KARADENİZ TECHNICAL UNIVERSITY
The Graduate School of Natural and Applied Science

Computer Engineering Graduate Program

DESIGN AND APPLICATIONS OF GRAMMAR-BASED METHODOLOGIES FOR AUTOMATIC GENERATION AND STEP-BY-STEP SOLVING OF MATHEMATICAL EXPRESSIONS

Ph.D. THESIS

Computer Eng. Mir Mohammad Reza Alavi Milani

FEBRUARY 2015
TRABZON

KARADENIZ TECHNICAL UNIVERSITY
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCE
COMPUTER ENGINEERING GRADUATE PROGRAM

DESIGN AND APPLICATIONS OF GRAMMAR-BASED METHODOLOGIES FOR AUTOMATIC GENERATION AND STEP-BY-STEP SOLVING OF MATHEMATICAL EXPRESSIONS

Mir Mohammad Reza Alavi Milani

This Thesis is Accepted to Give the Degree of
“DOCTOR OF PHILOSOPHY IN COMPUTER ENGINEERING
By
The Graduate School of Natural and Applied Sciences
at Karadeniz Technical University

The date of Submission: 18 . 11 . 2014
The date of Examination: 25 . 02. 2015

Thesis Supervisor: Assist. Prof. Dr. Hüseyin PEHLİVAN

Trabzon 2015

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Ana Bilim Dalında
Mir Mohammad Reza ALAVI MILANI Tarafından Hazırlanan

MATHEMATIKSEL İFADELERİN OTOMATİK ÜRETİMİ VE ADIM-SDIM ÇÖZÜMÜNE YÖNELİK GRAMER TABANLI METODOLOJİLERİN TASARIMI VE UYGULAMALARI

Başlıklı bu çalışma, Enstitü Yönetim Kurulunun 18 / 11 / 2014 gün ve 1577 sayılı kararıyla oluşturulan jüri tarafından yapılan sınavda
DOKTORA TEZİ
olarak kabul edilmiştir.

Jüri Üyeleri

Başkan: Yrd.Doç. Dr. Hüseyin PEHLİVAN …………………………...

Üye : Doç. Dr. Mustafa ULUTAŞ ……………………………

Üye : Prof. Dr. Adem Sefa AKPINAR …………………………...

Üye : Prof. Dr. Vasif V. NABİYEV …………………………...

Üye : Prof. Dr. Bahar KARAOĞLAN …………………………...

Prof. Dr. Sadettin KORKMAZ
Enstitü Müdürü

FOREWORD

I would like to offer my hearty thanks to my advisor, Dr. Hüseyin PEHLIVAN, for much support, guidance, and patient reading of drafts, who had the patience to read my writing and help make sense of it. I am also grateful to the other members of my dissertation committee, Dr.Vasif V.Nabiyev, Dr. Mustafa ULUTAS, Dr. Bahar Karaoğlan, and Dr. Adem Sefa AKPINAR for valuable feedback and encouragement. I am very fortunate to have had such a distinguished committee.

Thank you to my wonderful friends who I have neglected this past year. My Special thanks are due to Fardad Teymori, Saeid Agahian and Amir Rahmanparast for continuing to pursue me with love and confidence. I'm truly blessed by your friendships.

I would like to thank my parents for their constant encouragement and wonderful emotional support. I owe a huge debt of thanks to my father-in law and mother-in-law, who were helping to do my research and to deal with bureaucracy and with their best wishes.

Finally, I most deeply wish to thank my wife Sahereh and my daughter Yağmur. Their constant love and caring are every reason for where I am and what I am. My gratitude and my love to them are beyond words. Without her continued support and patience, none of this would be possible. She was always there cheering me up and stood by me through the good times and bad, and truly has sacrificed more professionally and socially than I, and in many ways has done more in achieving this degree than I. To you I am eternally grateful.

Thank you.

Mir Mohammad Reza ALAVI MILANI

THESIS STATEMENT

I declare that, this PhD thesis, I have submitted with the title "Design and Applications of Grammar-Based Methodologies for Automatic Generation and Step-by-Step Solving of Mathematical Expressions" has been completed under the quidance of my PhD supervisor Assistance Prof. Dr. Hüseyin Pehlivan. All the data used in this thesis are obtained by simulation and experimental works done as parts of this work in our research labs. All referred information used in thesis has been indicated in the text and cited in reference list. I have obeyed all research and ethical rules during my research and I accept all responsibility if proven otherwise. 27.02.2015

Mir Mohammad Reza Alavi Milani

TABLE OF CONTENTS

Page No

SUMMARY IX

ÖZET X

LIST OF FIGURES XI

LIST OF TABLES XIII

LIST OF ABBREVIATIONS XVI

1. INTRODUCTION 1

2. REVIEW OF THE LITERATURE 4

3. GENERAL INFORMATION 8

3.1. MATHEMATICAL EXPRESSIONS 8

3.1.1. TYPES OF MATHEMATICAL EXPRESSIONS 8

3.1.2. SYNTAX VERSUS SEMANTICS 9

3.2. REPRESENTATION OF MATHEMATICAL EXPRESSIONS 10

3.2.1. EXPRESSION STRUCTURE AND TREES 10

3.2.2. EXPRESSION TREE 13

3.2.3. SIMPLIFIED STRUCTURE OF ALGEBRAIC EXPRESSIONS 14

3.2.4. PROGRAMMING LANGUAGE FOR MATHEMATICAL EXPRESSION 17

3.3. CONVERTING MATH EXPRESSIONS INTO TREE STRUCTURE 18

3.3.1. CONCEPTS OF TRANSLATION 18

3.3.2. STRUCTURE OF A COMPILER 21

3.4. CONVERTING MATHEMATICAL EXPRESSIONS INTO TREES 24

3.5. GRAMMARS FOR LANGUAGES 27

3.5.1. VOCABULARY 28

3.5.2. A HIERARCHY OF GRAMMARS 29

3.5.3. ISSUES WITH PARSING CONTEXT-FREE GRAMMARS 30

3.5.4. PARSING TECHNIQUES 35

3.6. PARSER GENERATOR TOOLS 39

3.6.1. YACC 40

3.6.1. JAVACC 41

3.7. LANGUAGES FOR MATHEMATICAL EXPRESSIONS 41

3.7.1. MATHEMATICAL PSEUDO-LANGUAGE (MPL) 42
3.7.2. MATHML 42
3.7.3. LATEX 43
3.8. AN EVALUATION OF MATHEMATICAL EXPRESSIONS 44
3.8.1 INSTANCEOF OPERATOR 45
3.8.2 INTERPRET() METHODS 46
3.8.3 VISITOR DESIGN PATTERN 47
4. STEP-BY-STEP SOLUTIONS FOR MATHEMATICAL EXPRESSIONS 50
4.1. A BNF GRAMMAR DEFINITION 51
4.2. GRAMMAR CONVERSION 52
4.3. DEFINITION OF SYNTAX CLASSES 54
4.4. PARSER CONSTRUCTION 55
4.4.1. TOKEN SPECIFICATION 56
4.4.2. PARSER DEFINITION 57
4.5. GENERATING ABSTRACT SYNTAX TREE (AST) 58
4.6. EVALUATION OF MATHEMATICAL EXPRESSIONS 60
4.6.1. DIRECT EVALUATION OF MATHEMATICAL EXPRESSIONS 60
4.6.2. EVALUATION OF MATHEMATICAL EXPRESSIONS USING AST 61
4.7. REPRESENTATION OF MATHEMATICAL EXPRESSIONS 62
4.7.1. HUMAN-READABLE FORMAT 63
4.7.1. CONVERTING SYNTAX TREE TO *LATEX* FORMAT 64
4.7.1. CONVERTING SYNTAX TREE TO *MATHML* FORMAT 65
4.8. AUTOMATIC SIMPLIFICATION 66
4.8.1. BASIC TRANSFORMATIONS FOR SIMPLIFICATIONS 66
4.8.2. COMPLEX TRANSFORMATIONS FOR SIMPLIFICATIONS 70
4.8.3. OTHER SIMPLIFICATIONS 84
4.9. CONTROLLING THE STEP-BY-STEP SOLUTION 84
4.9.1. EXPRESSION SIMPLIFICATION CONTROL 86
4.10. DETERMINING TYPES OF MATHEMATICAL EXPRESSIONS 87
4.11. APPLICATIONS OF THE METHODOLOGY 90
4.11.1. FUNCTIONS 90
4.11.2. EQUATIONS 95
4.11.3. POLYNOMIALS 98

4.11.4. DERIVATION 103
4.12. DISCUSSION 106
5. AUTOMATIC PRODUCTION OF MATHEMATICAL EXPRESSIONS 107
5.1. METHODS FOR GENERATING EXPRESSION 107
5.2. STATISTICAL SPACE ANALYSIS 111
5.3. TYPE-SPECIFIC GRAMMARS FOR EXPRESSIONS 113
5.3.1. GRAMMARS FOR POLYNOMIALS 114
5.4. RULE-ITERATED CONTEXT-FREE GRAMMAR 115
5.4.1. GRAMMAR MANIPULATION 117
5.4.2. CFG VERSUS RI-CFG 118
5.5. A METHODOLOGY FOR EXPRESSION GENERATION 119
5.5.1. RI-CFG-BASED PRODUCTION OF EXPRESSIONS 119
5.6. APPLICATIONS 123
5.6.1. POLYNOMIALS 123
5.6.1. INDETERMINATE LIMITS 125
6. CONCLUTION 0
7. REFERENCE 0
8. APPENDIIX 5
CURRICULM VITAE .....

PhD. Thesis

SUMMARY

DESIGN AND APPLICATIONS OF GRAMMAR-BASED METHODOLOGIES FOR AUTOMATIC GENERATION AND STEP-BY-STEP SOLVING OF MATHEMATICAL EXPRESSIONS

Mir Mohammad Reza ALAVI MILANI

Karadeniz Technical University
The Graduate School of Science
Computer Engineering Graduate Program
Supervisor: Assistant Prof. Dr. Hüseyin PEHLİVAN
2015, 136 Pages, 31 Appendix Pages

In this thesis we propose methodologies for implementation and operation of the CAS systems, where it is generally organized in two parts, namely the proposal of two grammar-based methodological approaches as for automatic solving of math problems and generating template-based mathematical expressions. The first part of the study presents a methodology for the step-by-step solution of problems, which can be incorporated into a CAS-like system. The aim is to show all the intermediate evaluation steps of mathematical expressions from the start to the end of the solution. Simplification is done by applying various transformations on original problem.

The other part of the study addresses the production of new questions using the template expressions that are derived from the solved questions or entered by the users. With the parametric determination of values for such limitations, some templates can be dynamically constructed for the automatic generation of mathematical expressions and represented implemented in the form of classes. The proposed system currently focuses on the solutions of various problems associated with the subject of derivative, equations, single variable polynomials, and operations on functions however; it can be easily extended to cover the other subjects of general mathematics.

Key Words: Computer Algebra System, CAS tools, step-by-step solution, Mathematical expression, Simplification, Automatic solving, Dynamic template, Automatic expression generation, Compiler-Compiler tools, Grammars, AST.

Doktora Tezi

ÖZET

MATEMATİKSEL İFADELERİN OTOMATİK ÜRETİMİ VE ADIM ADIM ÇÖZÜMÜ İÇİN DİL BİLGİSİ TABANLI METODOLOJİLERİN TASARIMI VE UYGULAMALARI

Mir Mohammad Reza Alavi Milani

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Yrd. Doç. Dr. Hüseyin PEHLİVAN
2015, 136 Sayfa, 31 Ek Sayfa

Bu tezde CAS sistemlerinin uygulanması ve işletilmesi için iki yöntem önerilmiştir, deyişle matematik problemleri otomatik çözmesi ve matematiksel ifadeleri şablona dayalı üreten iki gramer tabanlı metodoloji yaklaşımlar verilmiştir. Çalışmanın ilk bölümünde, problemlerin adım adım çözümü için bir CBS türü sistem sunulmuştur. Çalışmada amaçlanan girilen matematiksel ifadelerin çözümünde tüm ara aşamaları göstermektir. Sadeleştirme işlemi orijinal problem üzerinde çeşitli dönüşümler uygulanarak yapılır. BCS sistem işlemleri gerçekleştirilerek karmaşık soruların basitleştirilmiş versiyonu elde edilir.

Bu araştırmanın diğer kısmı, kullanıcılar tarafından girilen sorulardan veya elde girilen şablonları kullanarak yeni sorular üretimidir. Önerilen metot kullanılarak, bir ifade için genel bir şablon oluşulma ve uygulanmasında bazı kısıtlamalar yapılabilir. Bu tür sınırlamalar değerlerin parametrik kararlılığıyla, matematiksel ifadeleri için dinamik şablonları elde edilebilir. Bu tür şablonlar sınıflar şeklinde uygulanabilir. Önerilen sistem şu anda türev, denklem, tek değişkenli polinomlar, ve fonksiyonlar üzerinde işlemler konuları ile ilgili çeşitli problemlerin çözümleri üzerinde durulumuş; ancak, kolayca genel matematiğin diğer konuları kapsayacak şekilde genişletilebilir.

Anahtar Kelimeler: Bilgisayar Cebir Sistemi, BCS araçları, Adım-Adım çözüm, Matematiksel ifadesi, Sadeleştirme, Otomatik çözme, Dinamik şablon, Otomatik ifade üretim. Compiler-Compiler araçlar, Gramerler, AST.

LIST OF FIGURES

Page No

Figure 1 - The conventional expression tree of "a + b * c^d". 13

Figure 2 - Some conventional expression trees. 14

Figure 3 - Conventional and simplified expression for "*a*+*b*+*c*". 15

Figure 4 - Conventional and simplified expression for "a - 2*c". 16

Figure 5 - Conventional and simplified expression for "-*a* * *b* / *c*". 16

Figure 6 - Overview of Compiler systems 19

Figure 7 - Stages of compilers 19

Figure 8 - The stages of an interpreter 20

Figure 9 - Process of mixed method 21

Figure 10 - Relations between Stages of Compile 22

Figure 11 - The process of lexical analysis 23

Figure 12 - The process of syntax analysis 23

Figure 13 - Converting the input string into tokens 24

Figure 14 - Two purpose for structural analysis 25

Figure 15 - The parse tree of Table 2 25

Figure 16 - Another parse trees for "*a* + *b* + *c*". 26

Figure 17 - The syntax tree of Figure 15 26

Figure 18 - The steps to generate AST 27

Figure 19 - The Different parse for expression "7 - 2 * 5" 31

Figure 20 - Exact parse tree for the expression "7 - 2 * 5" 32

Figure 21 - Different parse trees for the expression "7 - 2 - 5" 33

Figure 22 - LR parser 39

Figure 23 - How Compiler-compiler works 40

Figure 24 - Parser generation with YACC 40

Figure 25 - Parser generation with JavaCC 41

Figure 26 - "*bx+c*" in MathML 43

Figure 27 - an example of a math expression in LaTex 43

Figure 28-Some important phases and steps of the proposed methodology 51

Figure 29 - The steps required to produce the AST for the mathematical grammar 59

Figure 30 - The graphical illustration of AST for the expression "3*x*+5" 60

Figure 31- The AST for the expression "(3*cos* (*x* + 1)) / 2". 62
Figure 32-Some transformations with basic simplification rules 70
Figure 33-The simplification of similar operands 71
Figure 34-Simplifying the expression "3+7+2-5" 71
Figure 35-AST of the expression "2+ *x* + 6 +*x*" 72
Figure 36-Binary tree to list of operands 72
Figure 37-Structural form of the expression "*x*+3+2*x*" 73
Figure 38 - Simplification of the expression "12/6" 74
Figure 39-The Tree and list view of "6*x*/12" 74
Figure 40-Simplification stages of the expression "35/30" 75
Figure 41-AST of *fraction* children of a *fraction* node 76
Figure 42-An example of recursive fractions 77
Figure 43-Simplifying exponential expressions 79
Figure 44- The product of some power expressions 80
Figure 45 - Simplification of a radical expression as dominator 81
Figure 46 -Equivalent expressions of different ASTs 83
Figure 47 - Object representation of the expression "*f(x) =*3*x*+2" 89
Figure 48- Detecting expression type 90
Figure 49 - Parse tree and object view for expression "$f(x) = (3x+sin(x))/ x^2$" 91
Figure 50 - AST for expression "$f(x) = 3x+sin(x) / x^2$, $f(5)$ =?" 92
Figure 51 - AST of $f(x) = 5x - cos(x)$, $g(x) =(x^2+3)/12$ 93
Figure 52 – The summation of two functions 94
Figure 53 - AST of the expression "3*x* + (4-*x*)*2 = 3-5*x*" 96
Figure 54 – The AST of the expression "$2+3x^2-x = 12+x^2$" 98
Figure 55 – The AST of a polynomial expression 99
Figure 56 - Array of monomials for a polynomial 99
Figure 57 – The GCF of polynomials 102
Figure 58– The binary tree of the expression "((*x+y*)/2)×(*a+b*)-12" 109
Figure 59- The methodological components for random tree-based generation of expressions . 109
Figure 60 - Possible AST with up to 2 levels 112
Figure 61-Possible AST with up to 3 levels 112
Figure 62 – The generating tree for the expression "12+63+47" 116
Figure 63 - Components of the proposed methodology 120
Figure 64 - Flowchart for obtaining Taylor series 30

LIST OF TABLES

Page No

Table 1 - Mathematical expression types 9

Table 2 - Input string and its token stream 24

Table 3 - An example for grammar structure of the mathematical expressions 28

Table 4 - An example grammar without LL(1) attributes 36

Table 5 - LL(1) parse table for the grammar given in Table 4 37

Table 6 - LL(1) parse process for input "(a + a)" using Table 5 38

Table 7 - A comparison of syntax tree evaluation approach 44

Table 8 - Evaluation with the operator 45

Table 9 - Addition of *eval* methods to syntax classes 46

Table 10 - The main class for interpretation methods 47

Table 11 - Adding *accept()* methods to syntax classes 48

Table 12 - Visitor interface and its implementation for syntax tree 49

Table 13 - Main evaluation class for the visitor design 49

Table 14 - An Extended-BNF grammar for mathematical expression 52

Table 15 - A LL(1) grammar for mathematical expressions 53

Table 16 - The syntax classes for some mathematical components 54

Table 17 - The definition of some syntax classes shown in Table 16 55

Table 18 - JavaCC token declarations of the terminals in the grammar Table 15 56

Table 19 - The usual and combined rules for some non-terminals 57

Table 20 - Sample JavaCC method definitions for LL(1) grammar rules 58

Table 21 - The Java statements added to produce AST 59

Table 22 - Some JavaCC functions for string-based evaluation 61

Table 23 - *Print* method added to classes to generate human-readable output 63

Table 24 - LaTex method to generate LaTex statements 64

Table 25 - MathML methods to generate *MathML* statements 65

Table 26 - MathML example of the expression "(3*cos*(*x*+1))/2" 66

Table 27- Simplifications performed in classes 68

Table 28 - An implementation of simplification methods 69

Table 29 - Simplifying similar operands for *Num* and *Var* in Plus 71

Table 30 - Collecting operands for similar operators 73
Table 31 – The method simplify() for division class 75
Table 32 - Changes in method *simplify()* for the *Divide* class 76
Table 33 - Modifications in class *Plus* 78
Table 34 - The *simplify()* method for the Power class 79
Table 35 - The transformation code for the product of some power expressions 80
Table 36 – The simplify method() for the Divide class 81
Table 37 - Expansion of the formula $(a+b)^2$ 82
Table 38 – Equivalence of mathematical expressions 83
Table 39 - The main loop for simplifications 86
Table 40 - Modifications for the *simplify()* method 86
Table 41 - Some of variables and their applications 87
Table 42 - The control of the simplification of *radical removal* using control flags 87
Table 43 - Some forms of covered expressions 88
Table 44 - Additional grammar rules for various expression types 88
Table 45 - Additional rules in JavaCC format 88
Table 46 - Object structure for supported mathematical expressions 89
Table 47 - Operations on functions 91
Table 48 – A grammar for functions with values 92
Table 49 - Evaluation of a function value 93
Table 50 - Evaluation of a function value 94
Table 51 - Grammar rules for first-degree equations 95
Table 52 - Equ class definition in java 96
Table 53 - Grammar rules for quadratic equations 97
Table 54 – The codes for extracting exponential value of monomials 100
Table 55 – The division of two monomials 100
Table 56 – The division of two polynomials 101
Table 57 – The polynomial division of the expression"$(x(2x+1)+3x^3)/(x+1)$" 101
Table 58 – The GCF of the expression" $35ab^2-75a^2b+15ab$" 103
Table 59 – The derivation() methods to get the derivation of an object 104
Table 60 – The *derivation()* method to get the derivation of an object 105
Table 61 – An algorithm for the random generation of an arithmetic expression 108
Table 62 – An algorithm for the random generation of polynomial expression 108
Table 63 - Grammar rules for randomly generating mathematical expressions 110

Table 64 – The random expression generating algorithm 110
Table 65 - An improved algorithm based on the one in Table 64 111
Table 66 – The numbers of possible ASTs with some different levels 113
Table 67 – A CFG grammar for first-degree equations 114
Table 68 – A CFG grammar for quadratic equations 114
Table 69 - A sample grammar with no specific control structures 115
Table 70 - A sample RI-CFG grammar 116
Table 71 – A sample derivation with the grammar in Table 70 116
Table 72 – An implementation for a generating function called repeater(). 117
Table 73 – An enhanced version of the grammar in Table 65 117
Table 74 – CFG and RI-CFG grammar for the language $a^n b^n$ 118
Table 75 – CFG and RI-CFG grammars for the language $a^{3n} b^{3n}$ 118
Table 76 – The grammar of first-degree equations in the CFG and RI-CFG notations 119
Table 77 - An example of RI-CFG grammar 120
Table 78 – A method, *generate(),* of *RICFG* class 121
Table 79 - Class definitions of templates 121
Table 80 – The definitions of some expression templates 122
Table 81 - The outputs of the templates in Table 80 122
Table 82 – A RI-CFG grammar for polynomial expressions 123
Table 83 - Code implementation of Table 82 123
Table 84 - A sample template for polynomials 124
Table 85 – A random polynomial generator using templates 124
Table 86 - Random quadratic equation using Table 85 124
Table 87 - An example of dividable polynomials using templates 125
Table 88 - An example of polynomials using templates 125
Table 89 – An example of the limit using Table 88 126
Table 90 - An example of an indeterminate limit expression 126
Table 91 - Symbolic calculation of some math questions 3
Table 92 - Time and space complexity 3
Table 93 – Evaluation of produced expressions 4

LIST OF ABBREVIATIONS

CAS	Computer Algebra System
JavaCC	Java Compiler Compiler
YACC	Yet Another Compiler-Compiler
BNF	Backus-Naur Form
CFG	Context-Free Grammar
GCD	Greatest Common Divisor
LCM	Least Common Multiple
LL	Left-to-right and Left-most production
LR	Leftmost and Rightmost Derivations
AST	Abstract Syntax Tree
MathML	Mathematical Markup Language
RI-CFG	Rule-Iterated Context-Free Grammar

1. INTRODUCTION

Programing in the computer science has very huge improvement in the past decades. Assembly, as the first programming language was a revolutionary to the computing world using computers. Although it was hard to write complicated programs, even today it is used as an intermediate language for high level programming languages. High level programming languages such as Java and C#, facilitate the use of commands which are needed to control the computer itself. However, for various usages in innovative areas such as modeling, mathematics, simulation, and communication, there are many programming environments which are usually specialized to particular tasks. The core system behind the standard tools provided by these environments is based upon mathematics and formulas.

Mathematics has an important root in human evolutionary history. Computers are very good at solving problems if right commands are used. The first generation of programming languages, namely assembly, did not have special statements to work with mathematical expressions except for four basic operations of add, subtract, multiply and divide. Although the basic operations can be used to work with the hardware, given high-level programming languages, more sophisticated expressions are supported and break down to those operations. The breaking-down process is generally controlled by compilers which translate all statements and expressions in a high-level programming language into low-level or machine code. Due to very effective outcomes of the process, many advanced languages and systems have been introduced.

Mathematics systems are categorized into two main groups such as numerical computation and symbolic computation. Although numeric computation is widely used, the output is always an approximated value. Unfortunately the difference between the exact value and the approximate value can cause errors and failures. Error propagation happens in various forms such as floating point calculation, estimated values, uncertainty, etc. For very accurate systems, using numeric computation is not preferred.

Symbolic computation or algebraic computation refers to the development and evaluation of mathematical expressions. Symbolic computation offers exact solutions with expressions that can contain variables or symbols. If symbols are not set to any value, then the output will also be another expression of variables. Computer applications which can conduct symbolic calculations are called Computer Algebra Systems (CAS) or symbol

manipulation systems. The increasing use of CAS systems has enlarged the rule of computer systems in the teaching of mathematics. For example, computer algebra has an important role in designing and experimenting formulas that are required in numerical programs.

CAS can be used for two separate areas of general purposes and specific purposes. Programs such as Matlab, Maple, and Mathematica as general purpose applications provide rich computing facilities for complex and difficult general mathematical problems. Programs such as MyAlgebra, MathWay, webMath producing mathematical questions and DELIA working on deferential equations provide special uses of algebraic and general mathematics. But most applications of CAS are commercial, and there exist very few open source and academic projects. Academic researchers are conducted to increase the stock of knowledge in the scope of symbolic computation.

