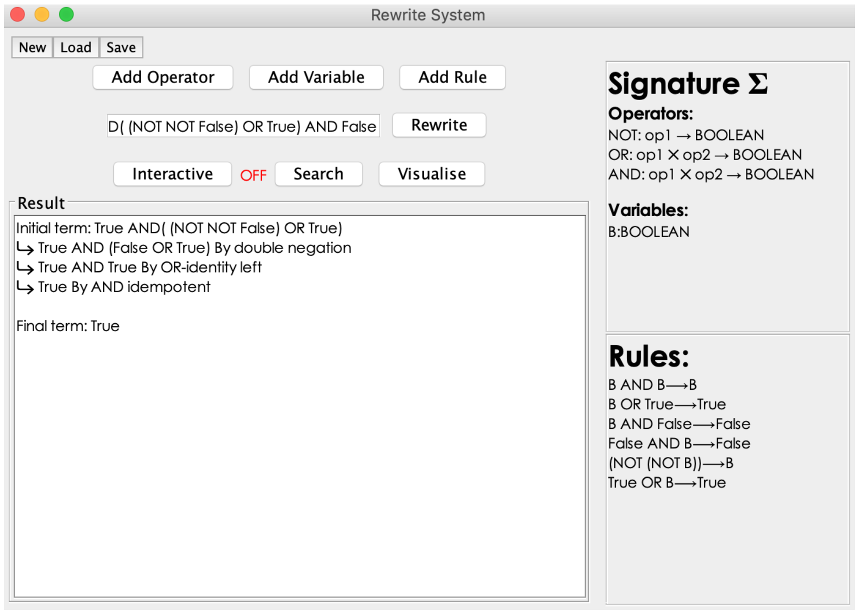
Converting postfix into infix

```
public String toInfix(String rpn) throws ParseException{


   StringBuilder sb = new StringBuilder();
   Stack<String> exprStack = new Stack<>();

   List<String> tokens = Arrays.asList(rpn.split(" "));

   for(String s: tokens) {

      if(!context.isOperator(s)) { //if not operator
         exprStack.push(s);
      }else {

         //if unary
         Operator o = context.getOperator(s);

         if(o instanceof UnaryOperator) {
            String next = "("+o.getSymbol()+exprStack.pop()+")";
            exprStack.push(next);
         }else {
            String r = exprStack.pop();
            String l = exprStack.pop();
            String next = "("+l+o.getSymbol()+r+")";
            exprStack.push(next);
         }

      }
   }

   return exprStack.pop();

}
```

C Shunting Yard parsing algorithm implementation

```
public Node parseAST(String infix) throws ParseException{
    Stack<String> opStack = new Stack<>();
    Stack<Node> nodes = new Stack<>();
    List<String> list = tokenizeString(infix);

    if(!verifyTerm(list))
        throw new ParseException("Syntax Error");

    //Begin Shunting Yard

    for(String s:list) {

        //cases, '(' , ')' , is operator, not operator

        if(s.equals("(")) {

            opStack.push("(");
        }

        else if(s.equals(")")) {

            boolean foundOpen = false;
            //append all operator until the opening parenthesis
            while(!opStack.isEmpty()) {
                String p = opStack.pop();

                if(p.equals("(")) {
                    foundOpen = true;
                    break;
                }else {
                    addNodeToStack(nodes,p);
                }
            }
            //mismatch if no opening is found
            if(!foundOpen)
                throw new ParseException("Mismatach parenthesis");
        }else {

            if(sig.isOperator(s)) {

                Operator current = sig.getOperator(s);
                while(!opStack.isEmpty()) {
                    Operator last = sig.getOperator(opStack.peek());

                    //if the last op is opening parenthesis
                    if(last == null)
                       break;
                    //add all the operators that are lower or same precedence as
                        current
                    if(current instanceof BinaryOperator &&
                        (current.comparePrecedence(last) == 0 ||
                        current.comparePrecedence(last) == -1)) {
                       opStack.pop();
                       addNodeToStack(nodes,last.getSymbol());
                    }else {
                       break;
                    }
                }
                opStack.push(s);

            }else {
                if(sig.isVariable(s))
                    nodes.push(new
                        ASTNode(s,null,null,sig.getVariable(s),NodeType.VARIABLE));
                else
                    nodes.push(new
                        ASTNode(s,null,null,DataType.CONST,NodeType.CONSTANT));
            }
        }
    }
    //add the remaining operators
    while(!opStack.isEmpty()) {
        String c = opStack.pop();
        //there should be no parenthesis left
        if(c.equals("("))
            throw new ParseException("Mismatach parenthesis");
        addNodeToStack(nodes,c);
    }
    //end shunting yard
    return nodes.peek();
}
```

D Screenshots of the system



D((NOT NOT False) OR True) AND False

```
Initial term: True AND( (NOT NOT False) OR True)
↳ True AND (False OR True) By double negation
↳ True AND True By OR-identity left
↳ True By AND idempotent

Final term: True
```

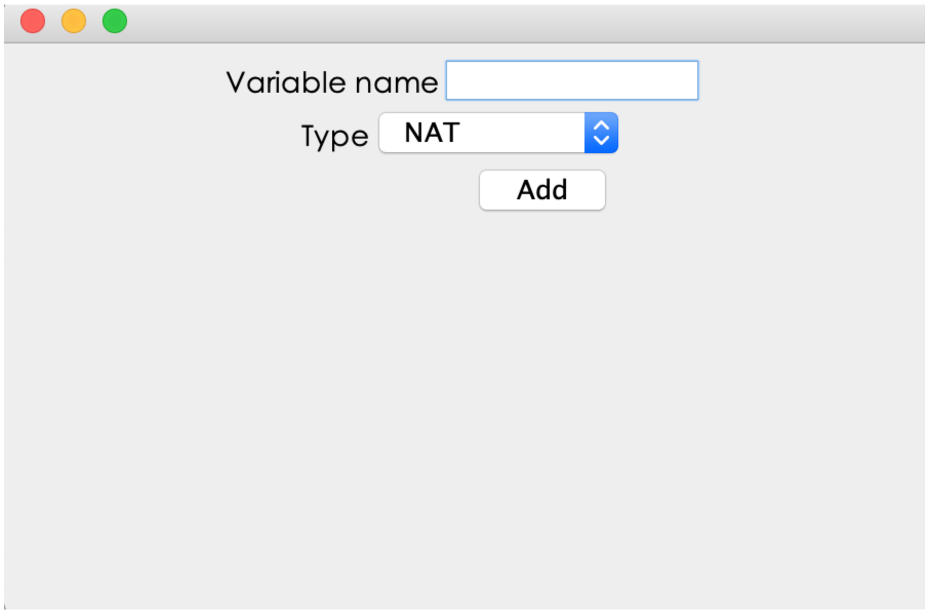
Signature Σ

Operators:
NOT: op1 → BOOLEAN
OR: op1 × op2 → BOOLEAN
AND: op1 × op2 → BOOLEAN

Variables:
B:BOOLEAN

Rules:
B AND B⟶B
B OR True⟶True
B AND False⟶False
False AND B⟶False
(NOT (NOT B))⟶B
True OR B⟶True