In this thesis we aim to propose two methodologies to solve and produce mathematical problems using symbolic computation. Symbolic solving of algebraic problems uses a divide and conquers strategy, where large expressions are divided into small pieces and each piece is separately handled to obtain the complete solution of the original problem. The solution of the problem pieces helps the production of intermediate or step-by-step solutions using the divide and conquers strategy, it would be possible to manage the evaluation, simplification and printing stages, and for example, required for the derivation of expressions, object orientation programming structures facilitate the implementation process of this strategy as a result of supporting the inheritance between classes.

On the other hand, producing mathematical questions is an important topic in learning systems. Learners need to practice and solve various mathematic problems to improve their solving skills. Obstacles such as limitations in resources and expenses can impact learning factors. This thesis proposes a mathematic question generator based on various types of expressions, with the aim of generating unlimited variety of questions for a topic such as first-degree equations, polynomials and etc.

Context Free Grammars (CFG) is used to parse and produce mathematical expressions. Input string passes through a parser which it verifies the syntax of that input. At the same time, an Abstract Syntax Tree (AST), which is a suitable structure to work with mathematical expressions, is created. This tree serves as the main core part of the proposed methodology to apply various improvement algorithms on input string.

However, in the productions of mathematical problems, because of some limitations, new syntax of grammar with modified rules is proposed. This grammar is similar to CFG, but it empowers the producer to gain a control over generation of rules for different expressions. Our work mainly focuses on generating mathematical issues with special formats and generating production templates. Produce of expression in proposed method is not completely random, and is based on the defined subject.

It is possible to create a symbolic computation environment via the proposed methodologies, where a mathematical problem can be generated and its step-by-step solution can be obtained. One of major benefits of these systems is that students can access many examples of similarly solved problems in an appropriate way without spending time and cost, particularly in order to comprehend a mathematical topic well.

In this thesis, algebraic expressions and the related developments in symbolic computation is studied in Section 2. Grammars, parsing techniques, and evaluation of expressions are discussed in Section 3. Solving and production methodologies illustrated with some particular applications are proposed in Section 4 and 5 respectively. Finally, a conclusion of the thesis is presented in Section 6.

2. REVIEW OF THE LITERATURE

From the beginning of computer science on, many algorithms, methods, and techniques have being developed. With the development of computer systems and their increasing use, it has been easier to see the effects of the technology on several different fields including education and health. In recent years, there have inevitably been many technological changes on educational practices and materials. Among educational disciplines, mathematical education, especially general mathematics, has attracted more attention [1].

Most of the current e-learning systems such as Coursera [2] and Edventure [3] have provided support for automatic assessment, but only with multiple choice questions as it is a great challenge to support other question formats. Recently, there has been a growing interest in intelligent tutoring systems for supporting mathematical learning such as geometry construction [4], algebra problem generation [5] and automatic solution assessment [6,7].

Currently, there are three main approaches for automatic mathematical solution assessment, namely computer algebraic approach, rule-based approach and structural approach. The computer algebraic approach is based on symbolic algebraic computation [8, 9] for automatic mathematical solution assessment. This technique is implemented in commercial Computer Algebra Systems (CAS) such as Mathematica [10] and Maple [11]. The rule-based approach [12, 13] is based on mathematical rules for equivalence assessment. Starting with a mathematical expression, this approach transforms it into another equivalent expression and compares it with the intended target expression for equivalence verification. The structural approach [14, 15] is based on the tree representation of a mathematical expression. In this approach, two mathematical expressions are first represented as two mathematical expression trees. Then, tree matching algorithm [16] is used to compare the two trees using dynamic programming [17] for equivalence verification.

The development of CAS has led to the idea of use of these systems in teaching mathematics. However, there are different discussions about the use of these systems in mathematical curricula [18]. The application of CAS systems to teaching programs has been approached on from two perspectives: The first approach focuses on the use of CAS

systems as part of mathematical teaching programs like instructing materials including solving equations, derivate, etc. [19-22]. The other approach examines the development of teaching programs based on technology, which is changing the course of teaching methods and structures [23-26].

In recent years, different researches on the impacts of use of CAS systems on improvement in mathematical teachings have been conducted, showing their favorable impact [27-30]. Also, studies on the use of technological educational systems for teaching the derivative subject have been performed [20, 21, 30, and 31]. The efforts made on the use of CAS systems in mathematical teachings have usually been in the form of non-numerical solutions and, in some cases, graphical representations, where the results suggest improvement in educational quality while utilizing these systems.

In 1986, the paper entitled "Computer Algebra System, Tools for Reforming Calculus Instruction", [32] addressed the use of CAS in development of conceptual comprehension, teaching method, wrong conception analysis, exercises, test questions, and finally the overcoming of restrictions leading to deficiency in algebraic operations.

Automatic mathematical solution assessment is different from other similarity-based solution assessment problems such as automatic essay grading [33] and short text marking [34].

It can be argued that the problem-based learning is more effective than traditional learning, and increases students' ability to solve the problems [35]. On the other hands textbooks cannot be suitable as a source of questions, because they are limited, and not interactive. In addition, textbooks do not usually solve the problem step-by-step and do not have appropriate visual features for motivation. In these resources, often there are not adequate facilities for on line helps to students. Therefore, the use of information and computer-based technologies in education, especially physics (e.g., CAPA) [36], mathematics (e.g., Mathway) [37, 38] and electronics (e.g., CHARLIE) [39] is widespread. The literature also has systems that emerged for generating and solving the questions [40, 41].

Many methods have been developed for solving algebraic equations and some of them have been used with automatic computers [42, 43]. The methods which are most suitable for use with automatic computers are ones that are applied to a wide class of equations and that are relatively rapid when the degree of the equation is large.

In addition to general-purpose computer algebra systems, there is some special-purpose software for mathematics and physics, developed to solve problems in a particular area. Typical examples are Cayley and GAP software, developed for group theory [44, 45], PARI, SIMATH, and KANT for number theory, CoCoA for commutative algebra [46], Macaulay for algebraic geometry and LiE for Lie theory [47].

Math software tools used in mathematics curriculum can serve in the five categories as: practical, public, private, environment, and communications [48]. Besides, Handal and Herrington (2003) [49] have identified exercises, lessons, games, simulations, hypermedia, and tools as other computer-based mathematics education categories.

In addition to solving problems, different systems are developed for expression generation using templates in Natural Language Generation (NLG). YAG [50] produces Template Based strings as real time and general-purpose. D2S (Data-to-Speech) [51] has been developed for different applications such as rout description, music, soccer report, and also for different languages including English. EXEMPLARS [52] is an object oriented, rule-based framework which supports dynamic text generator, and is a superset for JAVA, can be used as templates of HTML/SGML. XtraGen [53] is XML and JAVA-based software system for NLG which can be easily integrated with other applications.

Randomness is an interesting topic for scholars, since past times. There are many methods proposed for random numbers generation [54-57]. Random numbers have different applications in scientific areas such as cryptography [58-60], Computer Simulation [61,62], and even Animal Sciences [63]. Because of the importance of random numbers generation, some other methods have been proposed to produce random and pseudorandom numbers, such as ones that using chaotic functions [64] or electronic noises [65]. Besides, random generation is of great interest in other fields including E.N.GILBERT [66] studies which are concerned with random graphs and related probabilities. Random production is also considered in Natural Language Processing (NLP), resulting in different systems developed under the title of NLG, as measures taken by Langkilde [60], using stochastic techniques for NLG. These systems can be divided into two categories as real and template based ones. Kees Van Deemter [67] has compared these categories.

Tillman Bechar [68] presented a generation method for template-based NLG, using TAG. He proceeded on random language generation through integrating Basic Tree Nodes. Of course, template random generation is not limited to string random generation. It has

also other applications; for example, Amruth N. Kumar [69] used templates to produce problems and mainly programs. Test case generation is another application of random generation. Takahide Y. et.al [70] designed a tool for Just-In-Time (JIT) compilers as runtime test. The issue of random generation was also presented in designing automatic tests. Various systems were generated to help teachers generate question files in the internet [71-73]. In [74], Joao et.al generated a system to produce automatic mathematical tests with simple answers in which some structures have been designed to generate tests. In [75], Ana Paula designed a system for automatic generation of mathematics exercises based on Constraining Logic Programming (CLP). Such systems provide facilities for automatic generation of tests in environments such as Internet and virtual training systems.

The Microsoft MathWorksheet Generator automatically produce mathematical problem. This approach is limited to simpler algebraic domains such as counting, or linear and quadratic equation solving. Also, each domain has its own set of features that needs to be programmed separately. In this thesis a grammar-based methodology proposed for generate mathematical problems that could be used more widely than ones.

3. GENERAL INFORMATION

3.1. Mathematical Expressions

A mathematical expression is a set of mathematical symbols such as numbers, constants, variables, operations, functions, and other syntactic elements. Each mathematical expression can be viewed as the symbol of an operator followed by a sequence of operands. In computer algebra systems, this representation of the expressions is very flexible. An equation is an expression with an "=" symbol. A matrix is represented as an expression with an operator "*matrix*" and its rows as operands.

Computer programs can also be represented as expressions with operators such as "*procedure*" with at least two operands, the list of parameters and the body. In this way, a mathematical expression itself can be viewed as a program. For example, the expression "$a + b$" is a program which performs the sum of parameters a and b.

3.1.1. Types of Mathematical Expressions

There are various types of mathematical expressions. Each element in a mathematical expression can change the type of that expression. A list of all such elements [76] is presented in Table 1.

In this thesis, we will focus on algebraic expressions which consist of constant, variable, elementary arithmetic operation, factorial, integer exponent, n^{th} root, and rational exponent due to being all non-closed forms.

Table 1 - Mathematical expression types

	Arithmetic expressions	Polynomial expression	Algebraic expressions	Closed-form expressions	Analytical expressions	Mathematical expressions
Constant	Y	Y	Y	Y	Y	Y
Variable	Y	Y	Y	Y	Y	Y
Elementary arithmetic operation	Y	Y	Y	Y	Y	Y
Factorial	Y	Y	Y	Y	Y	Y
Integer exponent	N	Y	Y	Y	Y	Y
N^{th} root	N	N	Y	Y	Y	Y
Rational exponent	N	N	Y	Y	Y	Y
Irrational exponent	N	N	N	Y	Y	Y
Logarithm	N	N	N	Y	Y	Y
Trigonometric function	N	N	N	Y	Y	Y
Inverse trigonometric function	N	N	N	Y	Y	Y
Hyperbolic function	N	N	N	Y	Y	Y
Inverse hyperbolic function	N	N	N	Y	Y	Y
Gamma function	N	N	N	N	Y	Y
Bessel function	N	N	N	N	Y	Y
Special function	N	N	N	N	Y	Y
Continued fraction	N	N	N	N	Y	Y
Infinite series	N	N	N	N	Y	Y
Formal power series	N	N	N	N	N	Y
Differential	N	N	N	N	N	Y
Limit	N	N	N	N	N	Y
Integral	N	N	N	N	N	Y

3.1.2. Syntax versus Semantics

Being an expression is a syntactic concept. Mathematical expressions must be well-formed. Each operator must have a certain number of operators as operand. Strings of symbols that violate the rules of syntax have no well-forms and so are not valid mathematical expressions. Typical examples are "*x", "1) +2", etc.

The meaning of expression is the semantic of that expression. Some expressions can be syntactically correct, but semantically incorrect. For example the expression in "x / 0" has the correct syntax, but it is not accepted in semantic point of view since nothing can be divided by zero.

In a programming language, an expression can be computed to produce a value, which is called the result of the evaluation. The evaluation of an expression depends on the semantics of the mathematical operators used in that expression. Also, the type varies of the result from numeric values to strings or logical values. For example, a programming language evaluates the expression 2+3 to 5, and evaluates "*This*" + "*Message*" to "*ThisMessage*".

Many mathematical expressions include variables. A variable is a placeholder to denote a value in memory. For example, the expression "y+6", evaluated for "y = 1" will give "7".

3.2. Representation of Mathematical Expressions

Mathematical expressions can be represented in various types of data such as String, List and Tree. This section provides a short overview of representation forms of expressions.

- *String*

Mathematical expressions are normally typed view in string form such as "x+2", "$7x^3$ - x", and "*sin* (x + y)". The form of string representation is known as infix notation. Although this notation is so common in mathematics because of its easy readability, it is not suitable for computational purposes. There are two other alternative notations known as postfix and prefix ones. For example, the expression "x + 2" is written as "+ x 2" in prefix notation and "x 2 +" in postfix notation.

- *List*

In some cases, it is useful to get mathematical expression in a list. Polynomials are such expressions which suit well to the representation with list structure, where an element and its index in the list correspond to the coefficient and degree of the related term in the polynomial, respectively. For example, the polynomial "$7x^3$ - x" can be represented as a list of "[7, 0, -1, 0] ".

- *Tree*

Tree is the most suitable structure to work with mathematical expressions in computer science. It provides a hierarchy view of the expressions. In the following sections, we will go into the details of tree representation of mathematical expressions.

3.2.1. Expression Structure and Trees

As we focus on algebraic expressions in this thesis, let us first discuss the structure of algebraic expressions that have hierarchical nature and recursive forms.

3.2.1.1. Algebraic Expressions

An algebraic expression u should fall into one of these rules:

- *u* is a number
- *u* is a symbol
- *u* is a *sum, product,* or a *function, whose operands are another algebraic expression.*

Rule 3 makes the very definition for a recursive algebraic expression, because it requires each of its operands to be an algebraic expression as well. Algebraic expressions can be manipulated by the transformation rules of preliminary algebra. Here are some examples for demonstrating the correct form of an algebraic expression:

1, 1 + 2 / 3, $cos\ (x^2)$, $g\ (f\ (x, y))$, *true*

Hence, the following examples do not follow the above rules:

$[x, y, z]$, $x = 1$, *true* or *false*

3.2.1.2. Operator Classification

According to the place of an operand located within parentheses, we will have different parentheses levels and same parentheses level. Given the expression "(a + b) / c", the operators + and / reside at different parentheses levels. Another expression example is "$a/(b\text{-}c)*d$", where / and * are at the same level, while - is at a different level from both / or *.

Operators can also be classified based on the number and the location of the operands. Some classifications are listed below:

- *A unary prefix operator, as - in -x*
- *A unary postfix operator, as "!" in n!*
- *Function prefix operator, as f in f(x, y)*
- *Binary infix operator, as ^ in x^y*
- *An n-ary infix operator, as * in* 2*x*y

3.2.1.3. Conventional Structure of Algebraic Expressions

The conventional structure in an expression is similar to both mathematical and conventional programming languages structure. The following rules describe the conventional structure of an expression.

(a) Structural assumptions:

If u is an algebraic expression, the algebraic operators in u satisfy these assumptions:

- *The operators + and - are either unary prefix or binary infix operators.*
- *The operators *, /, and ^ are binary infix operators.*
- *The operator ! is a unary postfix operator.*

(b) Conventional precedence rules

The relationship between operators and operands in algebraic expression u is defined through the following rules:

- *The precedence of operators in u at the same parentheses level and its precedence hierarchy is as follows :*

  function names, !, ^, * /, + -

  these rules are also applied to the precedence table:

  A. *Unary + and - operators has higher precedence in comparison to * and /.*
  B. *For the operators +,-,*,/,! at the same parentheses level, the operator to the left has a higher precedence.*
  C. *For the operators ^, the operator to the left has a higher priority.*

- *For the operators at different parentheses levels, the operator inside of a parenthesis has a higher precedence than the one outside of those parentheses.*

Here are some examples for these rules:

$a + b * c - d$ *, +, - equivalent to $(a + (b * c)) - d$

$a / b * c$ /, * equivalent to $(a / b)*c$

$2 * exp\,(a - b)$ $a - b$, *exp*, *

x^ 2^ 3^ 4   equivalent to $((x$ ^ 2) ^ 3) ^ 4

3.2.2. Expression Tree

The relationship between the operators and operands of an expression and its structure can be shown graphically by using expression trees.

Figure 1 shows the expression tree of "*a* + *b* **c* ^ *d*".

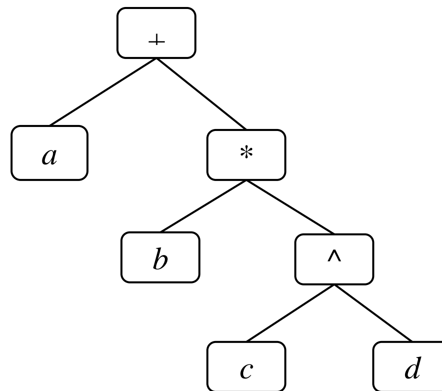
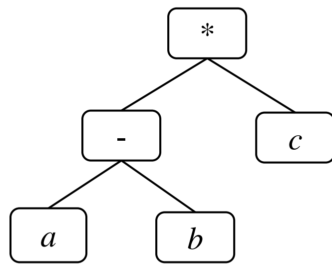


Figure 1 - The conventional expression tree of "a + b * c^d".

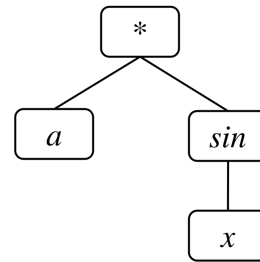
Each operator is represented by a node. Operators with the lowest precedence appear at the top of the tree, whereas the operators with higher precedence are put at the bottom of the tree.

Each level shows the connection between an operator and its operand or operands.

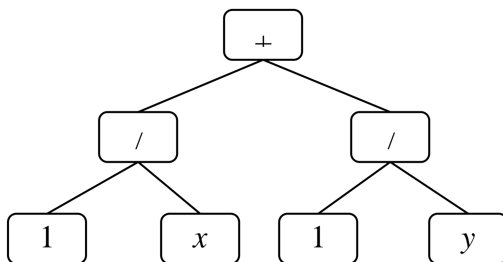
In Figure 2 more trees are shown for different expressions with different operands.



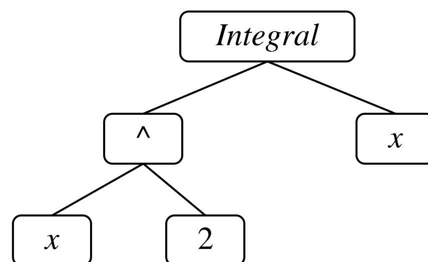
(a) The conventional expression tree for "$(a - b) * c$ ".



(b) The conventional expression tree for "$a * sin(x)$ ".



(c) The conventional expression tree for "$1/x + 1/y$".



(d) The conventional expression tree for "$Integral(x^2,x)$ ".

Figure 2 - Some conventional expression trees.

3.2.3. Simplified Structure of Algebraic Expressions

Mathematical expressions which are converted to trees can be easily processed for simplification and evaluation purpose. This form of structure simplifies the process of evaluating an expression by eliminating the some operators from the expression and therefore providing an easy access to the operands.

An expression u is an Automatically Simplified Algebraic Expression (ASAE), in case it follows one of the following rules:

1. u is an integer.
2. u is a fraction c / d where c and d are non-zero and integers.
3. u is a symbol.
4. u is *sum*, *product*, *power*, *fraction*, or *function*, where each operand of u is another ASAE.

Our intention is to simply describe the important properties of ASAE, although they have no by any means complete descriptions for these expressions.

a. Structural Assumptions:

If *u* is an automatically simplified algebraic expression, the operators in *u* must satisfy the following assumptions:

1. The operator + is an *n*-ray infix operator with two or more operands none of which is sum. And at most one operand of + is a number or fraction.

Figure 3 shows "*a* + *b* + *c*" in simplified form. Note that (*a* + *b*) + *c* is not in automatically simplified form as (*a* + *b*) is a sum operand for the latter +. For "+*a*", the simplified form is "*a*".

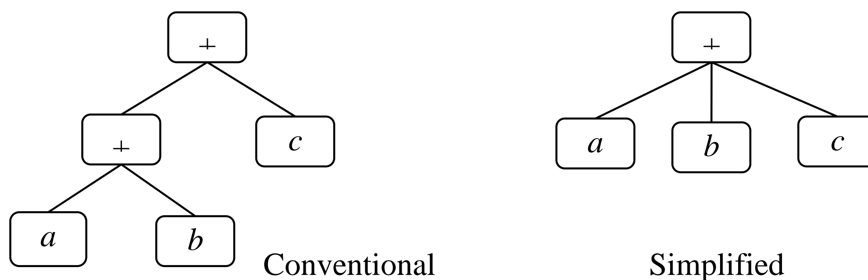


Figure 3 - Conventional and simplified expression for "*a*+*b*+*c*".

2. The unary operator - and binary operator will be replaced by ((-1) * ...).

This rule implies that the expression "-a" is not automatically simplified. The simplified form of this expression is "(-1)**a*" which is the product of an atomic number (-1) and another operand. In a similar way, the simplified form of "*a*- *b* * *c*" is "*a* +(-1)**b***c*".

3. The operator * is a unary infix operator with two or more operands which are a product. At most one operand of * is a number or fraction. The number or fraction operand of * should be the first operand.

Figure 4 shows "*a* - *b* * *c*" and Figure 5 shows "-*a* * *b* / *c*" expressions in simplified forms. According to rule (1.2), operator - is converted to (-1) in both expressions, and as a number, it is the first operand of the product.

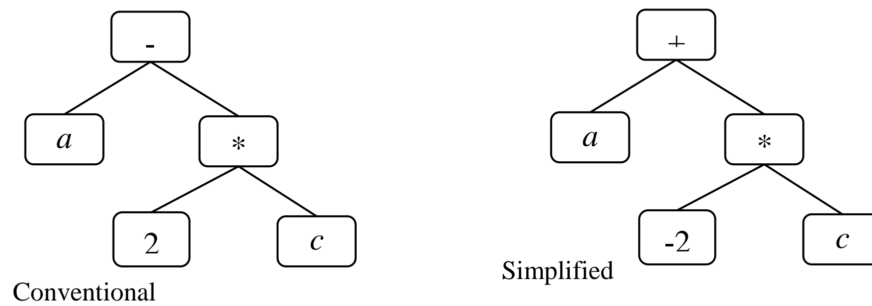


Figure 4 - Conventional and simplified expression for "a - 2*c".

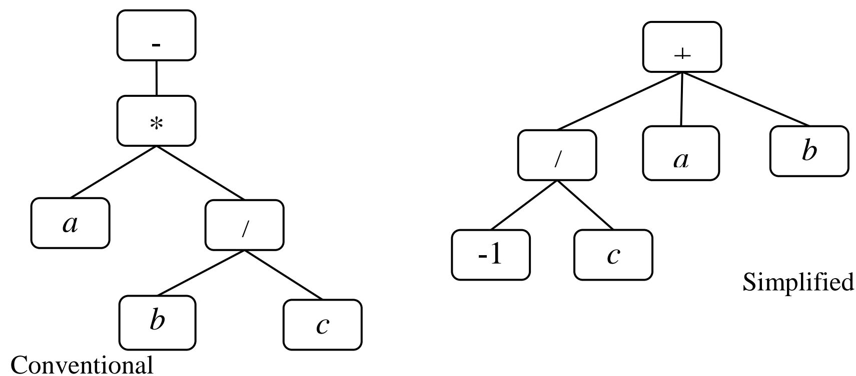


Figure 5 - Conventional and simplified expression for "-*a* * *b* / *c*".

4. Numerical fractions satisfy:
   i. *A negative fraction has a negative numerator and positive denominator.*
   ii. *Quotient that represent fractions a/b, where a and b are non-zero numbers, will be represented as "a frac b" by an expression tree.*

Rule 4.ii implies that expression -2 / 3 will be represented as "-2 *frac* 3" instead of a quotient.

5. The operator / does not appear in simplified expressions. Simplified quotients are in forms of products, powers, and a form of fractions.

Simplified form for "1 / *a*" is "*a* ^ -1" and the simplified form for "*a* / *b*" is "*a***b*^-1".

6. In the power operator, ^, if *u^n*, where n is a number, then *u* cannot be an integer, product, fraction, or power.

"(*a* * *b*) ^ 3" has the simplified form of (*a* ^ 3) * (*b* ^ 3). For "*a* ^ 2 ^ 3", and "3 ^ -1" it is "*a* ^ 8" and "1 / 3", respectively.

7. For unary postfix operator "!", the operand is a non-negative integer number. This rule implies that the simplified form for 4! is 24.

b. Simplified precedence rules:

The precedence of operators in *u* at the same parentheses level is in the order of the following:

function names, *frac*, !, ^, *, +

The location precedence for ! and ^ are the same as conventional precedence rules.

Note that for + and *, there is no need to account for location precedence since all operators are the same. Also the operator *frac* has higher precedence than !, ^, *, and +.

3.2.4. Programming Language for Mathematical Expression

In all programming languages, it is possible to implement a structure to manipulate and work with mathematical expressions. However, these structures are not part of the language such as Java and C#. It should be implemented as a library for the language.

C# has Expression data type to work with mathematical expressions. An example of usage of this data type is shown below:

```
Expression e =
      new Expression("Round(Pow([Pi], 2) +
       Pow([Pi2], 2) + [X], 2)");

e.Parameters["Pi2"] = new Expression("Pi * [Pi]");
e.Parameters["X"] = 10;

e.EvaluateParameter += delegate(string name,
    ParameterArgs args) {
 if (name == "Pi") args.Result = 3.14;
}

Debug.Assert(117.07 == e.Evaluate());
```

The methodology that is proposed in this thesis consists of several steps. These steps convert the input string into parse tree and then evaluate it. However, one can ignore the initial steps and use the parse steps alone.

3.3. Converting Math Expressions into Tree Structure

Converting a string into another model or data structure requires a converting process called translation. In an input string representing a mathematical expression, each operator has a rule that indicates the number of operands and the precedence of that operator to others. Such a translator which operates on mathematical expressions should keep these rules in converted model, too. In this section the structure and parts of such a translator are discussed.