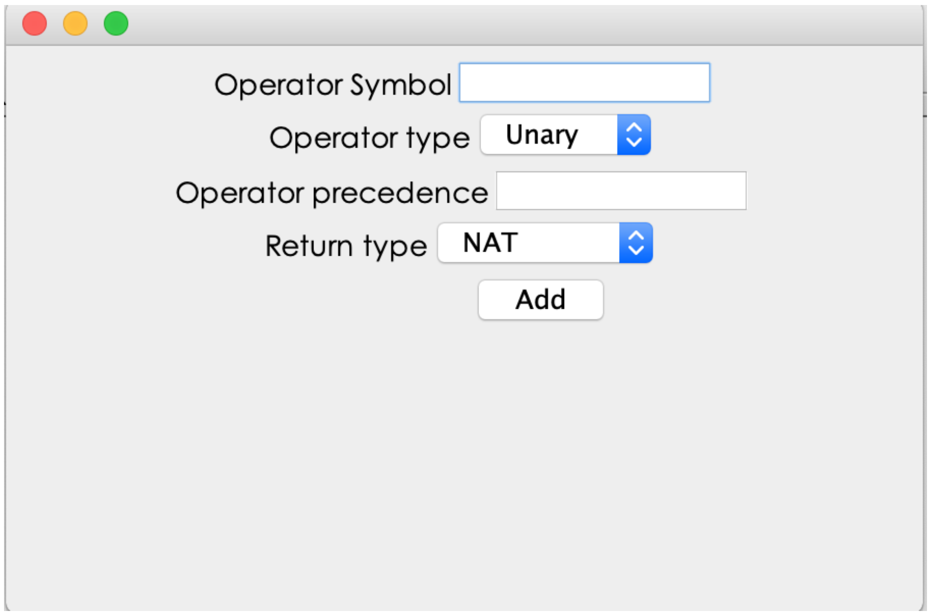
Figure D.1: Home Screen



Variable name

Type NAT

Figure D.2: Add Variable Window



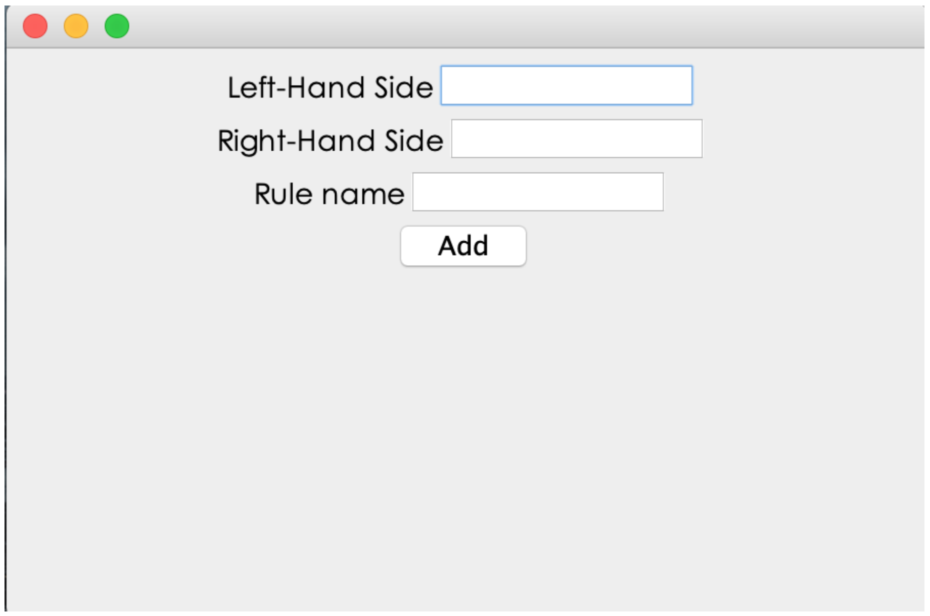
Operator Symbol

Operator type: Unary

Operator precedence

Return type: NAT

Figure D.3: Add Operator Window



Left-Hand Side

Right-Hand Side

Rule name

Figure D.4: Add Rule Window

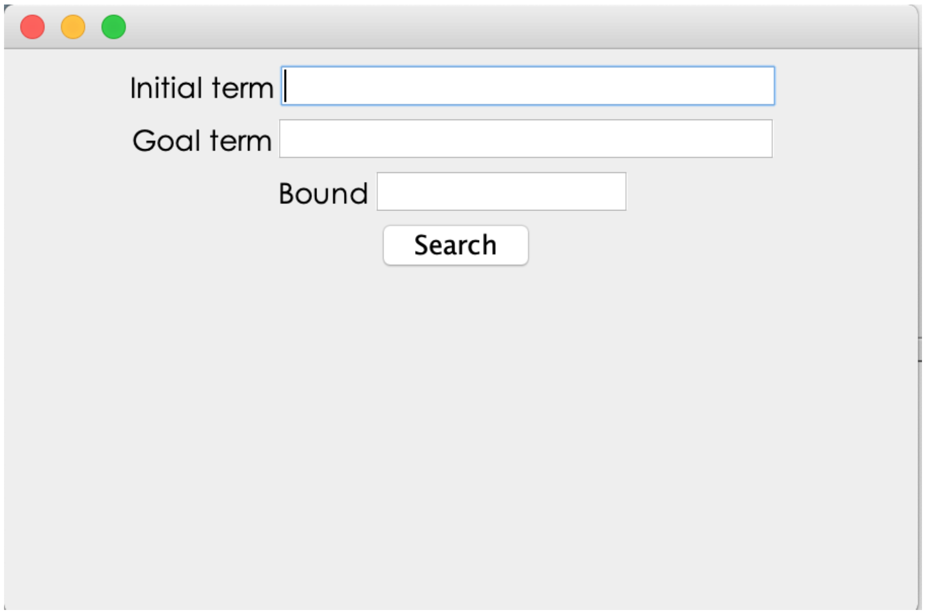
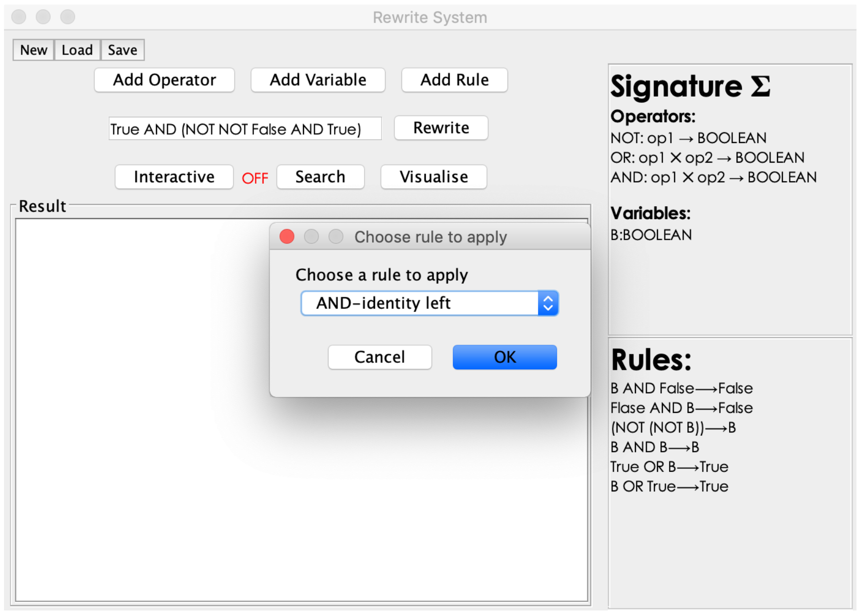