3.3.1. Concepts of Translation

In computer systems, translation, evaluation, and execution of the program perform in one of the following ways:

- *Natively execution:* the program in a higher level language translates into CPU commands. These commands may vary from a model of CPU to another. CPU picks the commands in its machine language and simply executes them. As this method directly runs instructions by CPU, it is fast.
- *Execute using Interpreter:* first, the user program written in a higher level language translates into a middle language. In execution time, interpreter use the commands in this middle language and translates them into machine language. This method makes the program independent from a specific machine language and thus it can run on any CPU as long as there is an interpreter for it. However, because commands must be translated every time, it is slower than a compiled program.

3.3.1.1. Compiler

A compiler is a program that accepts the typed text of a program. As input data the program is based on the compiler rules and structure. The compiler creates a program in a second language as output called machine language. Figure 6 shows how compilers work.

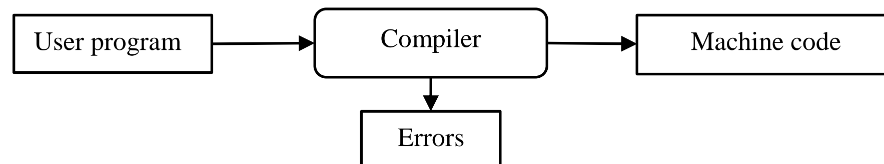


Figure 6 - Overview of Compiler systems

As mentioned before, the great advantage of compiling a code is that the execution time is very quick. However, the machine code is not readable by human being as it is in direct CPU commands. For this reason, the changes in a user program should be re-compiled to take effect.

A compiler uses several different stages to produce machine codes. Figure 7 shows these stages.

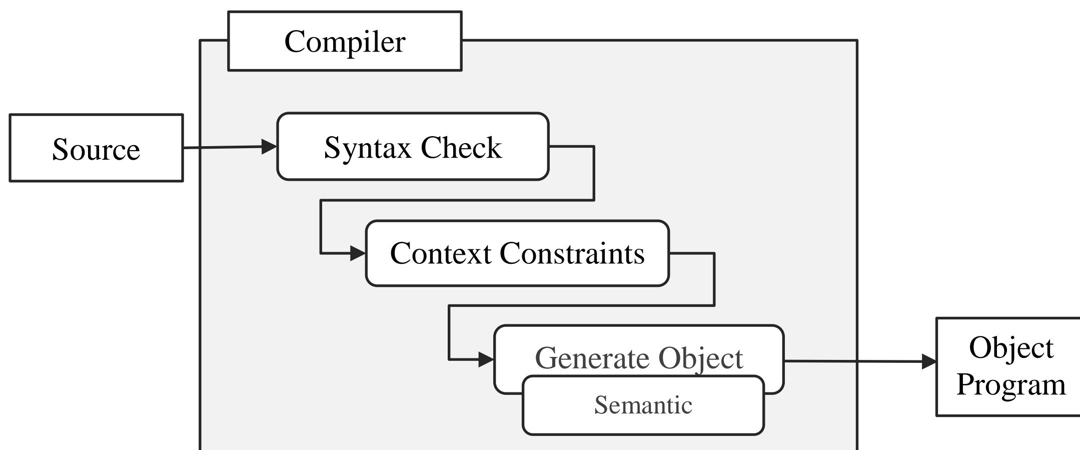


Figure 7 - Stages of compilers

3.3.1.2. Interpreter

Interpreters translate each command in user’s program on the fly into CPU commands. The advantage of this method is its simplicity. In addition, the user code can be

changed during the execution time. The deficiency of this method is that every time the program starts running, all the lines should be translated again, which makes the interpreters slower to execute. As an example, the BASIC language in primary computers was an interpreter. Other examples are JavaScript, Lisp and Forth.

The stages and their relations in an interpreter are shown in Figure 8.

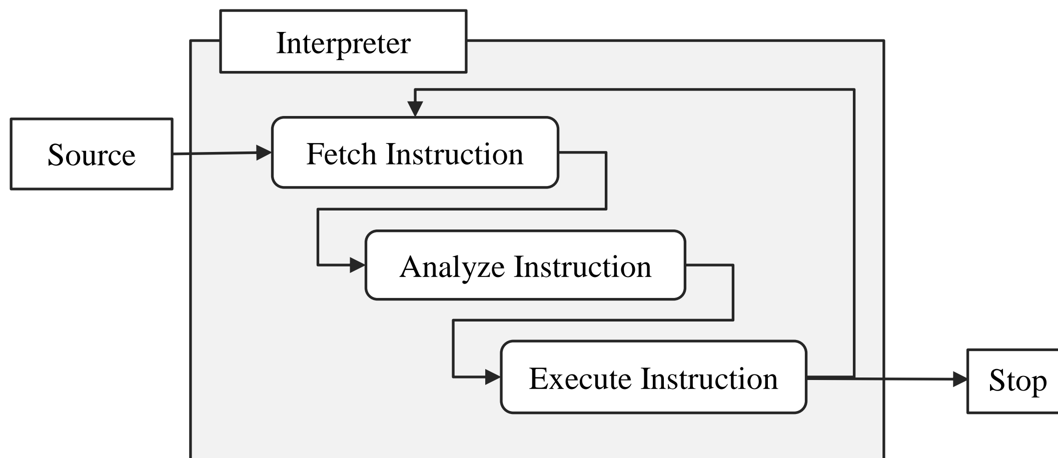


Figure 8 - The stages of an interpreter

3.3.1.3. Mixed

It is possible to use both interpreter and compiler as a mixed system. An example of such system is Java Virtual Machine (JVM) and Java bytecodes. Source files in java language first compiles into classes and java bytecodes. Then these codes will be run in the host computer over JVM which simply interprets the bytecodes. Figure 9 shows the process of executing a Java program. This method of mixed model will make the compiled code independent from the hosted operation system.

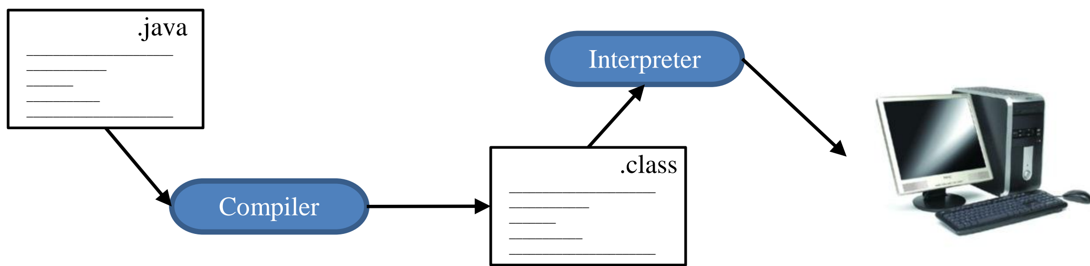


Figure 9 - Process of mixed method

3.3.2. Structure of a Compiler

The compilation operation includes the following six stages:

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generation

Beside the six fundamental stages of a compiler, there are two other parts, the Error Handler and Symbol table. The relation between these stages is shown in Figure 10.

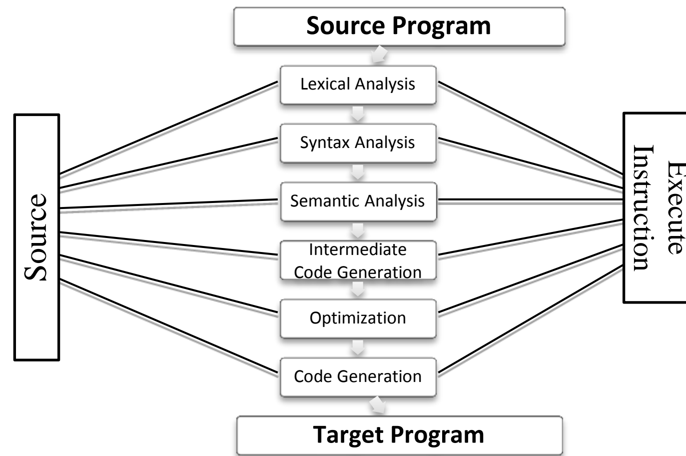


Figure 10 - Relations between Stages of Compile

The first four stages and a part of the optimization stage are dependent on source program but independent from machine, which is called Front End. The remaining part of the optimization stage, and the last stage which are dependent on target machine, is called Back End.

3.3.2.1. Lexical Analysis

The task of a lexical analyzer is to read the plain text from user program and converts it into series of tokens. Each programming language has their own forms of various words that can be used in that language. Lexical analysis reports a lexical error in user program if there is unknown word within the input text. The output of lexical analyzer in stream of tokens is delivered to the syntax analyzer as shown in Figure 11.

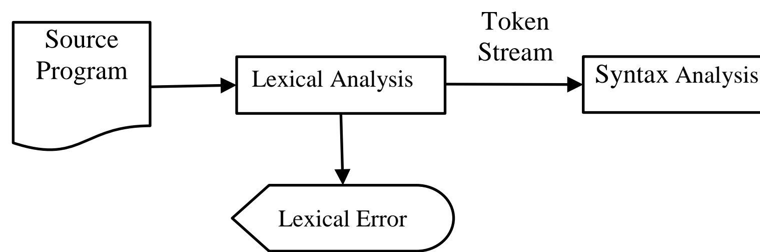


Figure 11 - The process of lexical analysis

3.3.2.2. Syntax Analysis

The task of a syntax analyzer is to map the stream of tokens from lexical analyzer into syntax of the language. Syntactic rules define the structure of a language. Therefore, the order and usage of tokens can be checked by the syntax analyzer. If a particular token is not matched by the language rules, then an error is reported. Figure 12 shows how this step works.

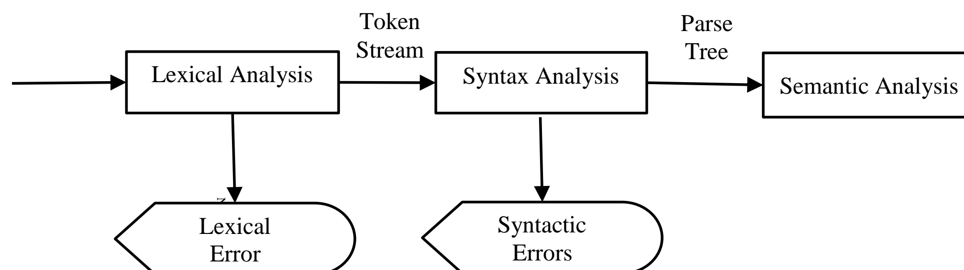


Figure 12 - The process of syntax analysis

3.3.2.3. Semantic Analysis

It is possible for a sentence to be syntactically correct and semantically incorrect. For example, “x=8 / 0” syntactically is correct but, semantically is not, due to division by zero. In most compilers semantic analysis is restricted to type checking. For example, an integer cannot be compared to a string.

As our goal in this thesis is to convert mathematical expressions from string form to a tree, due to nature of mathematical expressions, we do not need to check the data types

via a semantic analyzer. We do not also consider reviewing the remaining steps of a compiler here.

3.4. Converting Mathematical Expressions into Trees

The string form of mathematical expressions requires to be converted into mathematical tokens by a lexical analyzer. An example is shown below to demonstrate the input string and its token stream.

Table 2 - Input string and its token stream

Input string	Token Stream
$3x^2$+2(*x*-12)	Num(3) Var Power Num(2) Plus Num(2) LPR Var Minus Num(12) RPR

In Table 2 the terms used to describe the token stream are for presentation purposes.

The process of converting the input string into tokens is shown in Figure 13. Hereafter, we will use the term *scanner* for lexical analyzer.

Figure 13 - Converting the input string into tokens

The token stream should be handled by mathematical rules, which check if for example parentheses are in pair, and each operator has enough operand. These rules will be applied by a mathematical grammar, which will be discussed in the next section.
The structural analysis can be done for two main purposes.

a. *Checking for correct structure*
b. *Producing the parse tree*

The token stream will be analyzed whether it satisfies all rules defined by a mathematical grammar. According to required, the syntax analyzer can report true or false

for correct structure of the input stream, or generate a parse tree as an input to the next steps, which is shown in Figure 14.

(a) Checking for Correct Structure

(b) Producing the parse tree

Figure 14 - Two purpose for structural analysis

The parse tree is generated by a unit called parser. For incorrect token stream an error will be reported. Figure 15 shows the parse tree of Table 2.

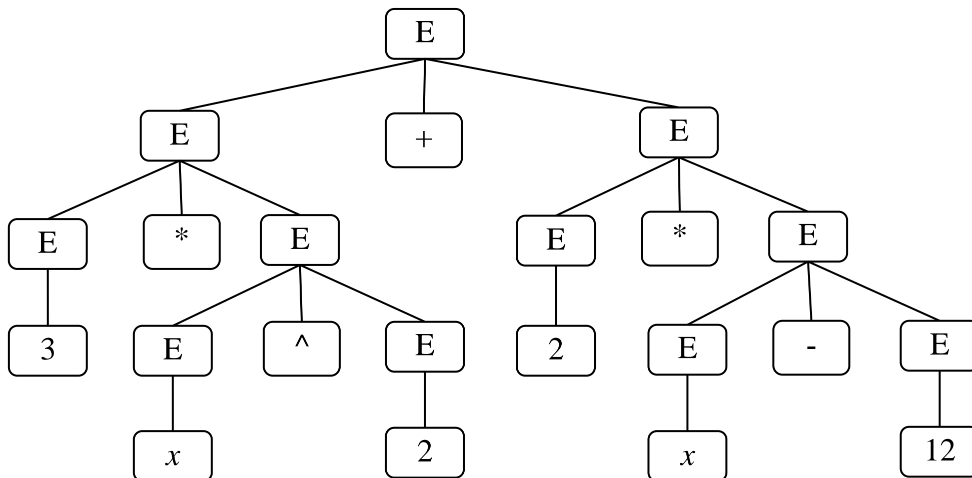


Figure 15 - The parse tree of Table 2

Depending on the grammar, there can be different parse trees. Figure 16 shows two another parse trees for "$a + b + c$".

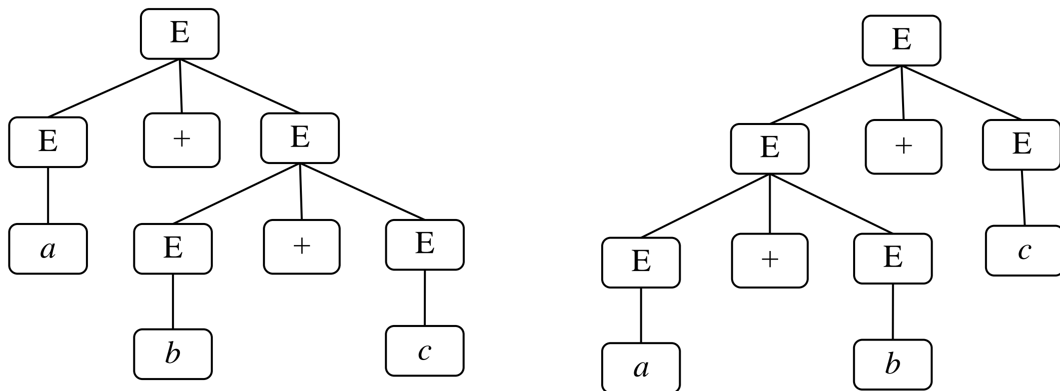


Figure 16 - Another parse trees for "$a + b + c$".

The leaves in parse tree are terminals, and other nodes are non-terminals. However, it is appropriate to use another kind of tree called syntax tree. The main difference is that leaves are operands and nodes are operators. Besides it offers a simple and efficient tree to work with mathematical expressions. Figure 17 shows the syntax tree of Figure 15.

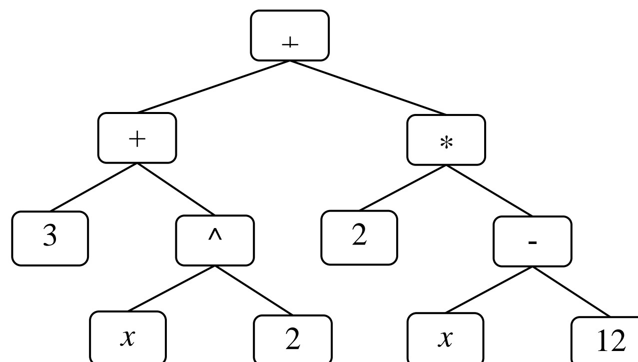


Figure 17 - The syntax tree of Figure 15

The syntax tree is also called Abstract Syntax Tree or shortly AST. Figure 18 shows the steps to generate AST from an input string as a mathematical expression.

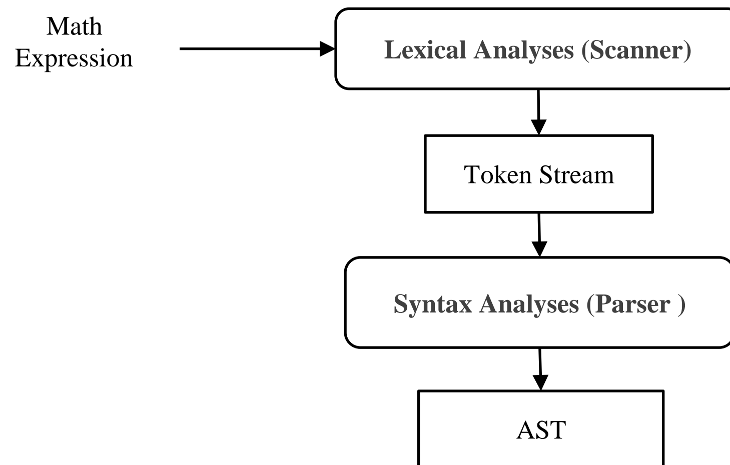


Figure 18 - The steps to generate AST

Type inconsistency does not occur in mathematical expressions. Therefore we will not use a semantic analyzer. However, we need to check division by zero in execution time for simplicity.

In this thesis, we generate AST and do not consider the remaining stages of a compiler. The next section discusses the grammars and parsing techniques.

3.5. Grammars for Languages

Backus Normal Form (BNF) or Meta Language of a specific language is determined by syntax rules or the grammar of a language. The C language is the first programming language that uses BNF to define its grammar. In this thesis we will focus on mathematical rules and introduce some grammars to work with them. Table 3 shows a trivial example of a math grammar.

The grammar in Table 3 consists of five symbols listed below:

1. Symbol → , which means "there is".
2. Symbol | , which means "or".
3. Symbol * , which means arbitrary number of elements.
4. Symbol ?, which means at most one element
5. Symbols (and) are usually used as separators.

Table 3 - An example for grammar structure of the mathematical expressions

$exp \rightarrow exp\ op\ exp \mid func(\ exp) \mid (exp) \mid Num \mid Id$

$op \rightarrow + \mid - \mid * \mid / \mid$ ^

$func \rightarrow Sin \mid Cos \mid Tan \mid Cot$

$Id \rightarrow x \mid y \mid z \mid a \mid b \mid c$

$Num \rightarrow [Digit]^*\ [.[Digit]^*]?$

$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid .. \mid 9$

Each rule has a non-terminal part on the left side and a set of rules on the right. During the language generation, depending on the top-down or bottom-up parsing method, a rule will be expanded or collapsed. We will discuss these two methods in the later sections. Using the rules or *productions* in Table 3, the following expressions can be obtained:

- $a + b - c$
- 12 * (Sin (x - 2) +6)
- $x + y$ ^ (6 - 2)

Here is the *leftmost derivation* of "$a + b - c$":

$$exp \rightarrow exp + exp \rightarrow exp + exp - exp \rightarrow a + exp - exp \rightarrow a + b - exp \rightarrow a + b - c$$

However, it is possible to generate semantically incorrect derivations like "a / (6 - 6) ", although it is syntactically correct. Grammars are a tool for syntax, not semantics. Fortunately the only semantic issue with mathematical expressions is division by zero, which can be controlled during the evaluation of the expression.

3.5.1. Vocabulary

We need to review some definitions before we can proceed:

- *Grammar is a set of rules that all valid sentences in a language are derived.*
- *Non-terminal is a symbol that specifies the production rule and can be replaced. Non-terminals cannot appear in output.*
- *Terminal is a symbol that cannot be replaced. Terminals appear in output.*

- *Production is a grammar rule that describes how to replace the symbols. The general form of a production for a non-terminal is:*

$X \rightarrow Y_1Y_2Y_3...Y_n$

Where non-terminal X is declared equivalent to the concatenation of $Y_1Y_2Y_3...Y_n$. Y can be a terminal or a non-terminal. This means wherever we encounter X, it will be replaced $Y_1Y_2Y_3...Y_n$.

- *Derivation is a set of productions that can produce a sentence based on that grammar. Derivation of rules is called parsing.*
- *Start symbol is a non-terminal that all sentences derive from:*

$S \rightarrow X_1X_2X_3...X_n$

Usually start symbol is shown by non-terminal S.

- *Null symbol sometimes is useful to indicate that a non-terminal produce nothing. To show this, we use symbol λ (Lamda):*

$A \rightarrow B \mid \lambda$

3.5.2. A Hierarchy of Grammars

There are four categories of formal grammars in the Chomsky Hierarchy introduced by Noam Chomsky. In this section, we briefly review these grammars.

- *Type 0 - Free or unrestricted grammars:* This type is the most general form. The form of productions is $u \rightarrow v$ where u and v are an arbitrary set of symbols in **V**. However, u cannot be null. There are no restrictions on what appears in u and v.
- *Type 1 - context-sensitive grammars:* The form of productions is $uXw \rightarrow uvw$ where u, v and w, are an arbitrary set of symbols in $\boldsymbol{V}$, v cannot be null, and X is a single non-terminal. In other words, X may be replaced by v only when it is surrounded by u and w.
- *Type 2 - context-free grammars:* The form of productions is $X \rightarrow v$ where v is an arbitrary set of symbols in $\boldsymbol{V}$, and X is a single non-terminal.

- *Type 3 - regular grammars:* The form of productions is $X \rightarrow a$, or $X \rightarrow aY$ where X and Y are single non-terminals, and a is a single terminal. This grammar is the most limited grammar in terms of expressive power.

Type 3 is a subset of type 2. Also type 2 is a type 1 and so on. Because of the lack of recursive constructs, type 3 grammars are particularly easy to parse. Efficient parsers exist for many classes of Type 2 grammars. Although Type 1 and Type 0 grammars are more powerful than Type 2 and 3, they are far less useful since we cannot create proper parsers for them.

In this thesis we will use context-free grammars or CFG. Rules in context-free grammar (CFG) are in the following form:

$$V \rightarrow w$$

Where V is a single non-terminal symbol and w is a string of terminals, non-terminals, or empty. A formal grammar is considered "context free" when the right hand side could be replaced with the non-terminal on the left side.

Definition: A context-free grammar is a quadruple G = (V, Σ, P, S), where

- V is a finite set of symbols called the vocabulary or set of grammar symbols;
- $\Sigma \subseteq V$ is a set of terminal symbols;
- $N = V - \Sigma$ is called the set of non-terminal symbols
- $S \in (V - \Sigma)$ is a designated symbol called the start symbol;
- $P \subseteq (V - \Sigma) \times V^*$ is a finite set of productions or rules;

$P \subseteq N \times V^*$, and each $\langle A, \alpha \rangle$ is also denoted as $A \rightarrow \alpha$. Production $A \rightarrow \lambda$ is called lambda rule, or null rule.

3.5.3. Issues with Parsing Context-Free Grammars

Dispute efficiencies in parsing most Type 2 grammars; however, there are some issues that can disturb the process of parsing: ambiguity, recursive rules, and left-factoring.

3.5.3.1. Ambiguity

If a grammar permits more than one parse tree for a sentence, it is said to be an *ambiguous grammar*. For example, consider the following grammar for simple arithmetic expressions:

S → E
E → E OP E | (E) | N
OP → + | - | * | /
N → 0 | 1 | 2 | ... | 9

This grammar is ambiguous because of some sentences that can have more than one parse tree. For example, consider the expression “7 - 2 * 5”. The parse trees of this expression are shown in Figure 19:

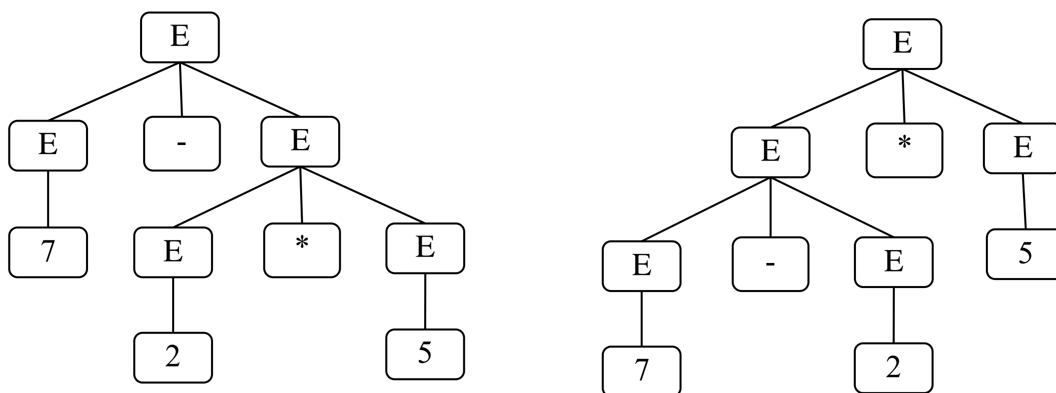


Figure 19 - The Different parse for expression "7 - 2 * 5"

Both trees are acceptable by the grammar, thus either one is valid. Although natural languages can tolerate some kind of ambiguity (puns, plays on words, etc.), it is not acceptable in programming languages, because the compiler randomly has to decide which way to parse the expressions. In this example, given our expectations from algebraic rules of precedence, only one of the trees seems correct.

It is fairly easy for a grammar to become ambiguous. Unfortunately, there is no such technique that can be used to resolve all varieties of ambiguity. However, it does not mean that ambiguity cannot be resolved.

An unambiguous grammar can be produced from ambiguous grammar or some additional disambiguating rules can be applied to remove undesirable parse trees to keep

only one tree for each input. For the grammar in Figure 19 one could provide additional codes into the parser and introduce the precedence to force the parser to build the correct tree. The advantage of this method is that the grammar remains simple and less complicated. However it will always requires this information alongside the grammar.

Another approach is to change the grammar to allow only one tree that correctly reflects our intention. For example, a grammar of simple arithmetic expressions can be like this:

S → E
E → *E A E* | *T*
A → + | -
T → *T B T* | *F*
B → * | /
F → (*E*) | *N*
N → *0* | *1* | *2* | ... | *9*

This grammar exactly follows the rules of operator precedence. For an example of the expression "7 - 2 * 5", there will be only one parse tree, which is shown in Figure 20.

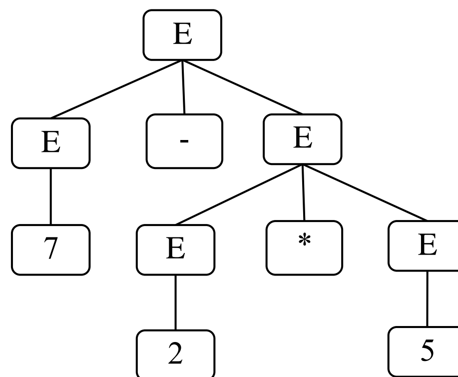


Figure 20 - Exact parse tree for the expression "7 - 2 * 5"

However, there is other ambiguity for this grammar. Figure 21 shows the parse tree for the expression "7 - 2 – 5"

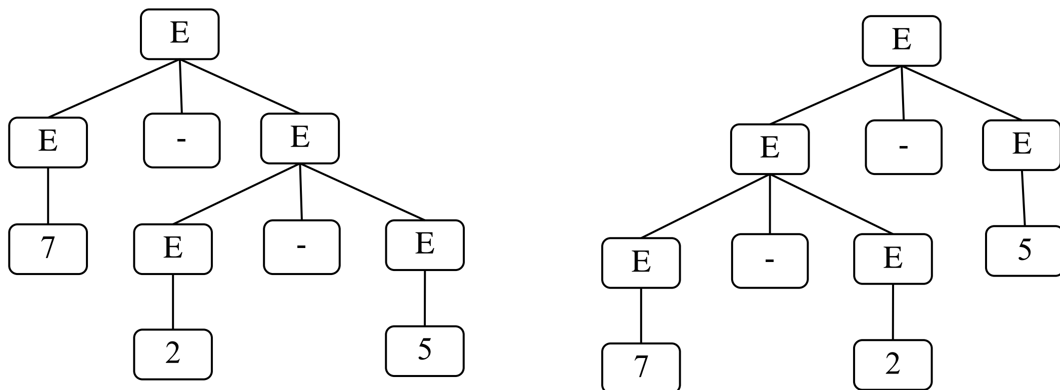


Figure 21 - Different parse trees for the expression "7 - 2 - 5"

It is possible to write another grammar to follow other algebraic rules to produce an unambiguous tree for "7 - 2 - 5". The obvious disadvantage of changing the grammar to remove ambiguity is that it may complicate and obscure the original definition of the grammar. Introducing an unambiguous grammar is not easy. It is also known to be an un-decidable problem. In fact, even determining ambiguity of a CFG is un-decidable as well. However, most programming languages have only limited the issues with ambiguity.

3.5.3.2. Recursive Productions

Productions are often defined in terms of themselves. For example a list of variables in a programming language could be specified by this rule:

variable_list → *variable*
variable_list → *variable_list*, *variable*

Such productions are said to be *recursive*. If the recursive non-terminal is on the left side of grammar rules, it is called a *left-recursive* rule. Similarly, we can define a *right-recursive* rule as in $A \rightarrow u \mid vA$.

Some parsing techniques have trouble with one or the other variants of recursive productions. So sometimes we have to change the grammar into a different but equivalent form. Left-recursive productions can be especially troublesome in the top-down parsers. However, there is a simple technique to rewrite the rule from one recursion to another. For example:

X → X*a* | X*b* | E

To convert this rule, we introduce a new non-terminal Y that we append to the end of all non-left recursive productions of X.

X → ~~X*a*~~ | ~~X*b*~~ | EY

The expansion for Y is basically the reverse of the original left-recursive rule.

X → EY
Y → *a*Y | *b*Y | λ

These rules are equal to the first left-recursive rules and it might seem pointless, but some parsers prefer or even require only left or right recursions.

3.5.3.3. Left-Factoring

A parser usually reads tokens from left to right and it is convenient if upon reading a token it can make an immediate decision about which rule should be chosen. This can become an issue when some rules have common first symbol or symbols.

Stmt → if Condition then Stmt
Stmt → if Condition then Stmt else Stmt
Stmt → other

The prefix "*if Condition then Stmt* " causes problems, because when a parser read "*if*", it does not know which rule should be used. A technique called left-factoring allows us to re-structure the grammar to avoid this situation. It requires to factor out the common parts of the rules in grammar with the beginning terminals and non-terminals into a shared rule which is equivalent to the first set of rules, but it makes clearer for the parser to pick the correct rule.

Stmt → if Condition then Stmt ELSE
ELSE → else Stmt | λ
Stmt → other

3.5.3.4. Hidden Left-Factors and Hidden Left Recursion

A rule may not appear to be a left-recursive or left factored, yet it may have these issues hidden and need to be exposed. For example, consider this grammar:

$A \rightarrow aB$
$A \rightarrow Bc$
$B \rightarrow aA \mid Ae \mid d$

By expanding the rules of this grammar, we will have the following rules:

$A \rightarrow aaA \mid aAe \mid ad$
$A \rightarrow aAc \mid Aec \mid dc$

Now we have left recursive and left factor exposed after one step of expanding rules.

3.5.4. Parsing Techniques

The parser works on strings from a particular context-free grammar. A parser consists of an input buffer, a stack, and parsing table. By reading the input stream, the parser applies the rules in its parse table.

The stack is used to stack up rules that there is no matching rules for now. At the end, parsing will be finished when stack is empty or has some other properties, and only the ending symbol remained in input. Symbol $ is used to indicate the ending symbol for input stream.

Symbol $ is used when parser starts parsing the stack with *S$* where S is the start non-terminal and $ is the end sign. Symbol $ also will be added to input stream like *input$*. The parser verifies the input if both $ in input$ and S$ match at the end of parsing.

Parsing can be done in two techniques: top-down and bottom-up. Here we will review these two main parsers called LL parsers where they use top-down technique, and LR parsers where they use bottom-up technique. There are also some other variations of parsers that can be found in [77, 78].

3.5.4.1. LL Parsers

LL parsers analyze the input string from Left to right, performing leftmost derivation of the sentence. All LL parsers are called LL(k), where k is the number of tokens when parser a sentence. For k = 1 where with only one token, the matching rule can be selected, then the grammar will be a LL (1) grammar. Considering the grammar in Table 4,

Table 4 - An example grammar without LL(1) attributes

S → F
S → (S + F)
F → *a*

This grammar is a LL(1), where by taking one symbol for example "(", the matching rule can be selected. However, for left-factoring and left-recursive grammars, LL(1) will not work and these grammars should be converted to a grammar without left-factoring and left-recursive attributes. For these reasons, we have the following definitions:

- **First set**: Considering the following rule,

  $$X \longrightarrow Y_1 / Y_2 / Y_3 | \ldots | Y_n$$

  Each terminal in each different expression of X is a member of First Set of X. For Table 4 we have the following first sets for each non-terminal:

  $$First(S) = \{(, a\}$$
  $$First(F) = \{a\}$$

- **Follow set:** Considering the following rule,

  $$Z \longrightarrow \alpha X \beta$$

  Each terminal after every non-terminal X is a member of Follow Set of X. For Table 4 we have the following follow sets:

  $$Follow(S) = \{+\}$$
  $$Follow(F) = \{), \$ \}$$

  In LL(1) parsers, the following rule should be satisfied where it indicates that non-terminals should not share a terminal.

  $$\forall i, j \in 1..n,\ i \neq j \Rightarrow \text{First}(Y_i) \cap \text{First}(Y_j) = \phi$$

  If there is a shared terminal like:

  $$i, j \in 1..n,\ i \neq j,\ \text{First}(Y_i) \cap \text{First}(Y_j) = \{a\}$$

Then, by seeing terminal '*a*' in input, parser cannot decide whether it should use $X \rightarrow Y_i$ or $X \rightarrow Y_j$. A grammar is LL(1) grammar if and only if the First set of all expressions of a none-terminal is empty. LL (1) grammars are not ambiguous grammars

anymore. Constructing an LL(1) parser can be found in [79]. LL(a) parsers are constructed as below.

1. *All terminals and $ symbol are column of parse table.*
2. *All non-terminals are rows of parse table*
3. *For each cell, use First set of the non-terminal to place the rule in a column which the terminal is achieved from.*

For example, the parse table of Table 4 is shown in Table 5.

Table 5 - LL(1) parse table for the grammar given in Table 4

	(	a	+	)	$
S	S → (S + F)	S → F			
F		F → a			

First(S) = {(, a}

First(F) = {a}

The parsing procedure starts with [input, $]. Also [$, S], where S is the top most symbol on stack, will be added to stack. The parses start with reading the next available symbol from the input and the top-most symbol from the stack. If the input matches the symbol on the top of stack, the parser discards the both. For the next step, the next input symbol will be read and another symbol from stack will be checked. If the input token and stack symbol does not match, parser will refer to parse table and re-write the available rule in the table. The re-write operation will push the rule into stack and repeat the matching process. An input will be acceptable for the grammar when the parser reports that both input stream and stack are empty. For an example, parsing input string "(*a* + *a*)" using the *LL* (1) parser in Table 5 is shown in Table 6.

Table 6 - LL(1) parse process for input "(*a* + *a*)" using Table 5

Input String	Input token	Stack	Re-write rule	Description
(a+a)				
(a+a)$		S $		Initialize
a+a)$	(	S $		Read token
a+a)$	(	(S+F) $	S → (S + F)	Re-write rule at S and (
a+a)$		S+F) $		Discard matching (
+a)$	a	S+F) $		Read token
+a)$	a	F+F) $	S → F	Re-write rule at S and a
+a)$	a	a+F) $	F → a	Re-write rule at F and a
+a)$		+F) $		Discard matching a
a)$	+	+F) $		Read token
a)$		F) $		Discard matching +
)$	a	F) $		Read token
)$	a	a) $	F → a	Re-write rule at F and a
)$		) $		Discard matching a
$	)	)$		Read token
$		$		Discard matching)
	$	$		Read token
				Discard matching $
				ACCEPTED

3.5.4.2. LR Parsing

LR parsers, Left to right parse, rightmost derivation in reverse, are the most general bottom-up parsing methods used to construct shift reduce parsers. Figure 22 shows a LR parsing method.

LR parsers start with a token from input and build sub trees of rules. The parse tree builds up incrementally. At each step, parser accumulates a list of sub-trees or phrases of input text that have been already parsed. These sub-trees are not connected yet, until parser reach the right end of the syntax pattern that will combine them.

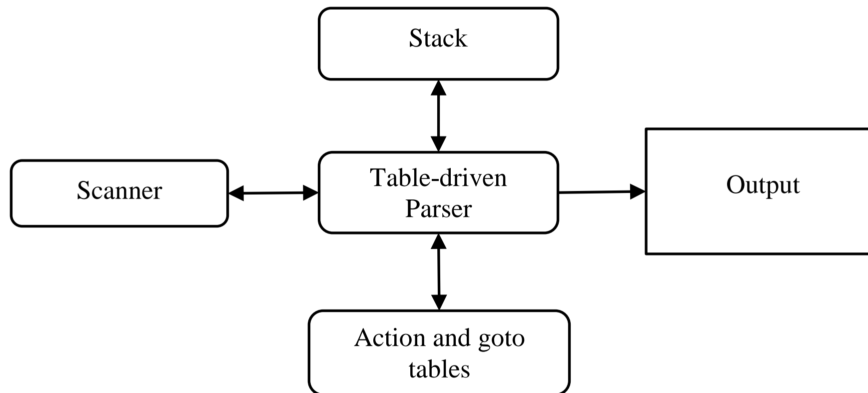


Figure 22 - LR parser

LR parsers are complicated and require more efforts than LL parsers. More information about LR parsers can be found in [79].

3.6. Parser Generator Tools

Compiler construction is a west area in computer science that deals with the theory and practice of developing programming languages. A compiler-compiler helps to produce any scanner and parser for a particular language from its grammar. Input grammar must follow the compiler-compiler rules. A compiler-compiler in fact is a compiler generator which takes grammar rules as its input and produces a compiler as an output for the given grammar. Figure 23 shows how a compiler-compiler generates the scanner and parser for the given grammar.

For automatically generating parsers, there are many different generator tools that can create source codes in various languages. Typically examples of these tools are *YACC* [80] and *bison* [81], for imperative languages, *ml-yacc* [82], and *happy* [83] for functional languages, *t-gen* [84], *JavaCup* [85], and JavaCC [86,87] for object-oriented languages. They require a special type of grammar as input, and generate either LL(k) or LR (*k*) parsers. Therefore, after choosing a parser generator, a proper grammar must be described according to the specifications of that generator.

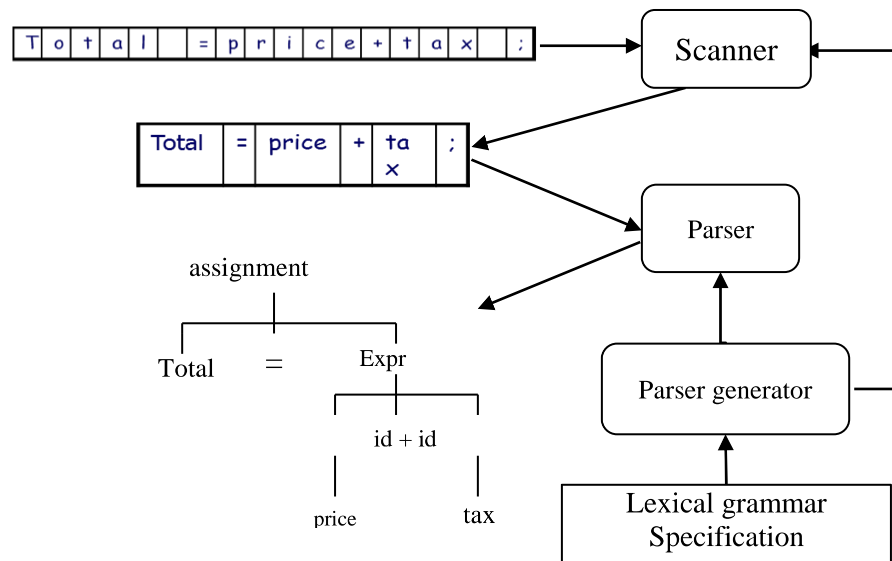


Figure 23 - How Compiler-compiler works

We do not review any particular compiler-compiler thoroughly in this thesis. However, we introduce YACC and JavaCC as two most popular tools for generating compilers from input grammars of a language.

3.6.1. YACC

One of the most well-known and popular compiler-compilers is YACC which stands for *Yet Another Compiler-Compiler*. The output of YACC is a source code of a compiler program in C language. YACC produce LALR parsers. However, for a full syntactic analysis, an external lexical analyzer like Lex or Flex is required.

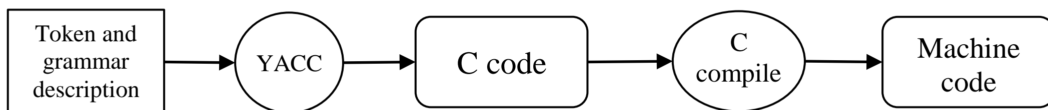


Figure 24 - Parser generation with YACC

3.6.1.JavaCC

JavaCC is a java-based parser generator that generates a top-down parser. Top-down parsers or recursive decent parsers allow the use of more general grammars. The only limitation of these parsers is that left recursion is not allowed because this may lead to infinite recursion. Top-down parsers have a structure identical to the grammar specification and are thus easier to debug. Embedding user code to generated abstract syntax tree in a top-down parser is simple due to the easiest of passing arguments and values across the nodes of parse tree.

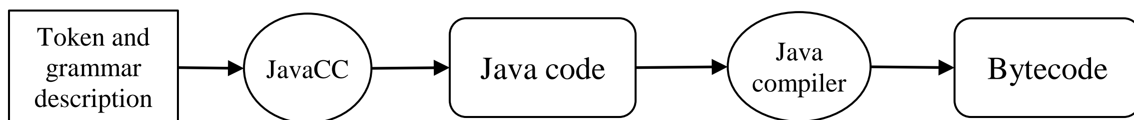


Figure 25 - Parser generation with JavaCC

JavaCC has a mechanism as a default that looks ahead one token on input stream. However it provides a capability which more than one token can be looked ahead. JavaCC also provides both syntactic and semantic look ahead to eliminate some ambiguities.

JavaCC also provides Lex like lexical analyzer. Token Manager is the component of JavaCC that is used to recognize tokens of the grammar where it is an implementation of a non-deterministic finite automaton.

For more information about JavaCC and its specifications, read more at [88].

3.7. Languages for Mathematical Expressions

Similar to programming languages, there are various mathematical languages to describe mathematical expressions. Three most well-known mathematical languages are discussed below: MPL, MathML, and LaTex.

3.7.1. Mathematical Pseudo-Language (MPL)

Mathematical pseudo-language (MPL) is an algorithmic language that is readily expressed in Maple, Mathematica, and MuPAD. It is a high-level user-oriented programming language intended particularly for developing, testing, and communicating mathematical algorithms. Mathematical expressions in MPL are consisting of the following symbols and operators:

- *Numbers* and *fractions*
- *Identifiers* are used both as *programming variables* and *mathematical symbols*
- *Algebraic operators* +, -, *, / , ^, and ! (factorial)
- *Function form* that are used for mathematical functions such as *sin(x)* and *arctan(x)*; mathematical operators such as *Integral(u,x)*; and undefined functions such as $f(x)$, $g(x,y)$.
- *Relational operators* =, ≠, <, ≤, >, and ≥
- *Logical constants* ***true*** and ***false***
- *Logical operators* ***and***, ***or***, and ***not***.
- *Finite sets* that utilize the set operations ∪, ∩, ~, and ∈.
- *Finite lists* represented by [and]. An empty list, which contains no expressions, is represented by [ ].

3.7.2. MathML

MathML or Mathematical Markup Language is an XML-based language for describing mathematical expressions. Its goal is to integrate mathematical expressions and formulae into web pages and other documents. MathML consists of two parts where Content MathML describes semantic meaning, and Presentation MathML describes screen view and layout of the expression.

```
<math>
   <apply>
      <plus/>
      <apply>
         <times/> <ci>b</ci> <ci>x</ci>
      </apply>
      <ci>c</ci>
   </apply>
</math>
```

Figure 26 - "*bx+c*" in MathML

A common and more human-readable representation of mathematical expressions than MathML is provided by LATEX where expressions are presentational and can be easily converted to Presentation MathML.

3.7.3. LaTex

LaTeX shortening of Lamport Tex is a document preparation system and document markup language. It is widely used for the communication and publication of scientific documents in many fields including mathematics, physics, computer science, etc. It also has a projecting role in the preparation and publication of books and articles that contain complex multilingual materials, such as Sanskrit and Arabic. LaTeX is not the name of a particular editing program, but refers to the encoding or tagging conventions that are used in LaTeX documents.

Like TeX, LaTeX started as a writing tool for mathematicians and computer scientists, but from early in its development it has also been taken up by scholars who needed to write documents that include complex math expressions and non-Latin scripts.

```
\lim_{x \to \infty} \exp(-x) = 0
```

$$\lim_{n \to \infty} exp(-x) = 0$$

Figure 27 - an example of a math expression in LaTex

3.8. An Evaluation of Mathematical Expressions

In cases where more complex mathematical expressions are evaluated they need converting into an intermediate representation, a corresponding data structure must be constructed for them. As mentioned in section 3.3, the structure of AST is suitable for this purpose, which can be created by adding some code into the parser generator.

An object tree can be evaluated starting from the innermost node towards the root node. The operations performed for each node are based on the types of those nodes. Using three distinct approaches, it is possible to determine the type of a node and to carry out the appropriate operation. Advantages and disadvantages of these approaches are summarized in Table 7.

Table 7 - A comparison of syntax tree evaluation approach

Method	Object derivation	Class compilation
IntanceOf Operator	Yes	No
Interpret Methods	No	Yes
Visitor Design Pattern	No	No

Object derivation as in object orientation programming, indicates that classes can inherited the properties of other classes. The parent class defines the interface of the object, and sub-classes of children can inherited those methods or override them. With this, parent class can hold all children objects and later on, cast back to children classes. Class compilation requires other extended children to be compiled before using the whole program. So, all sub-classes should be re-compiled whenever any changes made to one child.

In this thesis, we basically use *interpret()* methods. However, in some cases, a combination with *instanceof* is also used. This is due to the need to identify the type of child node for further evaluation. For example, if the type of the child node is *Num*, it is clear that this node does not need to be evaluated.

3.8.1. Instanceof Operator

In this approach, the type of a node is determined by the instanceof operator. After determining the type of the object, the objects of subclasses are derived from the super class reference variable for the practical realization of the nodes. The type derivation (cast) can be done using the structure of the relevant sub-class object. The *eval* function defined in Table 8, receives the object tree and the value of *x* as a parameter. Using the argument exp, it recognizes the type of the related node and performs the corresponding operation based on the type of that node, possibly calling its sub nodes with the value of *x* –parameter as well. In this example a mathematical expression is first given to the constructor *EvalParse* with which the parser is invoked to generate the syntax tree. Then the tree is sent to *eval* function for evaluation it with the value x typed by the user.

During the evaluation process, the *eval* function first recognizes the type of the root node, and calls its sub nodes, passing the value of x. For example if the type of a node is *Plus*, the *eval* () function recalls the left and right sub nodes separately and returns the sum of the resulting values of these two calls. The some operation is repeated for the other nodes. The function finally displays the value calculated and returned for the root node.

Table 8 - Evaluation with the operator

```
// EvalExp.java
public class EvalExp {
   public static void main(String[] args) {
     EvalParse parser = new EvalParse(System.in);
     try {
       System.out.println(eval(parser.parse(),
                            Double.parseDouble(args[0])));
     } catch(ParseException e) { e.printStackTrace(); }
   }
public static double eval(Exp, double x) {
   if (exp instanceof Plus)
     return eval(((Plus)exp).exp1, x) + eval(((Plus)exp).exp2, x);
     else if (exp instanceof Minus)
       return eval(((Minus)exp).exp1, x) -
                  eval(((Minus)exp).exp2,x);
     …
     else if (exp instanceof Power)    …
     else if (exp instanceof Euler)    …
     else if (exp instanceof Num)
       return ((Num)exp).num;
     else
       return x;
   }
 }
```

During the evaluation process, the *eval* method is called from the derivate object. The method finally returns the result to the root node.

3.8.2. Interpret() Methods

In this approach, for each class used to create the AST structure, the statements which evaluate the internal data of those classes are defined in eval methods. In order to perform some operation on a node, it is sufficient to call the *eval* method of that node. So there is no need to determine the type of the processed node. Table 9, shows the *eval* methods for some classes. The value of *x* is passed as a parameter to each *eval* function. *Interpret()* methods allow the overriding of parent methods by children nodes. This is a powerful technique that helps to reduce the complexity of the evaluation process by enforcing each operator to perform the related operation within itself.

Table 9-Addition of *eval* methods to syntax classes

```
...
class Plus extends Exp {
  …
   public double eval(double x) {
      return (exp1.eval(x) + exp2.eval(x));
   }
}
…
class Num extends Exp {
  …
   public double eval(double x) {
      return num;
   }
}
class Var extends Exp {
   …
   public double eval(double x) {
      return x;
   }
}
```

The main class of the evaluation is given in Table 10.

Table 10 - The main class for interpretation methods

```
public class EvalExp {
   public static void main(String[] args) {
      EvalParse parser = new EvalParse(System.in);
      try {

System.out.println(parser.parse().eval(Double.parseDouble(args[0])))
;
      } catch(ParseException e) {
         e.printStackTrace();
      }
   }
}
```

The program partially described in Table 9 and Table 10, calculates and prints the result of a mathematical expression for a given value of x. Each syntax class has a method *eval()* which interprets the corresponding component of the expression whiteout having the know the type of the related node. From the root node towards the left the eval functions invoke each other in a way managed the hierarchical structure of AST.

3.8.3. Visitor Design Pattern

This pattern includes a visitor interface and a visitor class. The class defines a method recalled *visit()*, for each syntax class, whose prototype is inserted into the interface. Beside s, a method called *accept()* is added into each syntax class. Whit the methods designed as visitors, the local data of syntax classes are separated from the operations that evaluate those data. The *visit()* methods are used to access and evaluate the nodes of syntax tree. A *Visit()* function calls the *accept()* function of each object in the evaluate node and the each *accept()* function recalls the visit function on its own object. In this way, the *visit* and *accept* methods are to recall each other until all the nodes of the object tree are visited. A deferent interpretation of the tree requires a new definition of the visitor pattern.