Figure D.5: Search Window



True AND (NOT NOT False AND True)

Choose a rule to apply

AND-identity left

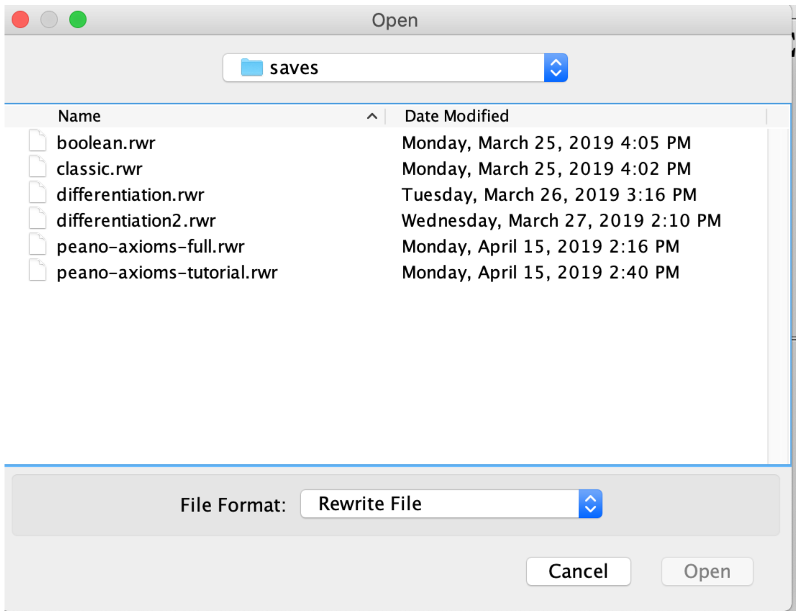
Signature Σ

Operators:
NOT: op1 → BOOLEAN
OR: op1 × op2 → BOOLEAN
AND: op1 × op2 → BOOLEAN

Variables:
B:BOOLEAN

Rules:
B AND False⟶False
Flase AND B⟶False
(NOT (NOT B))⟶B
B AND B⟶B
True OR B⟶True
B OR True⟶True

Figure D.6: Interactive Rewrite Window



Open

saves

Name	Date Modified
boolean.rwr	Monday, March 25, 2019 4:05 PM
classic.rwr	Monday, March 25, 2019 4:02 PM
differentiation.rwr	Tuesday, March 26, 2019 3:16 PM
differentiation2.rwr	Wednesday, March 27, 2019 2:10 PM
peano-axioms-full.rwr	Monday, April 15, 2019 2:16 PM
peano-axioms-tutorial.rwr	Monday, April 15, 2019 2:40 PM

File Format: Rewrite File

Cancel Open

Figure D.7: Load File Window

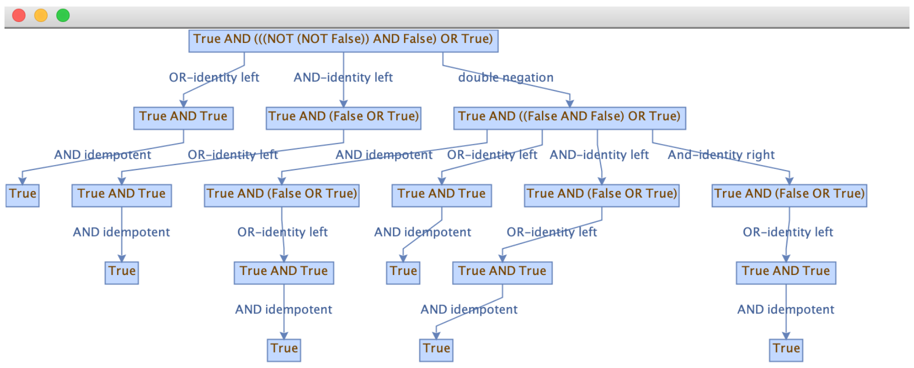


Figure D.8: Search tree visualisation

E User tutorial

Term Rewriting System Tutorial and User Study

This tutorial provides and step by step walkthrough of term rewriting using the rewriting system provided. The purpose of the system is to give users an understanding of term rewriting in an easy to understand manner. In the end of the tutorial user will partake in a user study to evaluate the system's effectiveness.