Table 11 - Adding *accept()* methods to syntax classes

```
abstract class Exp {
  public abstract double accept(Visitor v);
}
  class Plus extends Exp {
     public Exp exp1, exp2;
     public Plus(Exp e1, Exp e2) {
        exp1 = e1; exp2 = e2;
     }
     public double accept(Visitor v) {
        return v.visit(this);
     }
  }
...
  class Num extends Exp {
     public double num;
     public Num(double n) {
        num = n;
     }
     public double accept(Visitor v) {
        return v.visit(this);
     }
  }
…
}
```

The evaluation of an object tree starts with the invocation of the relevant visit() method. The visit function initially tries to find the type of the root node of the tree, and therefore calls the accept() method of the node with the current visitor reference. Then the accept() method calls back the visit() method on the correct type of the node object. This stage of the evaluation starts a possible new sequence of visit() and accept() calls. Table 11 shows *accept* methods added into some syntax classes. Table 12 shows the *Visitor* interface and *EvalVistor* class that is required to evaluate the object tree.

Table 12 - Visitor interface and its implementation for syntax tree

```
public interface Visitor {
   public double visit(Exp exp);
   public double visit(Plus exp);
   …
   public double visit(Num exp);
   public double visit(Var exp);
}
public class EvalVisitor implements Visitor {
   double x;
   public EvalVisitor(double a) {
       x = a;     }
   return (exp.num);
}
public double visit(Var exp) {
   return x;
}
public double visit(Exp exp) {
   return exp.accept(this);
}
public double visit(Plus exp) {
   double a = exp.exp1.accept(this);
   double b = exp.exp2.accept(this);
   return (a + b);
}
   …
}
```

Table 13 also shows the main class for the evaluation. As seen in Table 11, Table 12 and Table 13, the evaluation operations of the nodes are defined in the visit() methods of the Evalvisitor class. These operations access the data of relevant object by means of the pair of visit.accept methods.

Table 13 - Main evaluation class for the visitor design

```
public class EvalExp {
   public static void main(String[] args) {
      EvalParse parser = new EvalParse(System.in);
      try {
         // Evaluate f(x) for x = args[0];
         System.out.println(new EvalVisitor(
         Double.parseDouble(args[0])).visit(parser.parse()));
      } catch(ParseException e) {
         e.printStackTrace();
      }
   }
}
```

4. STEP-BY-STEP SOLUTIONS FOR MATHEMATICAL EXPRESSIONS

In this section we propose a grammar based methodology to solve mathematical expressions step-by-step, which is designed a multi-part system. The system uses an extended grammar to parse input data, input data, and then converts it into a tree data structure, which provides an appropriate model for easily processing a particular mathematical expression. Each node of the tree includes a symbol or term of the input expression, and it's the relations with other symbols or terms, where symbols can be operators, numbers, etc.

The tree representation also helps to apply recursive operations over a node and its children. For example, the evaluation of a node requires firstly evaluating the children of that node. Besides a certain combination of nodes can define a pattern to which some other specific operations needs applying. A typical example would be a basic distributive transformation where a complex representation of summation expression is simplified to a simple one.

The methodology consists of the following phases and steps:

Phase 1: Parser definition

Steps: Grammar design, grammar conversion, compiler-compiler input construction, code generation

Phase 2: Input data analysis

Steps: Token generation, syntax analysis, AST generation

Phase 3: Evaluation and interpretation

Steps: Solving expressions, simplification, printing intermediate evaluation results.

Some of these phases and steps are also shown in Figure 28.

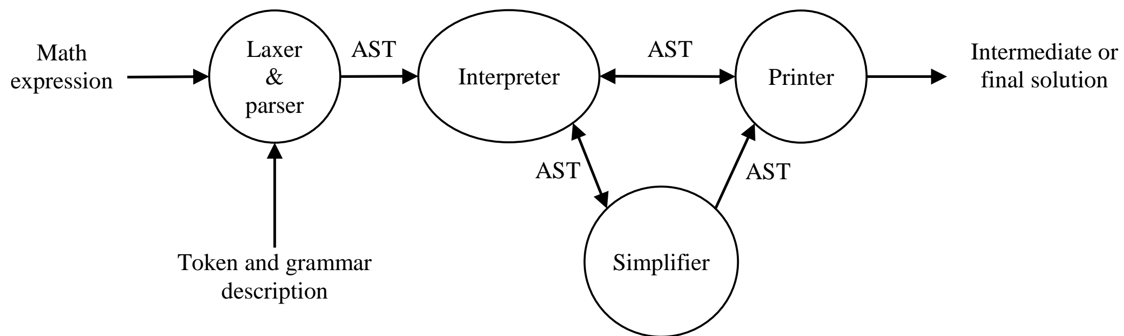


Figure 28-Some important phases and steps of the proposed methodology

Using these steps shown in Figure 28, one can implement a framework to process mathematical expressions. In this thesis, we will work on algebraic expressions to illustrate some applications of the methodology.

4.1. A BNF Grammar Definition

The BNF notation is used to define context-free grammars for formal languages, especially programming languages. It has simple notations; recursive structures, and widely available. Many compiler generation tools, such as YACC [80], LEX [89], and JavaCC [86, 87] are needs a BNF-like description of a source language.

As mentioned in Section 3 mathematical expressions can contain operations such as addition or subtraction, functions such as *sin* or *cos*, special symbols such as integral, etc. Given a grammar developed for a particular mathematical expression, all operators, functions, and symbols, variables and numbers will be members of the *terminal* set. The n*on-terminal* set will be determined based on the production rules of the grammar.

The designed grammar must generate arithmetic expressions with an operator and its operands, or a mathematical function with arguments. An operand or argument itself might be a number, a variable, or another mathematical expression.

The production rules of grammar might be recursive, since there are operands of a type "expression" shortly *expr*. In addition, the decimal (or integer) numbers can be generated up to desired number of digits.

Every different kind of mathematical expressions requires the use of different grammars. In this section, a derivative system is implemented by developing an extended

BNF Grammar for mathematical expressions, with some modifications to a grammar presented in [90], which is shown in Table 14.

Table 14 - An Extended-BNF grammar for mathematical expression

```
G = {Σ, T, V, P, S}
V = {expr, op, func, var, number, digit}⊆ Σ
T = {x, Sin, Cos, Tan, Log, Exp, Sqrt, +, -, *, /}⊆Σ
Σ = T ∪ V
S = { expr }
P :
<expr>::= <expr> <op> <expr>
|(<expr>)
|<func>( <expr> )
|<var>
|<Number>
<op>::='+' | '-' | '*' | '/' | '^'
<func>::= 'Sin' | 'Cos' | 'Tan' | 'Log' |'Exp' | 'Sqrt'
<var>::='x'
<number>::= '-' ? <digit> + ('. ' <digit> +)?
<digit>::=[' 0'-'9']
```

The grammar Table 14 has five operators and six functions that can reside in simple mathematical expression. However, it can be modified with the insertion of some other operators, functions, and symbols.

4.2. Grammar Conversion

Parsing an expression means processing expression structure via a grammar. One of the most important connections between grammar and expression is the type of parse tree derivation. This connection can be determined by a parsing method, which works under some certain conditions. In this thesis, we focus on LL parsers.

A parser with look-ahead *k* is deterministic, if, obtaining *k* symbols from input it can make decision without ambiguity. Generally, if a method *X* uses look-ahead *k*, it is called *X* (*k*). *LL(1)* parsers as an implementation of the top-down technique, are more famous than *LL(k)* parsers where $k>1$. In addition, there are some other parsers developed using the bottom-up technique. LR is the most useful one of them.

A LR(1) parser is more effective than *LL(1)* ones. However, in order to avoid their implementation complexity, LL and LR parsers can be easily developed using parser generators.

The grammar designed in Table 14 has to be converted to be able to use one of the conventional parsing methods. There are various tools developed for parser generation called compiler-compiler or CC. Each CC is suitable for a specific kind of parsing methods. These tools in fact are compiler generators where they take the syntax rules of a formal language as input, and return a compiler for that language as output.

In this thesis, we use JavaCC as a parser generator, which is a compiler-compiler tool developed for the top-down parsing. Top-down parsers can easily work with public grammars. The only limitation for this type of grammar is that left recursion and left-factoring are not allowed.

JavaCC produces parsers for LL (*k*) grammars that have the following features:

- *For any rule $A \rightarrow \alpha|\beta$, it should satisfy*

  $$First(\alpha) \cap First(\beta)=\Phi; \;\; if\ \alpha \Rightarrow \lambda \ then\ First(\beta) \cap Follow(A)= \Phi.$$

- *Rules are not left-recursive.*
- *If G is LL (k) then G is also LL(k + 1), where $k \geq 1$*
- *If G is LL(k) then G is not ambiguous, where $k \geq 1$*

It is easy to see that, given the grammar in Table 14, all operators have the same level of precedence. As we described in section 3.5.3.1, this will make it an ambiguous grammar. In order to use a compiler-compiler tool like JavaCC, the grammar has to be *LL(1)*. Table 15 shows the equivalent *LL (1)* grammar, which is modified by considering the operator precedence and associativity.

Table 15 - A LL(1) grammar for mathematical expressions

```
G={Σ, T, V, P, S}
V={expr, element, term, unary, power, func, number, digit}⊆ Σ
T={x, Sin, Cos, Tan, Log, Exp, Sqrt, ( , ), +, -, *, /, ^}⊆Σ
Σ=T ∪V
S={expr}
Productions
<expr> ⟶ <term> [("+"|"-") <term> ]*
<term> ⟶ <unary> [("*"|"/") <unary>]*
<unary> ⟶ ("+"|"-") ? <power>
<power> ⟶ <element> ("^"<power>)?
<element> ⟶ <func> (<expr>) | <number> | "x"
<func> ⟶ "Sin " | "Cos" | "Tan" | "Log" | "Exp" | "Sqrt"
<number> ⟶ "-"? <digit> + ("."<digit>+)?
<digit> ⟶ [" 0"-"9"]
```

The grammar in Table 15 is an optimized version of normal *LL (1)* grammar where all extra non-terminals are combined back to the original non-terminal.

4.3. Definition of Syntax Classes

Syntax is the writing rules of a language. Mathematical expressions, as well as other languages, have certain syntax where there is a finite combination of components. Numbers, variables, operations, functions, symbols of grouping, and other syntactic symbols are used in math expressions. Each component has a specific structure which needs to be defined.

The imperative languages, like C, have least support for implementing data structures. However, the functional programming languages, like Haskell, provide wide support for structural implementation. In these languages, the use of user-defined recursive data types is usually preferred. The object oriented languages are more suitable for representing structures than the imperative and functional languages [91].

In an object oriented approach, an appropriate method to define the structure of mathematical expressions is the use of syntax classes. In this method, based on the structure of the components, a class is defined for each one.

In this thesis, we use syntax classes that are implemented using the object oriented concepts of Java. Table 16 shows some syntax classes and their attributes adapted to our work.

Table 16 - The syntax classes for some mathematical components

Component	Syntax format	Syntax class	Attributes
+	*Exp + Exp*	Plus	*exp1, exp2*
*	*Exp * Exp*	Times	*exp1, exp2*
^	*Exp ^ Exp*	Power	*exp1, exp2*
Sin	Sin(*Exp*)	Sin	*exp*
Cos	Cos(*Exp*)	Cos	*exp*
Numbers	*n*	Num	*n*
Variables		Var	

As in Table 16, the additional syntax classes can be defined for other operators, functions, or symbols. In object-oriented programming, each rule is generally defined by a class, which is then used to evaluate the expressions. Table 17 shows the definitions of some classes given in Table 16.

The syntax classes can be used to create some structure. The AST tree structure is the most suitable one for mathematical expressions. Unlike parse trees, an AST can hold essential sections of input in the form of a tree. An AST can be defined simultaneously with a parser that produces it. In the next section, we will the creation of an expression parser via the JavaCC tool.

Table 17 - The definition of some syntax classes shown in Table 16

```
public abstract class Exp {
  public Exp exp1, exp2;
  public Exp (Exp e1, Exp e2){
   this.exp1 = e1;
   this.exp2 = e2;
  }
}
public class Plus extends Exp { ... }
...
public class Sin extends Exp {
 public Sin(Exp e) { super(e, null); }
}
...
public class Num extends Exp {
 public double num;
 public Num(double n){ this.num = n; }
}
public class Var extends Exp {
 private String var;
 public Var() { super(null, null) }
 public Var(String var) {
   super(null, null);
   this.var = var;
 }
}
```

4.4. Parser Construction

The parser generator tools often create AST. The structure of AST depends on the language in which the tool generates code. There are many different parser tools generating parser code for various languages. Some of these tools are developed for code generation in imperative languages, such as *yacc* [80], and *bison* [81], and the others such as *ml-yacc* [82], and *happy* [83] generates code for functional languages, as well as those generating code for object-oriented languages, including t-*gen* [84] and *JavaCup* [85]. Each of these tools requires defining a special type of grammar. Some tools are suitable for LL (*k*), and some others are suitable for LR (*k*) etc. Therefore, after selecting a tool for parser, the grammar must be developed according to the specifications of that parser generator.

In this thesis, *JavaCC* are used to produce AST. For this purpose, the designed grammar must be written in LL (*k*) format, because JavaCC uses this type of grammars. In the next sections we discuss the conversion process from Context Free Grammar of math expression derivation to *JavaCC* format and the code generation process for AST in two parts.

4.4.1. Token Specification

A token manufacturer (scanner) with input analysis from the perspective of the word produces a series of tokens. Any word within the input data, which cannot divided into smaller parts, is called token. Tokens are all elements in terminal set ($T \subseteq \Sigma$) which is defined by the grammar given in Table 15. For their grammar in Table 15, the tokens are shown in Table 18, following the specification rules of *JavaCC*.

Table 18 - JavaCC token declarations of the terminals in the grammar Table 15

```
TOKEN : {
 < NUMBER : (["0"-"9"])+("." (["0"-"9"])+)? >
}
TOKEN : { < EOL : "\n" > }
TOKEN : /* OPERATORS */
{
 < PLUS: "+" > | < MINUS: "-" > | < TIMES: "*" >
 | < DIV : "/" > | < POW : "^" >
....
}
TOKEN : /* FUNCTIONS */
{
 < SQRT: "sqrt" > | < SIN: "sin" > | < COS: "cos" >
....
}
TOKEN : /* SYMBOLS */
{
 < X: "x" > | < LPR: "(" > | < RPR: ")" >
}
SKIP : { " " | "\t" | "\r" }
```

In Table 18 each token is defined by the keyword TOKEN. For simplicity, we separate all tokens and categorize them into the groups of numbers, operators, etc. The keyword SKIP specifies the characters which should be discarded. In a similar way, the other tokens in the grammar can be added into Table 18.

4.4.2. Parser Definition

A parser analyzes input data in terms of productivity based on a series of produced tokens. It checks whether the sequence of tokens is generated or not, examining them from the perspective of grammar rules. Therefore, we need a mechanism for verification of tokens and their appearance order based on the language rules. Of course, in the process of analysis, the system must also do a semantic analysis for acceptable input data before evaluation. However, the syntax analyzer (parser) can guarantee the evaluation ability of those data due to structures of mathematical expressions. The parser can be designed by hand-written functions or using parser generator tools. The use of a parser generation tool involves developing the desired grammar based on the description conditions of that parser. For example, a grammar for *Yacc++* must be developed in LR and for *CppCC* in LL form. In this thesis we use *JavaCC* for the parser generation and validity test of entries.

The name of functions or methods in JavaCC declaration is determined according to non-terminal set in the grammar in Table 15. In general, a method must be defined for each non-terminal in a grammar. In some cases it may be useful to combine several non-terminal symbols, defining only a method for them. For example, consider the non-terminal < *expr* > in Table 14. For this non-terminal, there is a rule as <*expr* > ⟶ < *term* > < *expr'* >. Therefore, a method can be defined for it in the parse generator. But it can arise some difficulties in parse tree generation for the related non-terminal < *expr'* >. For such cases, these non-terminals can be merged, resulting in one non-terminal, and can be represented by only one method in the parse generator tool. Table 19 shows the typical examples of the merging operation.

Table 19 - The usual and combined rules for some non-terminals

Normal Rules	Combined Rules
< *expr* > ⟶ < *term* > < *expr′* > < *expr′* > ⟶ ("+"|"-") < *term* > < *expr′* > < *expr′* > ⟶ λ	< *expr* > ⟶ < *term* > { ("+"|"-") < *term* > }*
< *term* > ⟶ < *unary* > < *term′* > < *term′* > ⟶ ("*"|"/") < *unary* > < *term′* > < *term′* > ⟶ λ	< *term* >⟶ < *unary* >{ ("*"|"/") < *unary* >}*

All the parser methods are defined in accordance with the structure of the grammar rules. In Table 20, some *JavaCC* defined methods of the LL (1) grammar are shown.

The other methods for the grammar rules can be defined in the same ways seen in Table 20. The whole definitions of the tokens and rules must be specified extension " .jj", which is taken by JavaCC as input. The output of JavaCC is java code that serves as a parser for the related grammar. This parser can be used for determining the authenticity of input data. In Table 20, the tokens *<EOL>* and *<EOF>* represent the end of line and end of data entries respectively.

Table 20 - Sample JavaCC method definitions for LL(1) grammar rules

<table>
<tr><th>Grammar Rule</th><th>Corresponding Method</th></tr>
<tr><td>< start > ⟶ < expr > $</td><td>void parse() : { } {
  expr() (<EOF> | <EOL>)
}</td></tr>
<tr><td>< expr > ⟶
< term > { ("+"|"-") < term > }*</td><td>void expr() : { }{
term() (
  <PLUS> term() | <MINUS> term()
)*
}</td></tr>
<tr><td>< power > ⟶
< element > ("^"< power >)?</td><td>void power() : { }{
  element() (<POWER> power())?
}</td></tr>
</table>

4.5. Generating Abstract Syntax Tree (AST)

In the previous section, we have discussed about creating a parser to determine the accuracy of data entry. However, for being able to do some operation on mathematical expressions, it is necessary not only to detect the accuracy but also generate the desired data structure. One of the useful structures for mathematical expressions is the syntax tree.

A syntax tree (or mainly object tree) is composed of many nodes linked together in a hierarchical structure. Each node is derived from a syntax class, and is constructed by an object that represents a statement or expression of source data. The syntax tree is usually created with the help of a super class type. Sub classes that inherit from the same syntax super class can be used to create the nodes of an object tree, but these nodes on the tree can be represented by the reference of super class.

By adding a various java statements in *JavaCC*, it is possible to generate AST. Table 21 shows the commands to be added to the code in Table 20 to generate the nodes of AST.

Table 21 - The Java statements added to produce AST

```
Exp parse() : { Exp a; }{
 a = expr() (<EOF> | <EOL>)
 { return a; }
}
Exp expr() : { Exp a, b; }{
 a = term() (
 <PLUS> b = term() { a = new Plus(a, b); }
 | <MINUS> b = term() { a = new Minus(a, b); }
 )*
 { return a; }
}
...
Exp power() : { Exp a, b; }{
 a = element() (
 <POWER> b = power() { a = new Power(a, b); } )?
 { return a; }
}
...
Exp element() : { Token t; Exp a; } {
 t=<NUMBER>
{ return (new Num(Double.parseDouble(t.image))); }
 | <X> { return (new Var()); }
 | <LPR> a = expr() <PPR> { return a;}
 | <SIN> <LPR> a=expr() <PPR> { return new Sin(a);}
 ...
}
...
```

Codes added to Table 20 produce output data among side verifying the input string. Figure 29 demonstrates the steps required to produce the AST for our mathematical grammar.

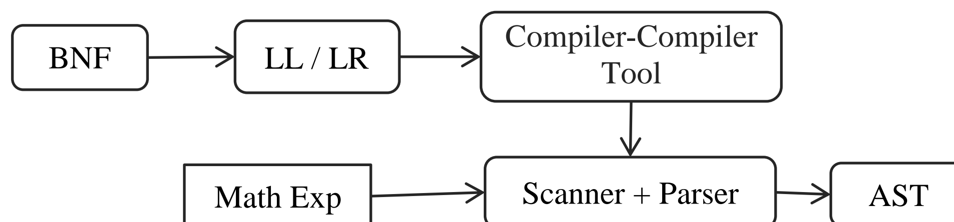


Figure 29 - The steps required to produce the AST for the mathematical grammar

For example, the expression "3*x*+5" will be converted to an AST which is graphically illustrated in Figure 30.

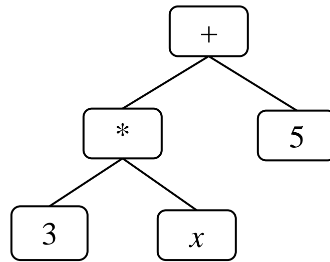


Figure 30 - The graphical illustration of AST for the expression "3*x*+5"

In fact, this AST is programmatically constructed in the parser as follows:

Exp ast = new Plus(new Time(new Num(3), new Var()), new Num(5))

To improve readability of all AST examples in this thesis, we can adapt a more elegant representation of the above AST like

Exp ast = Plus(Time(Num(3), Var()),Num(5))

4.6. Evaluation of Mathematical Expressions

The phase of verifying the input data results in the construction of the related AST. The next phase handles the evaluation of this AST. In general, a mathematical expression can be evaluated in two approaches, either directly or using AST of that expression. The first approach refers to the evaluation of the token components of the expression during the verification phase of the parser. The other approach involves evaluating the AST representation of the expression visiting its token residing nodes. These two approaches are described in the following sections.

4.6.1. Direct Evaluation of Mathematical Expressions

The evaluation of simple mathematical expressions can be conducted at the time of parsing. In this case, there is no need to create any data structure from an input expression and the desired evaluation can embed in the parser. As seen in the previous section, it is possible to issue programming language statements in the process of utilizing parser

generator tools. So, the desired operation can be performed by adding the evaluation code into the parser methods. As an example, let us consider calculating a mathematical expression based on a certain value of a variable. For this purpose, the value of the variable is passed as an argument to the method. Within each method, the corresponding operation is performed on expressions, and the result is returned, which is a double. Table 22 shows some of the methods of this example in *JavaCC*.

Table 22 - Some JavaCC functions for string-based evaluation

```
double parse(double x) : { double a; }{
 a = expr(x) (<EOF> | <EOL>) { return a; }
}
double expr(double x) : { double a, b; }{
 a = term(x) (
 <PLUS> b = term(x) { a = a+b; }
 | <MINUS> b = term(x) { a = a-b; }
 )*
 { return a; }
}
double power(double x) : { double a, b; }{
 a = element(x) (
 <POWER> b = power(x) { a = Math.pow (a, b); }
 )?
 { return a; }
}
double element(double x) : { Token t; double a;} {
 t = <NUMBER> { return Double.parseDouble(t.image); }
 | <X> { return x; }
 | <LPR> a = expr(x) <RPR> { return a; }
 | <SIN> <LPR> a = expr(x) <RPR> { return Math.sin(a); }
 ...
}
```

A significant difference between the functions given in Table 21 and Table 22 is associated with the type of their return values. The functions in Table 22 must always return a number. Given a simple example of the expression "$3x^3+7x+1$" evaluated for x=2, the combined process to passing and evaluation will return 31.

4.6.2. Evaluation of Mathematical Expressions using AST

This approach works on the intermediate representation of expressions, which is demonstrated through some particular applications of the methodology in this thesis. As mentioned in section 4.5, the parsing process produces a hierarchical structure of objects

with the components of a source expression, namely AST. For an example, input string "3x+5" will be converted by the parser into an AST as follows:

$$Plus(Time(Num(3), Var()), Num(5)).eval()$$

Where the class constructor Var represents the variable x. for the expressions with more than one variable, we can pass the name of the variable as an argument to the constructor. In this thesis, we basically use the approach of adding methods into syntax classes. However, in some cases, a combination of two approaches is required to implement the evaluation. In this way, the *instanceof* operator is mainly used in the methods that are added into the syntax classes. This is due to the need to identify the type of the child of each node, when these nodes are visited. To show the intermediate solution steps, when a node is visited, we need to know if that node has the child for further evaluation. For example, if the type of the child node is *Num*, it is clear that such nodes would need no more evaluation.

4.7. Representation of Mathematical Expressions

On each evaluation step of a mathematical expression through the relevant AST, the resulting expression needs converting into one of human-readable, formats such as LaTex, or MathML. This is especially necessary to display the current expression in the AST modified by an evaluation process. Traversing AST wıth the technique of binary search will allow us to get the proper terms to construct the desired output. Figure 31 shows the AST of the expression "($3cos\ (x + 1)$) / 2".

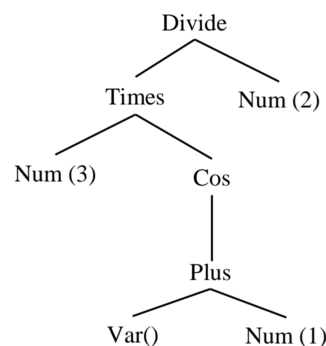


Figure 31- The AST for the expression "($3cos\ (x + 1)$) / 2".

The result of binary traverse or simply syntax tree of Figure 31 is

Divide (*Times*(*Num*(3), *Cos*(*Plus*(*Var*() , *Num*(1)))) , *Num*(2))

By applying the format-converting methods to each class, the required output format can be generated. Below, we will describe the details of what we need to do to achieve this goal.

4.7.1. Human-readable Format

Displaying mathematical expressions in human-readable format needs the conversion of the expressions in syntax tree to a string that is obtained from binary traverse of AST. Table 23 shows the *Print* methods for some classes.