The tutorial is based on the Peano-Axioms and shows how to apply term rewriting on the arithmetic definitions. We expect a decent understanding of mathematics.

Rewrite System Introduction

First let's define what is a rewrite system, a rewrite system uses a set of rules to transform objects. Specifically in term rewriting, it transform mathematical terms. For example, if we know that any number multiplied by 1 is itself, then based on that rule we can transform x*1 into simply x for every value of x. We do not need to perform any computation here, we just pattern match with the rule and apply the transformation.

There are three components to a term rewriting system.

Variables: These are values that can vary throughout the system. Variable have a symbol(x, y, i, etc) and a type(integer, natural number, Boolean, etc).

Operators: $+, -, *, etc$ are example of operators where they take two numbers(operands) and they return a value. Operators can be binary like + where they take two numbers or unary like – for negative numbers. All operators have a return type which are the type of the value they return, for example $+, - *$ has a integer return type.

Rules: Rules are the way to transform a term, it has a left and right hand side. For example: $x * 1 \rightarrow x$. If a term matches the left hand side then we can replace it with the right hand side. Note that when matching, it maps variables to a value. $7 * 1$ will match the rule because x will be mapped to 7 and when we replace it we swap x with 7, so $7 * 1 \rightarrow 7$. Variable can also be matched with operators with the return type, such as $(32 + 97 * 3) * 1 \rightarrow (32 + 97 * 3)$, in this scenario x maps to $(32 + 97 * 3)$.

Example using Boolean algebra

Rules:

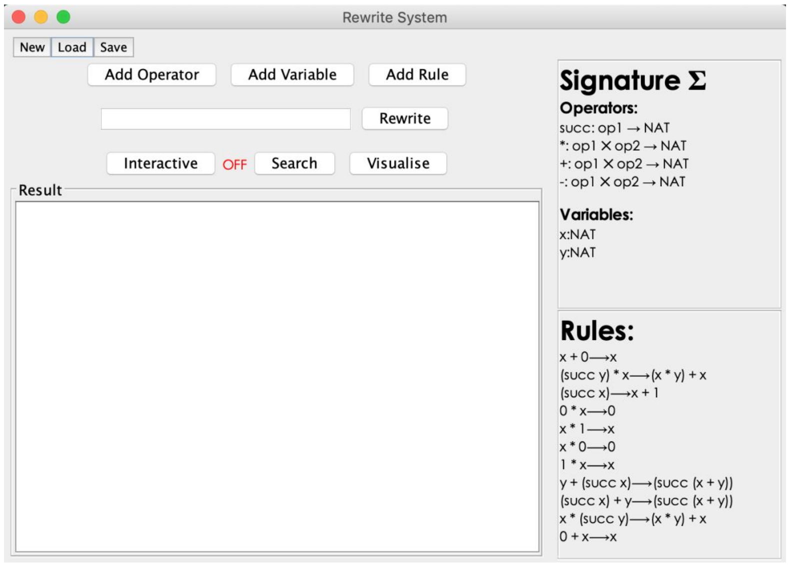
$\neg\neg x \to x$ (Double negation law)
$x \wedge x \to x$ (Idempotent law)
$x \vee True \to True$ (Identity law)

Rewrite:

$False \vee (True \wedge \neg\neg True)$
$\hookrightarrow False \vee (True \wedge True)$ (Double negation law)
$\hookrightarrow False \vee True$ (Idempotent law)
$\hookrightarrow True$ (Identity law)

Using the System

Let's dive into more details and see how term rewriting is performed. Open up the *RewriteSystem.jar* file. Click on the load button on the top left hand corner and open *peano-axioms-tutorial.rwr*. You should see the following window.



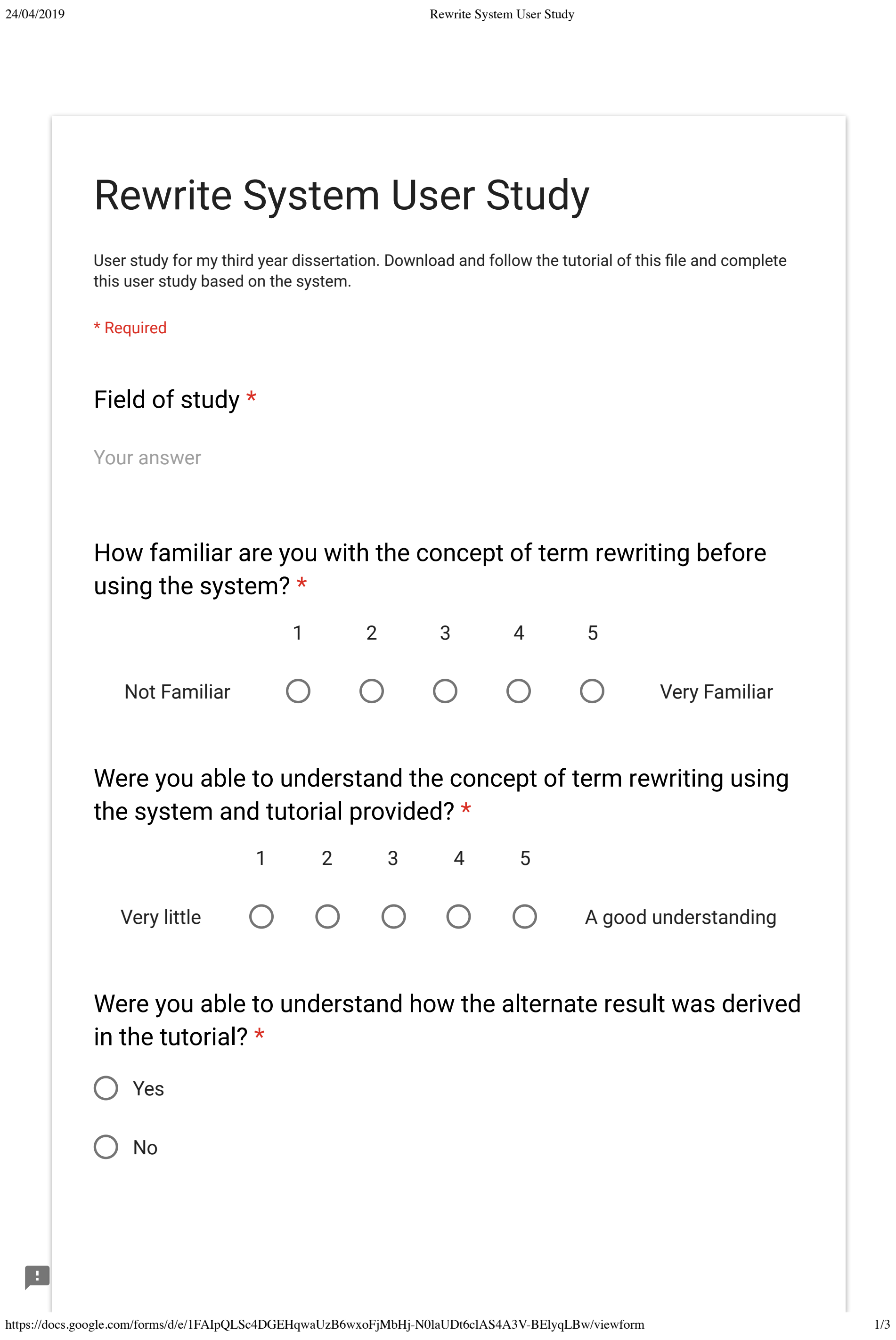
Signature Σ

Operators:
succ: op1 → NAT
*: op1 X op2 → NAT
+: op1 X op2 → NAT
-: op1 X op2 → NAT

Variables:
x:NAT
y:NAT

Rules:
x + 0⟶x
(succ y) * x⟶(x * y) + x
(succ x)⟶x + 1
0 * x⟶0
x * 1⟶x
x * 0⟶0
1 * x⟶x
y + (succ x)⟶(succ (x + y))
(succ x) + y⟶(succ (x + y))
x * (succ y)⟶(x * y) + x
0 + x⟶x

F Questionnaire



24/04/2019 Rewrite System User Study

Rewrite System User Study

User study for my third year dissertation. Download and follow the tutorial of this file and complete this user study based on the system.

* Required

Field of study *

Your answer

How familiar are you with the concept of term rewriting before using the system? *

	1	2	3	4	5	
Not Familiar	○	○	○	○	○	Very Familiar

Were you able to understand the concept of term rewriting using the system and tutorial provided? *

	1	2	3	4	5	
Very little	○	○	○	○	○	A good understanding

Were you able to understand how the alternate result was derived in the tutorial? *

○ Yes

○ No

https://docs.google.com/forms/d/e/1FAIpQLSc4DGEHqwaUzB6wxoFjMbHj-N0laUDt6clAS4A3V-BElyqLBw/viewform 1/3

Were you able to explore the system by providing your own inputs that are not in the tutorial? *

◯ Yes

◯ No

How much were you able to explore the additional features of the system that weren't covered in the tutorial. *

	1	2	3	4	5	
Did not explore	◯	◯	◯	◯	◯	Explored every aspect of the system

How intuitive is the GUI? *

	1	2	3	4	5	
Confusing to use	◯	◯	◯	◯	◯	Simple to use

How usable is the GUI? *

	1	2	3	4	5	
Lacking features	◯	◯	◯	◯	◯	Has the required features

Additional comments

Your answer

SUBMIT

Never submit passwords through Google Forms.