Table 23- *Print* method added to classes to generate human-readable output

Class Name	*Print* Method

Exp	`public String Print() {`
	`  return "";`
	`}`
	`public static String **Out**(String str) {`
	`  if(`
	`    str.contains("-") ||`
	`    str.contains("+") ||`
	`    str.contains("*") ||`
	`    str.contains("/")`
	`  )`
	`    str = "(" + str + ")";`
	`  return str;`
	`}`

Plus	`public String Print() {`
	` String a = exp1.print();`
	` String b = exp2.print();`
	` return Exp.Out(a) + "+" + Exp.Out(b);`
	`}`

Minus	`public String Print() {`
	` String a = exp1.print();`
	` String b = exp2.print();`
	` return Exp.Out(a) +"-" + Exp.Out(b);`
	`}`

Times	`public String Print() {`
	` String a = exp1.print();`
	` String b = exp2.print();`
	` return Exp.Out(a) + "*" + Exp.Out(b);`
	`}`

Functions	`public String Print() {`
	` return "sin(" + exp.Print() + ")";`
	`}`

Num	`public String Print() {`
	` return Integer.toString(num);`
	`}`

The output of these methods can directly be displayed to the user. Additional codes and methods can be added to produce more pretty results. The output of the methods will be an expression such as, for example, "$(3cos(x+1))/2$".

4.7.1. Converting Syntax Tree to *LaTex* Format

LaTex is widely used by scientific community. Similar to other formats of displaying AST in output, LaTex output can be achieved by adding some other methods to the printer component. Table 24 shows the *LaTex* methods for some classes.

MathML also accept LaTex input. Therefore, one can export the result to LaTex format and then use MathML renderer to display it. As mentioned before, LaTex is a very popular format and is supported by many other tools.

Table 24 - LaTex method to generate LaTex statements

<table>
<tr><th>Class Name</th><th>LaTex Method</th></tr>
<tr><td>Exp</td><td>public String LaTex() {
return "";
}
public static String Out(String str) {
if(str.contains("-") ||
str.contains("+") ||
str.contains("*") ||
str.contains("/"))
str = "(" + str + ")";
return str;
}</td></tr>
<tr><td>Plus</td><td>public String LaTex() {
String a = exp1.LaTex();
String b = exp2.LaTex();
return Exp.Out(a) + "+" + Exp.Out(b);
}</td></tr>
<tr><td>Divide</td><td>public String Print() {
String a = exp1.LaTex();
String b = exp2.LaTex();
return "\\frac {" + a + "} {" + b + "}";
}</td></tr>
<tr><td>Power</td><td>public String LaTex() {
String a = exp1.LaTex();
String b = exp2.LaTex();
return "(" + a + ") ^ {" + b + "}";
}</td></tr>
<tr><td>Functions</td><td>public String LaTex() {
return "\\sin {" + exp.Print() + "}";
}</td></tr>
<tr><td>Num</td><td>public String LaTex() {
return Integer.toString(num);
}</td></tr>
</table>

It is easier to understand LaTex output than MathML one. For example, the output of the expression "(3*cos*(*x*+1))/2" in LaTex format is *“\frac{(3\sin(x+1))}{2}”*. For more information about LaTex, refer to [92].

4.7.1. Converting Syntax Tree to *MathML* Format

MathML has become very popular due to its use in web-pages. To construct a string in *MathML* format, it is possible to use built-in functions within MathML SDK. However, by adding some simple *Print* methods this can be achieved with AST.

Similar to those in the previous section, each class has its own print method, but the output will be wrapped between MathML tags. For example, Num and Var classes must be surrounded with *<mi>* and *<mo>* tags respectively. Table 25 shows some *MathML* methods for some classes.

Table 25-MathML methods to generate *MathML* statements

Class Name	*MathML* Method

Exp	public String MathML() {
	  return "";
	}

Plus	public String MathML() {
	 String a = exp1.MathML();
	 String b = exp2.MathML();
	 return "<mi>" + a + "<mo>+</mo>" + b + "</mi>";
	}

Divide	public String MathML() {
	 String a = exp1.MathML();
	 String b = exp2.MathML();
	 return "<mfrac>" + a + b + "</mfrac>";
	}

Times	public String MathML() {
	 String a = exp1.MathML();
	 String b = exp2.MathML();
	 "<mi>" + a + "*" + b + "</mi>";
	}

Functions	public String MathML() {
	 return "<mi>sin</mi>" +
	        "<mi>" + exp.MathML() + "<mi>";
	}

Num	public String MathML() {
	 return "<mn>" + Integer.toString(num) + "<mn>";
	}

The output of these methods requires MathML parser/render, where it needs to be linked within the web-page. As an example, the raw view of the expression "(3*cos*(*x*+1))/2" and its rendered form is displayed in Table 26.

Table 26. MathML example of the expression "(3*cos*(*x*+1))/2"

Raw view	Rendered view
<math xmlns="http://www.w3.org/1998/Math/MathML"> <mfrac> <mrow> <mo>(</mo> <mn>3</mn> <mi>sin</mi> <mo>(</mo> <mi>x</mi> <mo>+</mo> <mn>1</mn> <mo>)</mo> <mo>)</mo> </mrow> <mn>2</mn> </mfrac> </math>	$\frac{(3\sin(x+1))}{2}$

For more information about MathML tags, refer to [93].

4.8. **Automatic Simplification**

One of important and underlying operations in symbolic computation is simplification. However, it has many difficulties in implementation, because some concepts of simplification are naturally challenging. For example, responding the question “which part is simplified?” Will justify this claim.

4.8.1. Basic Transformations for Simplifications

Automatic simplification is defined as a collection of algebraic and trigonometric simplified transformations that are applied to an expression as a part of the evaluation process. There are some basic transformations listed follow:

- *Basic numerical transformation* refers to
  1. Multiplication of numeric operands in a product
  2. Addition of numeric operands in a sum
  3. Evaluation of power operators for base and exponent of numeric operands
  4. Evaluation of factorial with a non-negative integer
- *Basic distributive transformation* refers to a collection of integer and fraction coefficients where the result can be simplified by performing some simple evaluations. Here are some examples of this transformation:

  $1 + x + 2(1 + x)$ will simply to $3 + 3x$

  $x + 1 + (-1)(x + 1)$ will simply to 0
- *Basic associative transformation* refers to a transformations that modifies the structure of an expression u in one of the following ways:
  1. Suppose that u is a sum expression. If s is an operand of u where s is also a sum, then the operators of s will be removed from the expression tree and operands of s will be added as operators of u.
  2. Suppose that u is a product expression. If p is an operand of u where p is also a product, then the operator of p will be removed from the expression tree, and the operands of p will be added as operators of u.

The following expression shows an example of this transformation:

$2\,(x\,y\,z) + 3\,x\,(y\,z) + 4\,(x\,y)\,z$ will simply to $2\,x\,y\,z + 3\,x\,y\,z + 4\,x\,y\,z$ *and then* $9\,x\,y\,z$

- *Basic Commutative transformation* refers to a transformation where it can displace or re-order operands of a product or a sum expression. The following expression shows an example of this transformation:

  $2\,z\,x\,y + 3\,y\,z\,x$ *will simplify to* $2\,x\,y\,z + 3\,x\,y\,z$ *and then* $5\,x\,y\,z$
- *Basic power transformation* refers to simplification associated with the power operator:

  $$u^{v} . u^{w} \rightarrow u^{v+w}\ ,\ (u^{v})^{n} \rightarrow u^{v.n}\ ,\ (u . v)^{n} \rightarrow u^{n} . v^{n}$$
- *Basic difference transformation* refers to a simplification of the negative operands:

  $$-u \rightarrow (-1) . u\ ,\quad u - v \rightarrow u + (-1) . v$$

- *Basic quotient transformation* refers to

$$\frac{u}{v} \rightarrow u \,.\, v^{-}$$

- *Basic identity transformation* refers to

$$u + 0 \rightarrow u \,,\; u \,.\, 0 \rightarrow 0 \,,\; u \,.\, 1 \rightarrow u \,,$$

$$0^{n} \rightarrow \begin{cases} 0, & if\ n\ is\ a\ positive\ integer\ or\ fraction, \\ undefined, & otherwise, \end{cases}$$

$$1^{n} \rightarrow 1 \,,\quad v^{0} \rightarrow \begin{cases} 1, & if\ v \neq 0, \\ undefined, & if\ v = 0, \end{cases} \,,\quad v^{1} \rightarrow v \,.$$

- *Basic unary transformation* refers to

$$+\,x \rightarrow x$$

- *Undefined transformation* refers to a case where u is undefined if the input expression is not well-formed. Another example can be stack overflow or stack underflow.

Notice that the simplification of mathematical expressions are not limited to these transformations. Table 27 shows some simplifications in classes using the above transformations.

Table 27- Simplifications performed in classes

Original Term	Object Tree	Simplification Results
0 + exp	*Plus (Num (0) , exp)*	*exp*
exp + 0	*Plus (exp , Num (0))*	*exp*
0 - exp	*Minus (Num (0) , exp)*	*Times (Num (-1) , exp)*
exp - 0	*Minus (exp , Num (0))*	*exp*
*0 * exp*	*Times (Num (0) , exp)*	*Num (0)*
*exp * 0*	*Times (exp , Num (0))*	*Num (0)*
*1 * exp*	*Times (Num (1) , exp))*	*exp*
*exp * 1*	*Times (exp , Num (1))*	*exp*
0 / exp	*Divide (Num (0) , exp))*	*Num (0)*
exp / 1	*Divide (exp , Num (1))*	*exp*

Transformations, similar to generating output from AST, can be achieved by inserting methods into the simplifier component. More than one transformation might be applied for a simple operator or class. Table 28 shows some methods to implement the simplification. The complete code can be found in Appendix B.

Table 28 - An implementation of simplification methods

<table>
<tr><td>Exp</td><td><pre>public abstract Exp Simplify();</pre></td></tr>
<tr><td>Plus</td><td><pre>public Exp Simplify() {
  Exp e1 = exp1.Simplify();
  Exp e2 = exp2.Simplify();
  if (e1.eval().equals("0"))
    return e2;
  if (e2.eval().equals("0"))
    return e1;
  return new Plus(e1, e2);
}</pre></td></tr>
<tr><td>Times</td><td><pre>public Exp Simplify() {
 Exp e1 = exp1.Simplify();
 Exp e2 = exp1.Simplify();
 If (e1.eval().equals("0"))
   return new Num(0);
 if (e2.eval().equals("0"))
   return new Num(0);
 if (e1.eval().equals("1"))
   return e2;
 if (e2.eval().equals("1"))
   return e1;
 return new Times(e1, e2);
}</pre></td></tr>
<tr><td>Sin</td><td><pre>Public Exp Simplify() {
 return new Sin(exp.Simplify);
}</pre></td></tr>
</table>

The result of the simplification process is also an AST. It can be used instead of the initial AST as it might speed-up the evaluation time if any simplification is performed. Figure 32 presents some structural transformations for some simplifications discussed above.

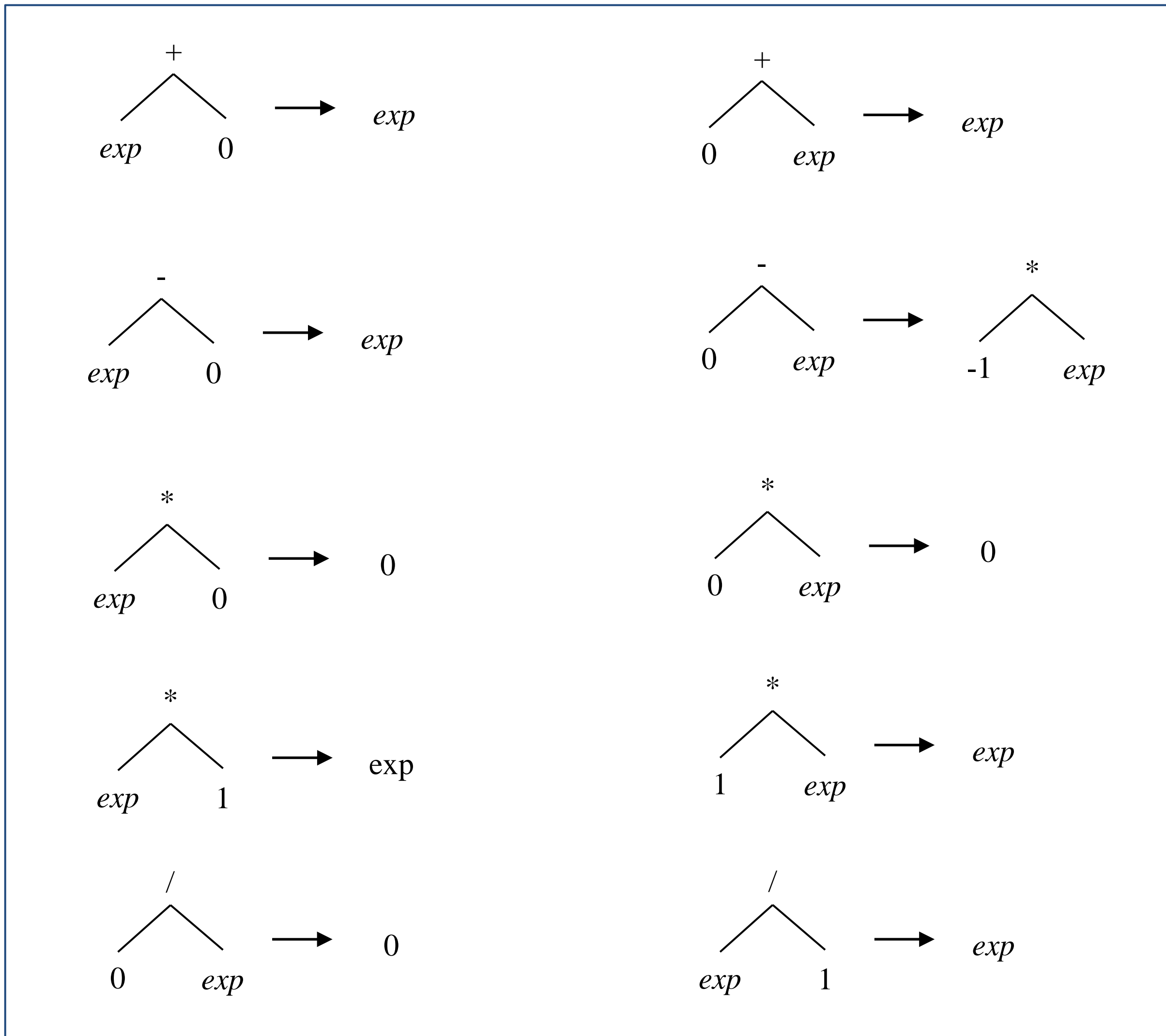


Figure 32-Some transformations with basic simplification rules

4.8.2. Complex Transformations for Simplifications

The simplification of expressions is not limited to basic operations that are listed in Table 27. In this section we present other transformations that can be applied by combining basic ones and properties of some operators. The issue is addressed in some case studies.

As a rule, all simplification methods will be applied until no change is made.

Case Study 1: Similar Operands

The summation of two expressions that have similar types, numbers or variables, can easily be performed. Operators are merged and replaced with their evaluation result or more general term. Figure 33 shows an example.

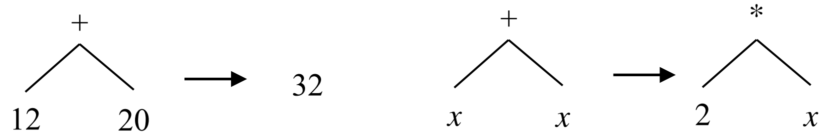


Figure 33-The simplification of similar operands

For all *Plus* nodes, this simplification can be easily applied. Table 29 shows the related code that will perform the transformation:

Table 29-Simplifying similar operands for *Num* and *Var* in Plus

```
if (exp1 instanceOf Num && exp2 instanceOf Num)
 return new Num(exp1.getNum()+ exp2.getNum());

if (exp1 instanceOf Var && exp2 instanceOf Var )
 return new Times (new Num(2), exp1);
```

Figure 34 shows an example of multiple simplifications applied to an AST.

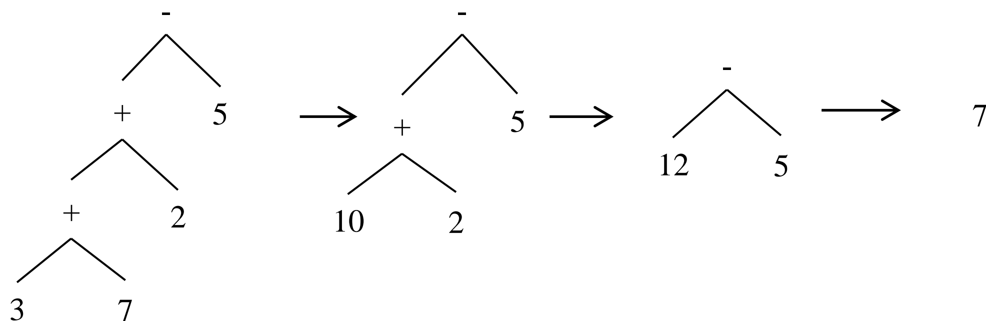


Figure 34-Simplifying the expression "3+7+2-5"

Case Study 2: Similar Operands on Different Levels

Similar operands might be at different levels of AST, therefore normal evaluation of these cases due to different types of operands cannot be performed. Assume that we want to simplify the expression "2+ x + 6 +x".

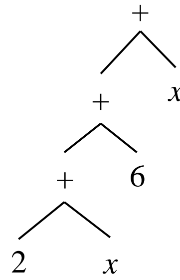


Figure 35-AST of the expression "2+ x + 6 +x"

The expression displayed in
Figure 35 can be simplified to "8+2.x"; however, as mentioned before, operand types differ at each level of tree. The solution to perform more accurate transformation for Plus is to change the tree from binary to a list and then
check if any evaluation can be performed or not. Figure 36 shows the changed structure from
Figure 35.

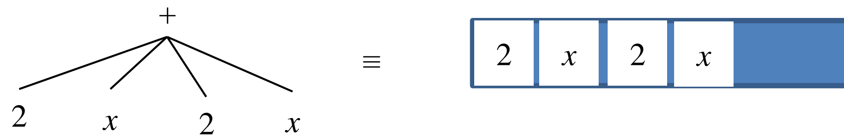


Figure 36-Binary tree to list of operands

Summation can be applied for all operands in the list, as the operator is a *Plus* node. To implement this transformation in codes, an array list is used to keep all operands of operator nodes. The same code can also be used for Minus, Multiply, and Division nodes.

In each operator node, or class, instead of similar operators, a list of all operands is collected and merged together. Table 30 shows the related code for collecting the operands of all similar operator nodes.

Table 30-Collecting operands for similar operators

```
public ArrayList<Exp> sameOpList (){
  ArrayList<Exp> a=new ArrayList<Exp>();
  ArrayList<Exp> b=new ArrayList<Exp>();

  if(exp1 instanceOf Plus || exp1 instanceOf Minus)
    a = exp1.sameOpList();
  else
    a.add(exp1);

  if(exp2 instanceOf Plus || exp2 instanceOf Minus)
    b = exp2.sameOpList();
  else
    b.add(exp2);

  return mergeList(a, b);
}
```

In Table 30, *mergeList ()* is a simple method that concatenates two list. Using basic distributive transformation, the list will be simplified. To add the new result back to the main AST, a new parse will be performed over the result and it will be added as a sub-AST. It is quite possible that some other operators within the Plus or Minus nodes may exist. Therefore, it is also needed to do the same simplifications over those nodes. An example of the expression *"x+3+2*x"* is displayed in Figure 37.

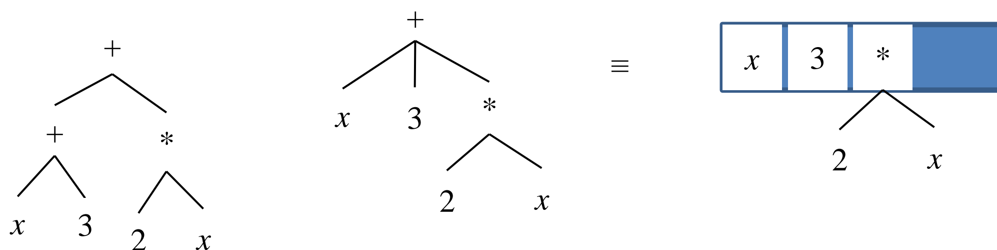


Figure 37-Structural form of the expression "x+3+2x"

If no more simplification can be done, the process will be aborted. The some is true for other operators.

Case Study 3: Fraction Simplification

Numbers and variables can be evaluated and simplified for a fraction operator. This case is similar to basic division transformation.

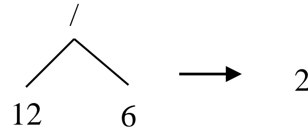


Figure 38 - Simplification of the expression "12/6"

If we consider the structure of the *Multiply* and *Division* operators alike, then we can perform the simplification for expressions that have both numbers and variables. Figure 39 shows the tree and list view of the expression "*6x/12*".

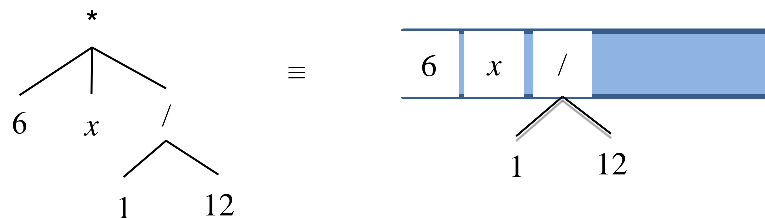


Figure 39-The Tree and list view of "6*x*/12"

In such hybrid case of numbers and variables, each type is individually evaluated with the same type and the result is merged back as a sub-AST to the main AST.

Case Study 4: More Fraction Simplification

It is obvious that some numbers cannot be divided as integer numbers. Such an example is the expression “35/30” where the result is not an integer. A way for simplifying a fraction is to eliminate common parts or GCD. GCD stands for greatest common divisor which is discussed in Appendix C. Figure 40 shows an example.

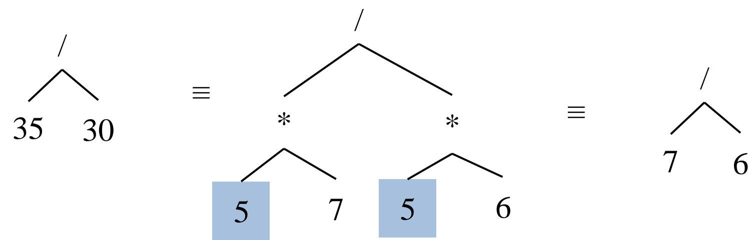


Figure 40-Simplification stages of the expression "35/30"

If one operand is a fraction, and other side an integer, then the integer operand must be converted to a fraction and then be evaluated for simplifications. Table 31 shows changes in the *Divide* class to perform this simplification.

Table 31 – The method simplify() for division class

```
public class Divide extends Exp {
 public Exp simplify() {
   if (exp1 instanceof Divide ||
       exp2 instanceof Divide){
     if (!exp1 instanceof Divide)
       exp1 = new Divide (exp1, new Num(1));
     if (!exp2 instanceof Divide)
       exp2 = new Divide (exp2, new Num(1));
  }
  ...
  //codes for GCD
  ...
  return new Divide(exp1.simplify(), exp2.simplify());
 }
}
```

If two operands of a fraction are other fraction operations, then they need to be transformed into multiplications and a fraction. We use one of fraction properties for this case, which is shown in Figure 41.

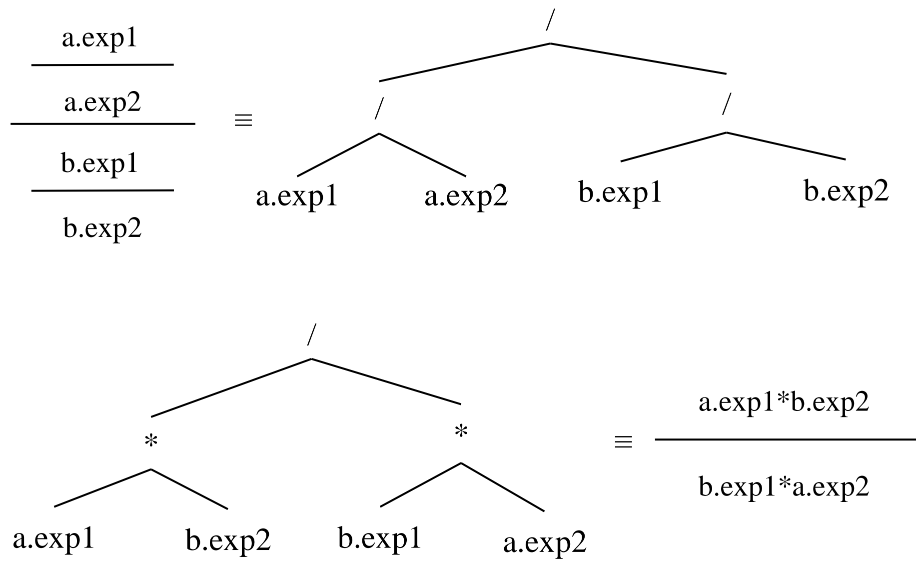


Figure 41-AST of *fraction* children of a *fraction* node

Table 32 shows the related code added to the *Divide* class in Table 31.

Table 32-Changes in method *simplify()* for the *Divide* class

```
exp1 = new Times(exp1.exp1.simplify(), exp2.exp2.simplify()).simplify();
exp2 = new Times(exp2.exp1.simplify(), exp1.exp2.simplify()).simplify();
...
//codes for GCD
...
```

Figure 42 shows an example of fraction simplification for $\frac{\frac{4x^2}{3}}{6x}$

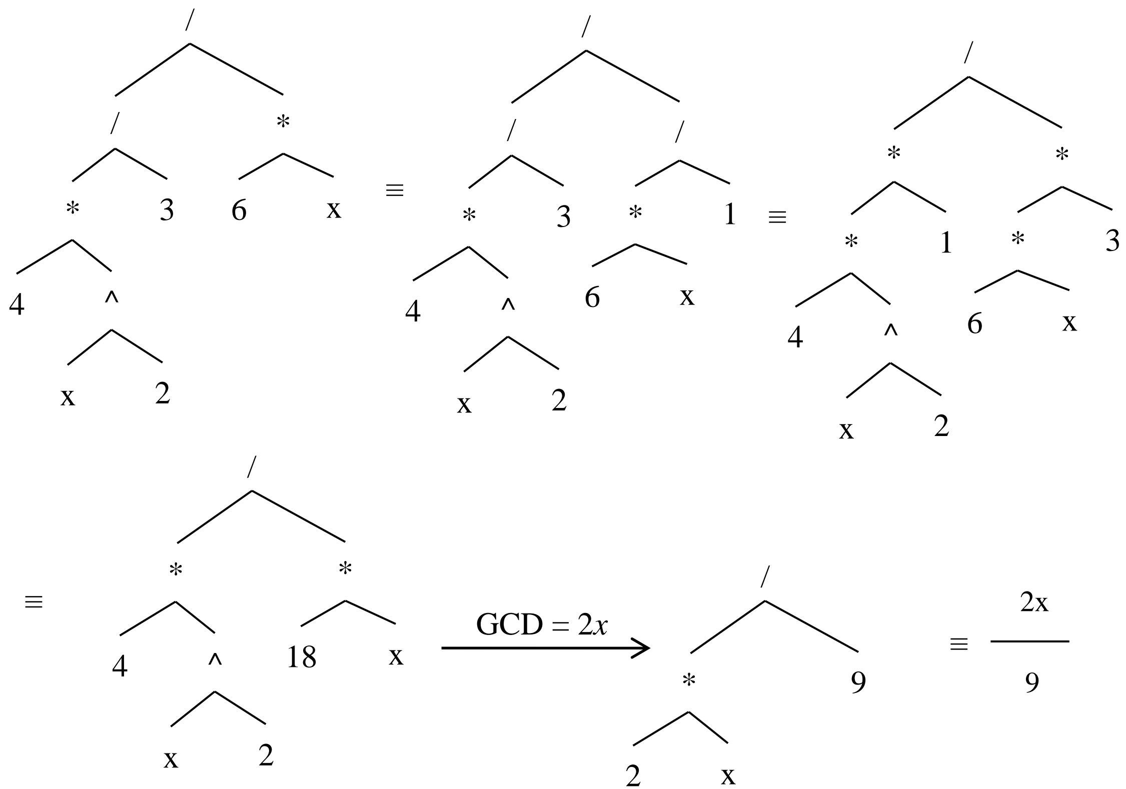


Figure 42-An example of recursive fractions

Other complex simplifications can be performed if operator types of Plus or Minus are multiplications or divisions. Here is an example that shows a fraction expression as operands.

$$\frac{3x}{8} + 2 \equiv \frac{3x}{8} + \frac{2}{1} \equiv \frac{3x}{8} + \frac{16}{8} \equiv \frac{3x + 16}{8}$$

This case indicates that if one side of the operator is a fraction, the other side should be a fraction, too. So it has to be converted into a fraction and then will be simplified as a whole new expression. Some modifications applied to the *Simplify ()* method of *Plus* in Table 28 is displayed in Table 33.

Table 33-Modifications in class *Plus*

```
...
  if (exp1 instanceOf Divide || exp2 instanceOf Divide){

     if (!exp1 instanceOf Divide)
        exp1 = new Divide (exp1, new Num(1));

     if (!exp2 instanceOf Divide)
        exp2 = new Divide (exp2, new Num(1));

     Exp a = exp1.exp1;
     Exp b = exp1.exp2;
     Exp c = exp2.exp1;
     Exp d = exp2.exp2;

     Exp G = polynomialLCM (b, d);

     a = new Times (a, polynomialDivide(G, b).result);
     c = new Times (c, polynomialDivide(G, d).result);

     return new Divide(new Plus(a, c), G);
 }
...
```

Similar modifications must also be made for other classes. However, code factoring can be applied to reduce the number of changes in base codes for each class.

Case Study 5: Exponential Simplifications

The use of the *power* operator is very convenient due to the removal of repeated multiplications for numbers and variables. The exponent part is placed on the top of base by a number or an expression, and refers to the number of times it is multiplied by itself.

The *power* operator and exponential expressions can be simplified in various ways. An example would be the expression $x^{a^{b^c}}$ that is equivalent to $((x^a)^b)^c$ or x^{a*b*c} in terms of left-associativity. All exponentials will be multiplies and then will be placed back as an exponential part (see Figure 43).

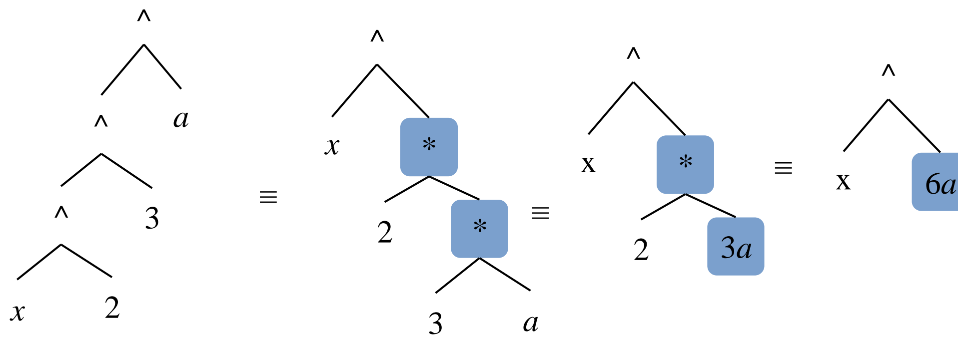


Figure 43-Simplifying exponential expressions

As being clear from the Figure 43, in these cases the *power* operator can be changed to *multiple* operators. Of course, note that according to the presented methodology in this thesis, duplicate powers always appear in *exp1* (left child). It is possible to identify this situation, examining the value of *exp1*. The s*implify ()* method for the *power* class must contain the codes given in Table 34.

Table 34. The *simplify()* method for the Power class

```
public class Power extends Exp {
  ...
 public ArrayList<Exp> sameOpList(){
  ArrayList<Exp> a = new ArrayList<Exp>();
  if(exp2 instanceOf Power )
    return mergeList(a, exp2.sameOpList());
  else
    return exp2.simplify();
 }
...
 public Exp simplify() {
  Exp b = exp1.exp1;
  Exp e = new Num(1);
  ArrayList<Exp> pow = sameOpList();
  for(Exp tmp: pow)
   e = new Times(e, tmp);
  return new Power(b, e);
 }
  ...
}
```

There is another interpretation of the *power* operator where the right side of the expression has higher precedence that is to say; the operator is right-associative. For

example, the expression $x^{a^{b^c}}$ will be evaluated as $x^{\left(a^{\left(b^c\right)}\right)}$, instead of $((x^a)^b)^c$. However, we do not use this case.

Case Study 6: Product of power expressions

There are some other cases for power expressions which are listed in Figure 44 where the transformations of the expressions $u^n.v^n$ and $u^n.v^m$ are demonstrated.

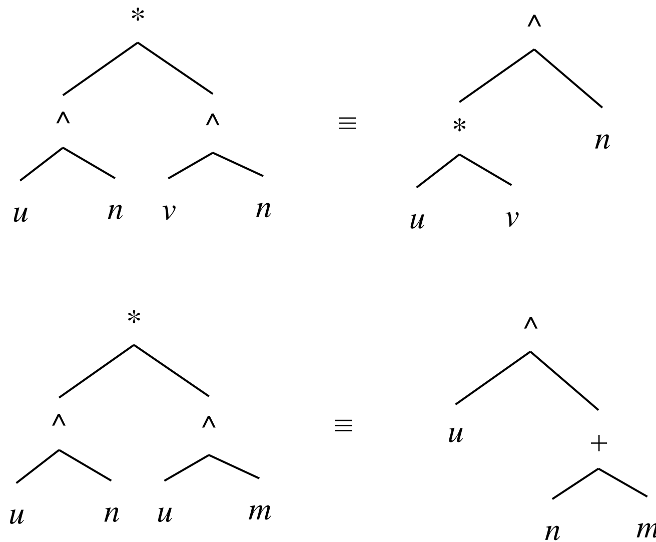


Figure 44- The product of some power expressions

Table 35 shows changes made to *simplify*() method to cover such cases.

Table 35-The transformation code for the product of some power expressions

```
public class Times extends Exp {
 ...
 public Exp simplify() { ...
  if (exp1 instanceOf Power && exp2 instanceOf Power) {
     if (expEqual(exp1.exp1, exp2.exp1)){
       Exp b = exp1.exp1;
       Exp p = new Plus (exp1.exp2, exp2.exp2);
       return new Power(b, p);
     }
  else if (expEqual(exp1.exp2, exp2.exp2)){
       Exp b = new Times(exp1.exp1, exp2.exp1);
       Exp p = exp1.exp2;
       return new Power(b, p);
     }
  }
  return new Times(exp1.simplify(), exp2.simplify());
 }
}
```

Case Study 7: Removal of radical expression in denomination

If a radical expression appears as a dominator, it can be removed by multiplying that expression to nominator and itself. An example is $\frac{1}{\sqrt{x}}$ where the simplified form will be $\frac{\sqrt{x}}{x}$. Figure 45 shows another example for this case where the expression $\frac{3x}{\sqrt{x}}$ is simplified to $\frac{3x\sqrt{x}}{x}$. Later this can be simplified by the *Multiply() method* to a more concise expression $\frac{3\sqrt{x}}{x}$.

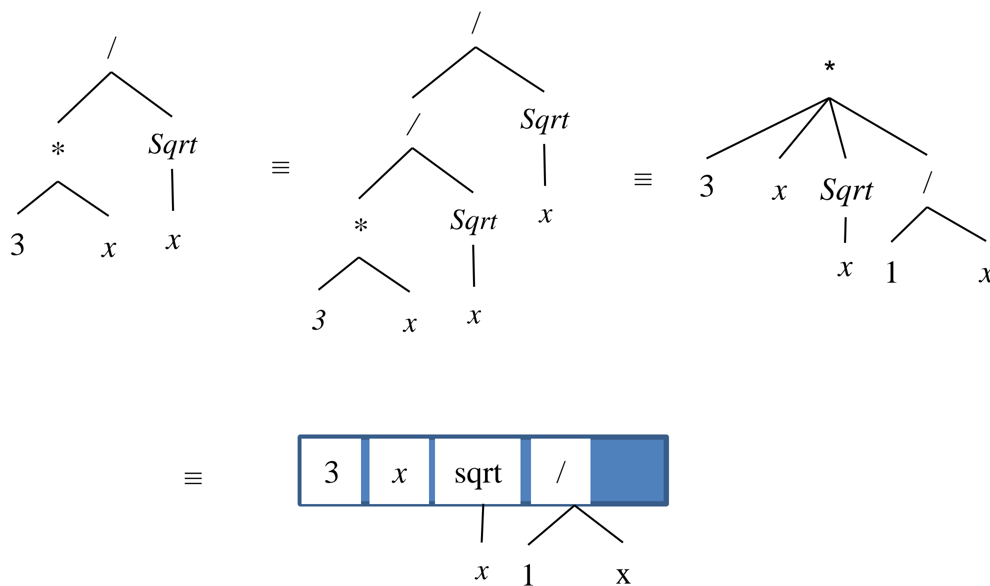


Figure 45 - Simplification of a radical expression as dominator

Table 36 shows the code added to the *Divide* class to cover this case.

Table 36 – The simplify method() for the Divide class

```
...
if (exp2 instanceof Sqrt){
  Exp sq = exp2.exp1;
  return new Div(new Times(exp1, sq), sq);
}.
```

Case Study 8: Algebraic Formulas

The simplification of algebraic formulas is a complex issue. We will briefly focus on expression and factorization of algebraic formulas (i.e. algebraic identities). To determine the expanded or factored form of an algebraic formula, we need to conduct some basic simplification and comparison operations.

Example: Algebraic expression

Let us consider the formula $(a+b)^2$, which is expanded into $a^2 + 2ab + b^2$ using the components of the formula (i.e. a,b and power 2), the expansion can be generated. Figure 37 shows the related code for this expansion.

Table 37 - Expansion of the formula $(a+b)^2$

```
public class Power extends Exp {
  ...
 public Exp simplify() {
   ...
   if(Controls.isExapnding() &&
       exp1 instanceOf Plus &&
       exp2.eval().equal("2")
   ){
     Exp A = ((Plus) exp1).exp1;
     Exp B = ((Plus) exp1).exp2;
     Exp N = new Num (2);

     Exp tmp1 = new Power(A, N);
     Exp tmp2 = new Times(N, new Times(A, B));
     Exp tmp3 = new Power(B, N);

     return new Plus(new Plus(tmp1, tmp2), tmp3);
   }
  ...
 }
}
```

Example: Algebraic factorization

Obviously it is more complicated to factor than to expend an algebraic formula programmatically. In this work, we construct a list with the components of the expression represented in an AST. After simplifying each component, the resulting components in the list are checked with the ones required for algebraic formulas. If a match is found, then the list is replaced by the factorization of the formula.

However, the solution process must execute under a terminating criterion, otherwise, the expansion or factorization step of intermediate evaluations might cause an infinite loop

through multiple invocations of simplification methods. The code for the simplifications of algebraic formulas is given in Appendix C.

Case Study 9: Equality of expressions

The *isEquals()* method checks if two expressions are equivalent. The elements of expressions can have different permutations; therefore, *isEquals()* should detect these differences. An example would be $3x + 12 + x^2$, $x^2 + 3x + 12$, and $x^2 + 12 + 3x$. Figure 46 shows the related AST for each of these expressions.

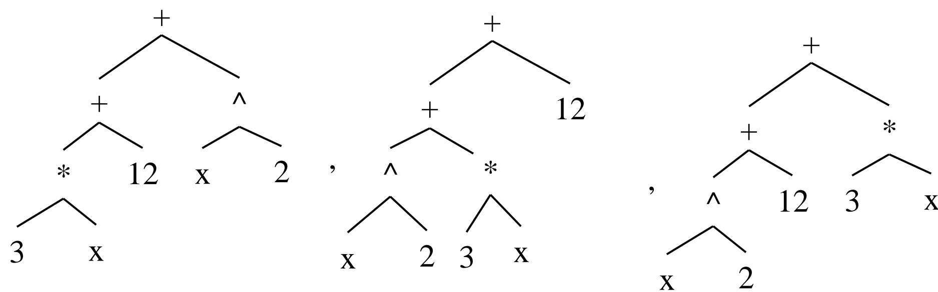


Figure 46 -Equivalent expressions of different ASTs

However, checking the similarity of expressions seems computationally exhaustive. One solution is to check all permutation of operators and operands and find a matching one. Another solution is to divide two equations; if two equations are equal, then the result will be one and true will be returned. Before division, if nominator and denominator are both zero, then *isEquals()* directly will return true.

Table 38 – Equivalence of mathematical expressions

```
public class Exp{
 public boolean isEqual(Exp e) {
   Exp a = this.simplify();
   Exp b = e.simplify();
   if(a.eval().equal("0") && b.eval().equal("0"))
     return true;
   Exp res = new Div(a, b);
   return res.eval().equal("1");
 }
}
```

4.8.3. Other Simplifications

In addition to those presented in the previous section, there are many other formulas and equations that can be simplified. Here are some transformations for which we develop some code in this work.

- $x^2 + (a + b)x + a.b = (x + a).(x + b)$
- $x^{2a} - y^{2b} = (x^a + y^b).(x^a - y^b)$
- $Sin\ (exp1 + exp2) = Sin(exp1) * Cos(exp2) + Cos(exp1) * Sin(exp2)$
- $a^2 - 2ab + b^2 = (a - b)^2$
- $a^3 + b^3 = (a + b).(a^2 - ab + b^2)$
- $a^3 - b^3 = (a - b).(a^2 + ab + b^2)$

However, it is not easy to correctly decide the selection of the related components of an expression for some simplifications. Each component can be a part of different simplifications. The result of the simplification may differ from what user expects which can be handled by some controlling methods that decide whether it should allow a formula be factored, expanded or ignored. The relevant codes for the controller are included in Appendix C.

4.9. Controlling the Step-By-Step Solution

One goal of the thesis is to help students or users to see the solution steps for a given mathematical expression. The solution comes into existence once a user evaluates the expression in an AST. However, a step-by-step solution will increase the user's understanding of handling an expression. Besides, for hundreds of random expressions, the solution can be produced via the tool prepared with this work, which will be helpful to increase the skill solving math for students.

Automatic evaluation and simplification on an expression are discussed in the previous sections. The solution process uses a method of the recursive implementation of simplification transformations.

The *eval()* method invoked by the root element of an AST recursively calls the *eval()* methods of the children. This process will continue, until it reaches to the leaves which are only numbers or variables. The return value of each *eval()* method is a number, variable or another expression node. It is worth mentioning that, on assigning values to variables, the result will be always a number. Finally the result of the evaluated nodes will be displayed to the user.

Simplifying an expression slightly differs from evaluating it, where all variables should be kept as strings, and also performing each transformation might change the view of the final output. Considering these two properties, at some point, depending on if user wants to see the steps of the solution, the transformations at each level must be reported.

There are two main techniques that can be used to show each transformation:

1. Print the transformation into an output buffer, and continue.
2. Return to root after performing each transformation

In the first technique, there should always be an object which we can use as a pointer to the address of the output buffer. However, it is also possible to use a static object, or a global method which is accessible by all AST nodes. Each node should call a method to report the changes and therefore, the method must be imported into the classes such as Plus, Multiply and Sin. Other approaches like addressing one node or retrieving parent nodes with the transformations to them also require extra methods and data to access the parent nodes.

However, in the second technique, a flag is used to check if a simplification should be performed or not. In this method, once a change is applied to the related AST, it triggers the flag and stops the simplification process for other nodes. The effect of this action will be only one change at a time, and therefore, it must be done with a loop. In each time, we reset the flag and then continue the process until there is no more simplifications to perform in which case the flag status stays the same. Table 39 shows the main loop for simplifications.

Table 39-The main loop for simplifications

```
public Exp performSimplification(Exp root){
  do{
    Controls.resetFlags();
    root = root.simplify();
    if(Controls.hasError() != null){
      System.out.println(Controls.getError());
      return null;
    }

    root.Print();

  } while ( Controls.isSkipSimplifying() );

  root.Print();

  return root;
}
```

The second technique guarantees that we obtain the exact AST at each iteration of the loop, so we can call the *Print()* method of the AST in one place. Table 40 shows the relevant code added to the *simplify()* method of each class.

Table 40 - Modifications for the *simplify()* method

```
public Exp simplify(){
  if(Controls.isSkipSimplifying())
    return this;
  ...
  // rest of the codes simplifying the element
  ...
  // in case of performing any
  // simplification call Controller.skipSimplification();
  ...
}
```

4.9.1. Expression Simplification Control

Each step of simplification can be controlled by a class called *Controls* which denotes which transformation to perform or not. As an example, we use a *skip simplification method*, *isSkipSimplifying()*, when we want to return to the main loop and report the current state of the AST. Another example is *hasError()* where we can perform illegal evaluations like division by zero, negative value for root square, etc.

Varieties of other flags are used to control each transformation. Some of these flags are listed in Table 41.

Table 41-Some of variables and their applications

Variable Name	Usage	Case Study
isSkipSimplifying	Status of simplification	All
isExapnding	Expand algebraic formulas	7
isSqrtRemoveal	Remove radicals in **denomination**	6
isPowerSimplify	Exponential simplification	5
isMergeFractions	Merge fractions	4
...		

These flags are customizable for each object. Table 42 shows an example of how to control the removal of a fraction in dominator.

Table 42**-**The control of the simplification of *radical removal* using control flags

```
public Exp simplify(){
  if(Controls.isSkipSimplifying())
    return this;
  ...
  if (exp2 instanceof Sqrt && Controls.isSqrtRemoveal()){
    Exp sq = exp2.exp1;

    // Notify the transformation to controller
    Controller.skipSimplification();

    return new Div(new Times(exp1, sq), sq);
  }
  ...
}
```

The same controls can be added to other objects as well. However, to control errors, some code will be added to *eval()* method of each class. *Division by zero* and other mathematical errors are semantic ones that should have to be prevented. Syntactical errors have been resolved once we create AST, so no possible kind of syntax errors will occur.

4.10. Determining Types of Mathematical Expressions

Several forms of mathematical expressions are covered in this thesis. Table 43 shows a list of all supported forms that an algebraic expression can have.

Table 43 - Some forms of covered expressions

exp	$f(x) = exp$
$f(exp) = exp$	$exp = exp$
$f(x) = exp = exp$	$f(exp) = exp = exp$
exp , *exp*	$f(x) = exp$, *exp*
$f(x)=exp$, $g(x)=exp$	$f(exp)=exp$, $g(x)=exp$
$f(exp)=exp$, $g(exp)=exp$	$exp=exp,exp=exp$
$f(x)=exp=exp$, $g(x)=exp=exp$	

To apply an operation for one kind of expression, the type of input data together with AST must be determined. The grammar proposed in Table 15 does not reflect the expression type. Therefore, it is equipped with extra rules which are shown in Table 44.

Table 44 - Additional grammar rules for various expression types

```
S={Comma}
Production:
<Comma> ⟶ <Equ> ("," <Equ>)? $
<Equ>   ⟶ (<Func> | <Exp> )(<Exp>)?
<Func>  ⟶ ("f"|"g"|"h") "(" <Exp> ") =" <Exp>
```

In Table 44 some additional classes are added including *Comma*, *Equ*, *Func*. As each expression starts with start symbol *S*, it should be considered as a *Comma* object. Then it can be an *Equ* object that it covers functional and non-functional expressions.

Table 45 - Additional rules in JavaCC format

```
Comma parse() : { Equ e1, e2=null;} {
  e1=equ()(< COM > e2= equ()) ? (<EOF> | <EOL>)
  return new Comma(e1,e2);
}

Equ equ(): { Exp a,b=null; Func c; } {
  a = expr()  (< EQU > b= expr())? { return new Equ(a,b); }
      |
  c= func()  (< EQU > b= expr()) ? { return new Equ(c,b); }
}
Func func() : { Exp a,b; } {
  < F >< LPR > a=expr() <RPR> <EQU> b=expr()
    { return new Func("f", a, b); }
  | < G >< LPR > a=expr() <RPR> <EQU> b=expr()
    { return new Func("g", a, b); }
  | < H >< LPR > a=expr() <RPR> <EQU> b=expr()
    { return new Func("h", a, b); }
}
```

A function name is limited to f, g, and h. Multiple expressions can be used in input string depending on if symbols ‘,’ or ‘=’ are used. The input grammar for JavaCC is also changed so that it can accept new features for input string.

Table 45 shows these additional rules.

Figure 47 shows an object representation of the expression "*f(x)*=3*x*+2".

```
Exp e1 = Plus(Times(Num(3), Var), Num(2));
Exp e2 = Var();
Func f1 = Func("f", e1, e2);
```

Figure 47 - Object representation of the expression "*f(x)* =3*x*+2”

It is obvious that there would be no need for an expression to contain functions, or equality expressions. In these cases, the objects of *Comma* and *Equ* classes will be used and others will be set to *null*. Table 46 shows an overview of objects that can be used for various expressions.

Table 46 - Object structure for supported mathematical expressions

			Comma											
			Equ						*Equ*					
			Func		*Exp*	*Func*		*Exp*	*Func*		*Exp*	*Func*		*Exp*
			Exp	*Exp*		*Exp*	*Exp*		*Exp*	*Exp*		*Exp*	*Exp*	
One Element	*Expression*	*Exp*			*exp*									
	Function Type 1	*f(x) = exp*	*Var*	*exp*										
	Function Type 2	*f(exp) = exp*	*exp*	*exp*										
	Equation Type 1	*exp = exp*			*exp*			*exp*						
	Equation Type 2	*f(x) = exp = exp*	*Var*	*exp*				*exp*						
	Equation Type 3	*f(exp) = exp = exp*	*exp*	*exp*				*exp*						
Two Element	*Expression*	*exp , exp*			*exp*						*exp*			
	Function Type 1	*f(x) = exp ,exp*	*Var*	*exp*							*exp*			
	Function Type 2	*f(x)=exp, g(x)=exp*	*Var*	*exp*					*Var*	*exp*				
	Function Type 3	*f(exp)=exp, g(x)=exp*	*exp*	*exp*					*Var*	*exp*				
	Function Type 4	*f(exp)=exp, g(exp)=exp*	*exp*	*exp*					*exp*	*exp*				
Misc.	*Equation Type 1*	*exp=exp, exp=exp*			*exp*			*exp*			*exp*			*exp*
	Equation Type 2	*f(x)=exp=exp, g(x)=exp=exp*	*exp*	*Exp*				*exp*	*exp*	*exp*				*exp*

For specific operations, the expression type needs to be identified. As seen in the diagram in Figure 48, it will be detected based on the objects that are used with each of *Comma*, *Equ*, and *Func*.