Were you able to explore the system by providing your own inputs that are not in the tutorial? *

○ Yes

○ No

How much were you able to explore the additional features of the system that weren't covered in the tutorial. *

	1	2	3	4	5	
Did not explore	○	○	○	○	○	Explored every aspect of the system

How intuitive is the GUI? *

	1	2	3	4	5	
Confusing to use	○	○	○	○	○	Simple to use

How usable is the GUI? *

	1	2	3	4	5	
Lacking features	○	○	○	○	○	Has the required features

Additional comments

Your answer

SUBMIT

Never submit passwords through Google Forms.

G Test cases

Syntax tree unit tests cases

Name	Input	Expected	P/F
testEquals	Two different trees with + as the root and x, y as left and right nodes	*assertEquals* is true and *assertSame* is false	P
testEquals2	Same as *testEquals* but one of the tree is a sub-tree	*assertEquals* is true and *assertSame* is false	P
testNotEquals	Comparing a tree with + as root and x, y nodes with a tree with + as root and y, z as nodes.	*assertEquals* is false	P
testNotEquals	Comparing two trees with completely different nodes.	*assertEquals* is false	P
testComparePrecedenceBinaryBinary1	*o1=BinaryOperator("*",3,INT)* *o2=BinaryOperator("+",2,INT)* *o1.comparePrecedence(o2)*	1	P
testComparePrecedenceBinaryBinary2	*o1=BinaryOperator("*",3,INT)* *o2=BinaryOperator("+",2,INT)* *o2.comparePrecedence(o1)*	-1	P
testComparePrecedenceBinaryBinary3	*o1=BinaryOperator("-",2,INT)* *o2=BinaryOperator("+",2,INT)* *o2.comparePrecedence(o1)*	0	P
testComparePrecedenceBinaryBinary4	*o1=BinaryOperator("*",3,INT)* *o2=BinaryOperator("/",3,INT)* *o2.comparePrecedence(o1)*	1	P
testComparePrecedenceUnaryBinary1	*o1=BinaryOperator("-",1,INT)* *o2=UnaryOperator("!",1,INT)* *o2.comparePrecedence(o1)*	1	P
testComparePrecedenceUnaryBinary2	*o1=BinaryOperator("*",3,INT)* *o2=UnaryOperator("!",1,INT)* *o1.comparePrecedence(o2)*	-1	P
testComparePrecedenceUnaryUnary1	*o1=UnaryOperator("-",1,INT)* *o2=UnaryOperator("!",1,INT)* *o1.comparePrecedence(o2)*	0	P
testComparePrecedenceUnaryUnary2	*o1=UnaryOperator("-",1,INT)* *o2=UnaryOperator("!",2,INT)* *o1.comparePrecedence(o2)*	-1	P
testComparePrecedenceUnaryUnary3	*o1=UnaryOperator("-",1,INT)* *o2=UnaryOperator("!",2,INT)* *o2.comparePrecedence(o1)*	1	P

Parser unit test cases

Name	Input	Expected	P/F
testTokenizeString1	$3+4$	$[3,+,4]$	P
testTokenizeString2	$13+10/2$	$[13,+,10,/,2]$	P
testTokenizeString3	$13+4-2+(10-87)$	$[13,+,4,-,2,+,(,10,-,87)]$	P
testTokenizeString4	$y+z-t$	$[y,+,z,-,t]$	P
testTokenizeString5	True+False−(True+True)	[True,+,False,−,(,True,+,True)]	P
testTokenizeString6	True AND False OR True AND True	[TrueANDFalseORTrue-ANDTrue]	P
testTokenizeString7	True+False−True(True+True)	[True,+,False,-,True,(,True,+,True,)]	P
testTokenizeString8	$34+21-2(7)$	$[34,+,21,-,2,(,7,)]$	P
testTok-enizeStringChain-Operations1	$a+-b+c$	$[a,+,-,b,+,c]$	P
testTok-enizeStringChain-Operations2	$a+-++++b+c$	$[a,+,-,+,+,+,+,b,+,c]$	P
testTok-enizeStringChain-Operations3	$+-++++b$	$[+,-,+,+,+,+,b]$	P
testTokenizeString-Function1	$succ(x)$	$[succ,(,x,)]$	P
testTokenizeString-Function2	$succ(x+y)$	$[succ,(,x,+,y,)]$	P
testTokenizeString-Function3	$succ(succ(x+y))$	$[succ,(,succ,(,x,+,y,),)]$	P
testInfixToRPN1	$3+4$	3 4 +	P
testInfixToRPN2	$3+4+2$	3 4 + 2 +	P
testInfixToRPN3	$3+4*2$	3 4 2 ∗ +	P
testInfixToRPN4	$13+4*2$	13 4 2 ∗ +	P
testInfixToRPN5	$(13+4)*2$	13 4 + 2 ∗	P
testInfixToRPN6	$1+10*(2+2)+(2-1)$	1 10 2 2 + ∗ + 2 1 − +	P
testInfixToRPN7	$2*((3+2)-1)$	2 3 2 + 1 − ∗	P
testInfixToRPN8	$True+False-(True+True)$	True False + True True + −	P
testInfixToRPN9	$True+False-(True+(False-False))+True+True$	True False + True False False − + − True + True +	P
testInfixToRPN10	$!!True$	$True$! !	P
testInfixToRPN11	$True+(!!True)$	$True$ $True$! ! +	P
testInfixToRPN12	$succ(x)$	x $succ$	P

testInfixToRPN13	$succ(x)+y$	$x\ \ succ\ \ y\ \ +$	P
testInfixToRPN14	$succ(x+y)$	$x\ \ y\ \ succ$	P
testInfixToRPN15	$y+succ(x)$	$y\ \ x\ \ succ\ \ +$	P
testInfixToRPN16	$True+!!False$	$True\ \ False\ \ !\ \ !\ \ +$	P
testInfixToRPNException1	$13+(4*2$	$ParseException$	P
testInfixToRPNException2	$()13+4*2$	$ParseException$	P
testInfixToRPNException3	$13)+4*2$	$ParseException$	P
testInfixToRPNException4	$(13+4))*2$	$ParseException$	P
testInfixToRPNException5	$14+4*2+($	$ParseException$	P
testInfixToRPNException6	$2+-4*2$	$ParseException$	P
testInfixToRPNException7	$2+-+4*2$	$ParseException$	P
testInfixToRPNException8	$succ(2)+-+4*2$	$ParseException$	P
testInfixToRPNException9	$2+(-2+3)$	$ParseException$	P
testInfixToRPNException10	$2+2(-2+3)$	$ParseException$	P
testInfixToRPNException11	$2+2-(2+3)+$	$ParseException$	P
testInfixToAST1	$3+4$	3 4 +	P
testInfixToAST2	$3+4*2$	3 4 + 2 +	P
testInfixToAST3	$3+4+2$	3 4 2 * +	P
testInfixToAST4	$(13+4)*2$	13 4 + 2 *	P
testInfixToAST5	$1+10*(2+2)+(2-1)$	1 10 2 2 + * + 2 1 − +	P
testInfixToAST6	$2*((3+2)-1)$	2 3 2 + 1 − *	P
testInfixToAST7	$True+False-(True+True)$	True False + True True + −	P
testInfixToAST9	$True+False-(True+(False-False))+True+True$	True False + True False False − + − True + True +	P
testInfixToAST8	$!!True$	True ! !	P
testInfixToAST10	$True+(!!True)$	True True ! ! +	P

Rewrite Engine Unit test cases

Name	Input	Expected	P/F
testRewrite1	True AND True	True	P
testRewrite2	NOT NOT True	True	P
testRewrite3	True AND False OR (True AND True)	True	P
testRewrite4	True AND False OR (True AND (False OR False)) AND True AND True"	False	P
testRewrite5	$succ(succ(0)) + z$	z *succ succ*	P
testRewrite6	$succ(succ(0)) + succ(0)$	0 *succ succ succ*	P
testRewrite7	True AND False OR ((True AND TRUE)OR TRUE AND False) OR False	False	P
testRewrite8	True AND ((NOT NOT True) AND (False OR True AND True)) AND (False OR (True AND True OR False))	True	P
testRewrite9	$0 + succ(0) * succ(0) + z$	z	P
testRewrite10	$succ(0) + (succ(4) + (succ(0) * succ(0)))$	4 *succ succ*	P
testRewrite11	$succ(1) * (succ(10) + (((succ(0) + 0) * succ(1))))$	10	P
testRewriteSingle1	NOT NOT True, rule:NOT NOT B $\rightarrow$ B	True	P
testRewriteSingle2	True AND (NOT NOT True), rule:NOT NOT B $\rightarrow$ B	True AND True	P
testRewriteSingle3	True AND (NOT NOT True), rule: B AND B $\rightarrow$ B	True AND (NOT NOT True)	P
testRewriteSingle4	True AND (NOT NOT True), rule: NOT NOT B $\rightarrow$ B followed by B AND B $\rightarrow$ B	True	P
testRewriteSingle5	True AND (NOT NOT True) OR False, rule: NOT NOT B $\rightarrow$ B followed by B AND B $\rightarrow$ B	True OR False	P
testRewriteSingle4	True AND (NOT NOT False) OR False, rule: NOT NOT B $\rightarrow$ B followed by B AND B $\rightarrow$ B	True AND False OR False	P
testRewriteNone1	True AND 1	True AND 1	P
testRewriteNone2	True	True	P
testRewriteInfinite1	1+2	*RewriteException*	P