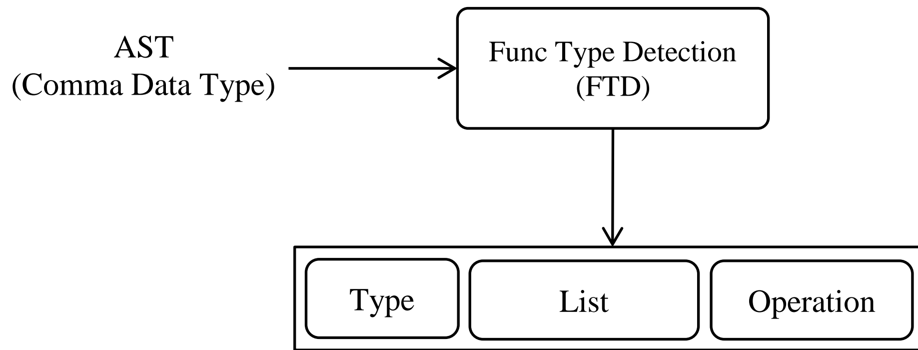


Figure 48- Detecting expression type

The output of FTD unit in Figure 48 is an object with three fields where Type, according to Table 46, indicates the type of an expression, List is a list of extracted Exp objects, and Operation is a list of valid operations such as simplification and derivation which can be applied on the related AST. In the next section, using the proposed methodology, these operations are explained.

4.11. Applications of the Methodology

Based on the type of an expression, various operations can be used. This section illustrates the examples of some particular operations.

4.11.1. Functions

Functions are a special kind of mathematical expressions that have a certain structure in our grammar. Function *f* with domain *X* and codomain *Y* is denoted by:

$$f: X \rightarrow Y \qquad \text{Or} \qquad X \xrightarrow{f} Y$$

Elements of *X* are called arguments of *f*. For each argument $x \in X$, corresponding $y \in Y$ in codomain is called the image of *x* under *f*. One may say that *f* maps *x* to *y* or associates *y* with *x*. Simple form is "$f(x) = Y$". Various forms of functions can be used with basic operations of algebra, listed in Table 47.

Table 47 - Operations on functions

Operation on two functions	Example
Summation	$(f + g)(x) = f\,(x) + g(x)$
Subtraction	$(f - g)(x) = f\,(x) - g(x)$
Production	$(f.g)(x) = f\,(x)\,.\,g(x)$
Division	$(\frac{f}{g})(x) = \frac{f(x)}{g(x)}$ *where* $g(x) \neq 0$

There is also another operation to combine functions, which is composition of two functions. The composition of $f(x)$ and $g(x)$ is symbolized as $(f \text{ o } g)(x) = f\,(g(x))$, pronounced as "*f of g of x*". The concept is simple. The value of *g* at *x* will be used as an argument for *f*. For example:

$$f\,(x) = 3x, \text{ and } g(x) = x + 2$$

$$(f \text{ o } g)(x) = f\,(g(x)) = 3(x + 2) = 3x + 6$$

$$(g \text{ o } f\,)(x) = g\,(f\,(x)) = 3x + 2$$

These five operations for two functions are supported if two forms of functions and an expression with one of five operands exist in AST. For example, parsing input string "$f(x)=(3x+sin(x))/\,x^2$" will return an AST demonstrated in Figure 49.

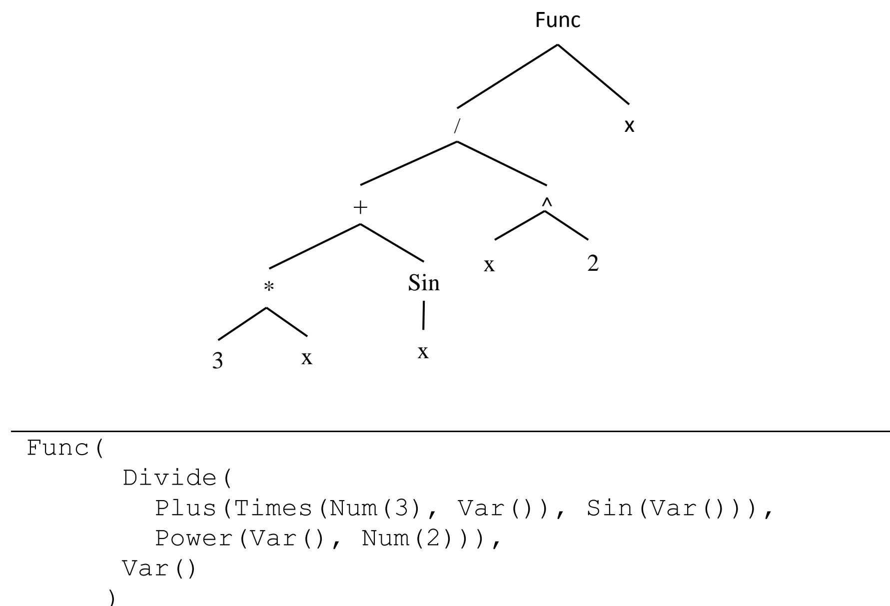


Figure 49 - Parse tree and object view for expression "$f(x) = (3x+sin(x))/\,x^2$"

The grammar to support functions in our main grammar is listed in Table 48.

Table 48 – A grammar for functions with values

```
< start > ⟶ f (<expr>) = <expr> [, [f|g](<expr>) = ("?"|<expr>)]?
```

The sample usages of this grammar are expressed in the following sections.

4.11.1.1. Function Value Evaluation

The grammar in Table 48 accepts mathematical expressions in the form of "$f(x) = Y$ " and "$f(x) = Y, f(Num) = ?$ ". For example, for "$f(x) = 3x + sin(x) / x^2, f(5) = ?$ " parser will generate an AST demonstrated in Figure 50.

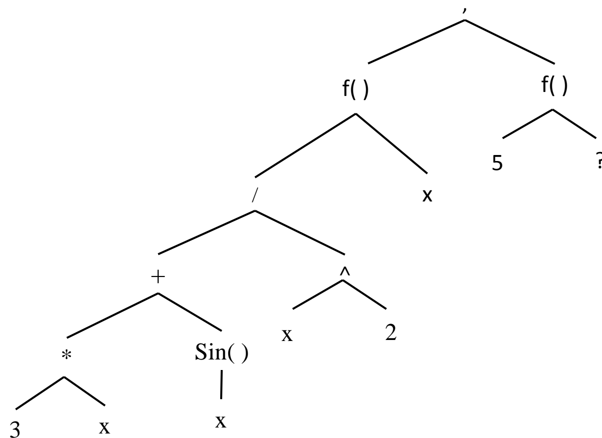


Figure 50 - AST for expression "$f(x) = 3x + sin(x) / x^2,\ f(5) = ?$"

All expressions in this form of function evaluation have the same AST. Every node of *Exp* objects has an *eval()* method that can evaluate its value by visiting its children. Now, in this model, we have a *Comma* node with two *Func* nodes. Using the left value of right Func object to evaluate the left Func object, it is possible to evaluate the function value.

Table 49 - Evaluation of a function value

```
public class Comma extends Exp{
   ...
   public Exp evalFuncValue(){
     double val = exp2.exp1.getNum();
     return exp1.eval(val);
   }
   ...
}
```

4.11.1.2. Operations on Functions

Similar to the previous section, it is possible to apply and evaluate extra operations on two functions. Basic functions as mentioned in Table 47 are evaluated in the same way as simple expressions without functions. Let us consider the following example:

$$f(x) = 5x - cos(x), \qquad g(x) = \frac{x^2 + 3}{12} \tag{1}$$

The AST for Eq. (*1*) is shown in Figure 51.

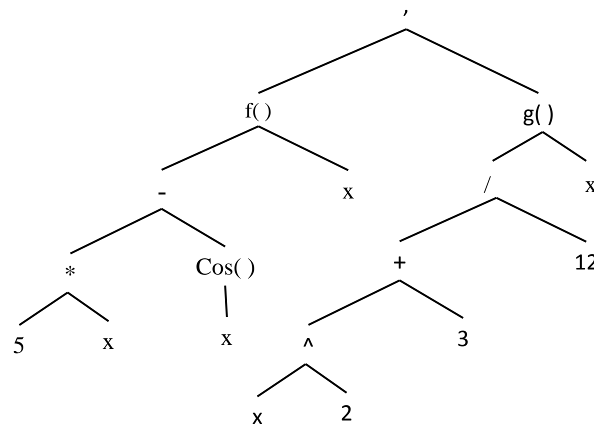


Figure 51 - AST of $f(x) = 5x - cos(x)$, $g(x) =(x^2+3)/12$

According to the AST in Figure 51, all function operations can easily applied on its nodes. Table 50 shows the related sample code to perform all operations.

Table 50 - Evaluation of a function value

```
public class Comma extends Exp{
  ...
  public Exp getFuncAdd(){
    return new Func(new Plus(exp1, exp2), Var.X());
  }
  public Exp getFuncMinus(){
    return new Func(new Minus(exp1, exp2), Var.X());
  }
  public Exp getFuncMul(){
    return new Func(new Mul(exp1, exp2), Var.X());
  }
  public Exp getFuncDiv(){
    return new Func(new Div(exp1, exp2), Var.X());
  }
  public Exp getFuncCompose(){
    return new Func(exp1.eval(exp2), Var.X());
  }
  ...
}
```

The modified *Comma* class in Table 50 produces the summation of two functions in Figure 51, which is illustrated in Figure 52.

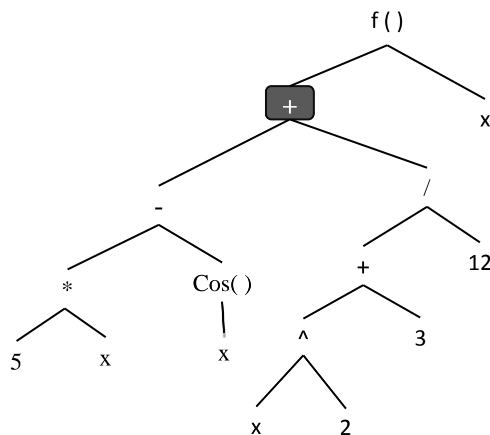


Figure 52 – The summation of two functions

The *Subtraction*, *Multiplication* and *Division* operations work in a way similar to those of *Summation*. However, the composition of two functions is similar to the evaluation of *function value* discussed in section 4.11.1.1 where the codomain of the second function is used.

4.11.2. Equations

A function expression starts with a function and some "=" signs. However, for equations like an expression "*x*+3 = 9", The AST must be marked as a non-functional equation which is evaluated if two algebraic expressions are equal. The left expression of "=" is called the left-hand side and right-hand side expressions of it, is called the right side of the equation. The result of an equation might be a *true* sentence like "5+2=7", a *false* sentence like"6-3=5", or an open sentence like "*x*+3 = 9" where it should be solved to find the value or root of *x*.

Two types of equations are covered in this thesis, *first degree equations* and *quadratic equations*. The main difference is that in the first degree equations, the power of variable *x* is 1, as it is 2 or less in quadratic equations.

4.11.2.1. First Degree Equations

Additional *LL(1)* grammar rules are used in Table 51 to parse *first degree equations*.

Table 51 - Grammar rules for first-degree equations

```
<start>   →   <expr> " = " <expr>
<expr>    →   <term> { ("+"|"-") <term> }*
<term>    →   <num>   ("x")? | "x" (<num>)?
```

This grammar accepts expressions in the form of "*expression* = *expression*". The normal form of first degree equations are "*ax*+*b* = 0" where *a* and *b* are numbers and the power of variable x should have to be 1. Input equation has to be simplified to normal form. Figure 53 shows an example of the expression "3*x*+(4-*x*)*2 = 3-5*x*". Note that 3*x* differs from 3**x* and it is used as a compound term.

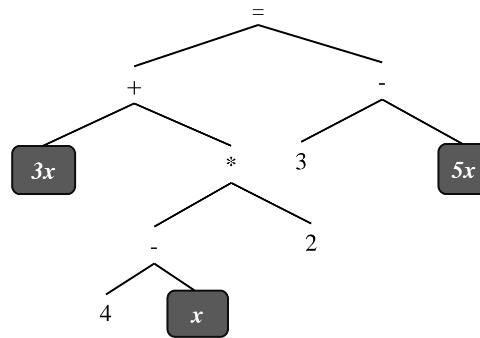


Figure 53 - AST of the expression "3*x* + (4-*x*)*2 = 3-5*x*"

Simplification and the solution to *first degree equations* require identifying "=". Also class *Equ* is used to keep track of the left and right side of the equation. The root of AST in Figure 53 is an Equ object.

Table 52- Equ class definition in java

```
public class Equ extends Exp{
  public Equ Simplify() {
    Data d1 = exp1.Simplify().eval();
    Data d2 = exp2.Simplify().eval();
    double a = d1.a() + d2.a();
    double b = d1.b() + d2.b();
    Return new Equ(Var.X(), new Num(b / a));
  }
  public String Print() {
    String a = exp1.Print();
    String b = exp2.Print();
    return a + "x = " + b;
  }
}
```

Class *Data* is a simple class to extract *a* and *b* parameters of "$ax+b = 0$". The related code for this class is given in Appendix C.

4.11.2.2. Quadratic Equations

A *quadratic equation* is in the form of "$ax^2 + bx + c = 0$", where *a*, *b*, and *c* are numbers and $a \neq 0$ or else the equation will be a *first degree* one. The values of *a*, *b*, and *c*, similar to a first degree equation, are extracted by the *Data* class. Table 53 shows additional grammar rules for this kind of expressions.

Table 53 - Grammar rules for quadratic equations

```
<start>     →   <expr> " = " <expr>
<expr>      →   <term> { ("+"|"-") <term> }*
<term>      →   {<term1>|<<term2>} {"*" <term3>}*
<term>      →   <term2> "*"<term2>
<term1>     →   <element> ("^2")?
<term2>     →   <element>
<term3>     →  <num>
<element>   →  <num> | <"x">
```

Various methods exist to solve quadratic equations such as factoring by inspection, completing the square, and discriminant. A quadratic equation with real or complex coefficients has two solutions, called roots. These two solutions may or may not be distinct, or real. In this thesis, the discriminant method is used.

The value of *Delta* or Δ which is called *discriminant value* is required for calculations where:

$$\Delta = b^2 - 4ac \tag{2}$$

The discriminant determines the number of roots and their nature where:

$$\begin{cases} x_1, x_2 & 0 < \Delta \\ x_1 = x_2 & 0 = \Delta \\ NaN & 0 > \Delta \end{cases} \tag{3}$$

x_1 and x_2 are defined below:

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a}, \qquad x_2 \frac{-b - \sqrt{\Delta}}{2a} \tag{4}$$

There has to be an *x-square* term in AST. Therefore, it should be simplified to extract this term and a, b, and c as well. Figure 54 shows the AST of the expression "$2+3x^2-x=12+x^2$".

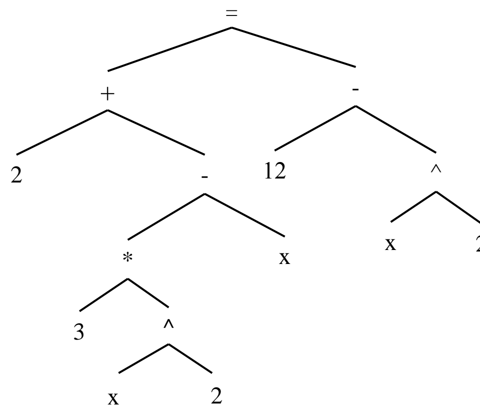


Figure 54 – The AST of the expression "2+3 x^2-x = 12+x^2"

The steps of transforming a quadratic equation into its general form are discussed in section 4.11.3. After the transformation, it is possible to obtain the values of *a*, *b* and *c*. The roots for the equation will be calculated using *Eq.* (4).

4.11.3. Polynomials

A polynomial is an expression containing a variable or indeterminate and coefficients. Polynomial expressions use addition, subtraction, multiplication operators and non-negative integer for exponentials of power operator.

$$\sum_{i=0}^{n} a_i x^i = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0, \qquad 2 \leq n \qquad (5)$$

where i and n are a non-negative integer, a_n, … , a_0 are real numbers, and x is a variable. A polynomial is the summation of monomials. A monomial, like "$3x^7$" can be a number or product of numbers and variables with an exponent. The number part of the monomial is called coefficient.

4.11.3.1. Standardization of Polynomials

A polynomial Input string must be converted to standard form, which needs to become a standard monomial. Also monomials must be arranged according to the power of

variable x or the degree of the term. Figure 55 shows the AST output of our parser for a polynomial expression. Each monomial node is a sub-tree of *Prod* object for polynomial terms.

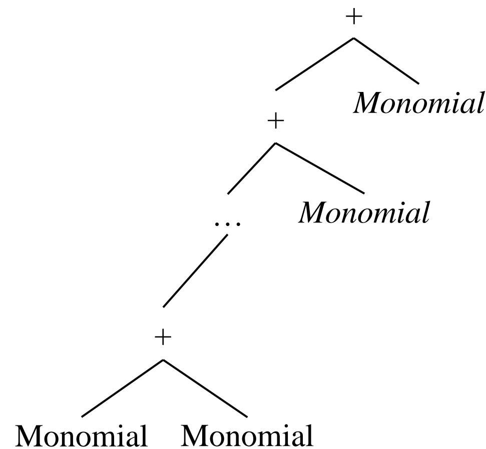


Figure 55 – The AST of a polynomial expression

Standarding a polynomial starts with standardization of each monomial. Therefore, each term of a polynomial should be simplified and turned into "$a_i x^i$" form, where a_i as an *Exp* object will be a product of x^i in an AST. Class *Data* will simplify each monomial. The code of this class is presented in Appendix C. The next step is to order each monomial according to its degree. This is done by putting all monomials into an array list and sorting it. Figure 56 shows this issue.

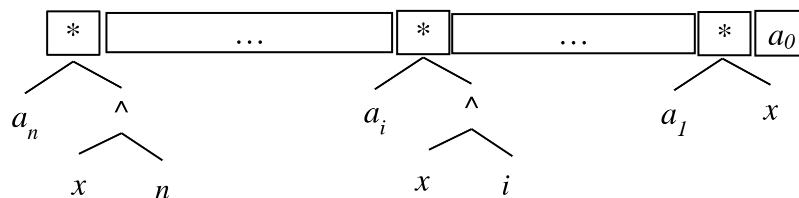


Figure 56 - Array of monomials for a polynomial

The AST of a polynomial in an array is obtained from the *Plus* class. According to Figure 55, AST nodes are Plus objects. Therefore, we will use the *sameOpList()* method from Table 30. To extract the exponential value of monomials, the code shown in Table 54 are used.

Table 54 – The codes for extracting exponential value of monomials

```
public static int getMonomialDegree(Exp e){
  int result=0;
  if(exp2 instanceof Power)
    result = ((Power) exp2).exp2.getNumValue();
  else if ( exp2 instanceof Var)
    result = 1;
  return result;
}
```

Sorting the array in Figure 56 and converting it back to an AST on *Plus* nodes will standardize the polynomial expression.

4.11.3.2. Division of Polynomials

Polynomial long division is dividing a polynomial by another polynomial of the same or lower degree. Suppose that there are two monomials as follows:

$$a_i x^i \quad , \quad a_j x^j \tag{6}$$

The division of two monomials is expressed in Table 55.

Table 55 – The division of two monomials

```
function getMonomialDivision(aixi, ajxj)
  while(j≠0 and i≥j){
    b = a_i / a_j
    m = i − j
  }
return  bx^m
```

In the *Euclidean Division*, which is used in this thesis, the polynomial division provides:

$$P_1 = P_2 Q + R, \qquad \deg(P_1) \leq \deg(P_2)$$

Where Q is a quotient and R is a remainder. Q and R can be polynomials as well. The method *deg* returns the highest degree in its polynomial argument. For P_1 and $P_2 \neq 0$ that are two standard polynomials, the algorithm in Table 56 will return Q and R respectively:

Table 56 – The division of two polynomials

```
q_0 = 0
r_0 = P_1
i = 1
Repeat {
```

$$q_i = q_{i-1} + \frac{LC(r_{i-1})}{LC(p_2)} x^{\deg(r_{i-1}) - \deg(P_2)}$$

$$r_i = r_{i-1} - \frac{LC(r_{i-1})}{LC(p_2)} x^{\deg(r_{i-1}) - \deg(P_2)} . P_2$$

```
} Until deg(r_i) < deg(P_2)
Return (q_i and r_i)
```

In Table 56 *LC()* returns the coefficient of highest monomial in the given polynomial. If each r_i is a sorted array list of monomials, then finding *LC* value will be easy. It is necessary to mention that the coefficients of polynomials are fixed numbers, but it is possible to generalize this theorem to the case when the coefficients are other variables. Table 57 shows an example of polynomial division.

Table 57 – The polynomial division of the expression"$(x(2x+1)+3x^3) / (x+1)$"

P_1	P_2	Q	R
$(x(2x+1)+3x^3)$	$(x+1)$		
$3x^3+2x^2+x$	$x+1$	$3x^2$	$-x^2+x$
$-x^2+x$	$x+1$	$3x^2-x$	$2x$
$2x$	$x+1$	$3x^2-x+2$	-2
-2	$x+1$	$3x^2-x+2$	-2
Results:		$3x^2-x+2$	-2

4.11.3.3. Greatest Common Divisors of Polynomials

The GCD of two polynomials is a polynomial of the highest degree, which corresponds to a factor of both polynomials. The GCD can be achieved with the code expressed in Appendix C. This indicates that the GCD of polynomials is the same as the GCD of integers. All rules for the GCD of integers is true for polynomials as well.

4.11.3.4. Least Common Multiple of Polynomials

The LCM of two polynomials is similar to the LCM of two integers, which is discussed in Appendix C.

4.11.3.5. Polynomial Factoring

One of intermediate operations such as simplification, GCD and LCM, etc. is polynomial factoring. There are several methods for factoring a polynomial, which is mentioned below:

- *Greatest Common Factor Method*
- *Synthetic Division*
- *Factoring by Grouping*
- *Factorization of Perfect Square*
- *Factorize the Difference of Two Squares*
- *Factorization of Quadratic Trinomials*

In this thesis we discuss Greatest Common Factor method and Synthetic Division.

4.11.3.6. Greatest Common Factor method

This method relies on the GCD of all monomials. The idea is similar to the GCD of numbers. The Greatest Common Factor algorithm is listed in Figure 57.

```
M = poly.monomials;
sort(M)
gcf = M1
for i = 2 to M.length {
    gcf = GCD(gcf, Mi)
}
return product(gcf, div(poly, gcf).Simplify())
```

Figure 57 – The GCF of polynomials

In Figure 57, after fetching all monomials from a polynomial expression, it will be turned into a sorted list of elements. The initial value for gcf is the first element of the list. Then, this value will be used to collect other elements in the list. The final value will be a product of the GCF and the division result of poly over GCF. The following is an example of GCF:

Table 58 – The GCF of the expression" $35a\,b^2 - 75a^2\,b + 15ab$"

Polynomial	*GCF*
$35a\,b^2 - 75a^2\,b + 15ab$	$35a\,b^2$
$35a\,b^2 - 75a^2\,b + 15ab$	*5ab*
$35a\,b^2 - 75a^2\,b + 15ab$	*5ab*
$35a\,b^2 - 75a^2\,b + 15ab$	*5ab*
Return: 5ab (7b - 15a +3)	

4.11.3.7. Synthetic Factoring method

Synthetic Factoring is similar to synthetic division. The idea is to find the roots of a polynomial and write it as:

$$Standard\ Polynomial = \prod_{i=1}^{k} (x - c_i) \tag{7}$$

Where k is the number of roots, and c_i is a root for a given polynomial. Here is an example of how this method works:

$$x^5 - 4x^4 - 12x^3 + 34x^2 + 11x - 30 = (x-1)(x-5)(x+3)(x-2)(x+1) \tag{8}$$

Synthetic factoring is preferred in this thesis and we will not go into details of required steps for it. However, it is implemented, and the related codes can be found in Appendix C.

4.11.4. Derivation

AST is used to find the derivation of an expression, which is defined as below:

Definition**:** A Generalized Syntax Directed Derivative (*GSDD*) is $F = (G, \Delta, R_c)$, where:

1. *$G = (\Sigma, T, V, P, S)$ is a proper Context Free Grammar.*
2. *$\Delta = \Sigma$ is a finite set of output symbols.*
3. *R_c is a function implemented according to* [91,94] *where:*
   $R_c: L(G) \rightarrow L(G)$
   $L(G) = \{w \mid w \in \Sigma^*, S \underset{G}{\rightarrow} w\}$
4. *If U= L1 and V = L2, then*
   $R_c(U \pm V) = R_c(U) \pm R_c(V)$
   $R_c(U \times V) = R_c(U) \times V + U \times R_c(L_2)$

$$R_c\,(U\,/\,V) = (R_c\,(U) \times V + U \times R_c\,(V))\,/\,(V^2)$$
$$R_c\,(U^V) = U^V \times (V \times R_c(U)\,/\,U + R_c\,(V) \times \mathrm{Ln}(U))$$
$$R_c\,(Sin(U)) = R_c(U) \times Cos\,(U)$$
$$R_c\,(Cos(U)) = -R_c(U) * Sin\,(U)$$

R_c is a method defined in each *Exp* object, returning the derivation of that object. Therefore, as it is shown in Rule(4) of the definition, R_c is a recursive method. Table 59 shows the *Derivation()* method for some classes.

Table 59 –The derivation() methods to get the derivation of an object

Class	Derivation() method
Exp	`public Exp Derivation( );`

Plus	`public Exp Derivation() {`
	`  Exp U = exp1;`
	`  Exp V = exp2;`
	`  Exp rU = U.Derivation();`
	`  Exp rV = V.Derivation();`
	`  return new Plus(rU, rV).Simplify();`
	`}`

Times	`public Exp Derivation() {`
	`  Exp U = exp1;`
	`  Exp V = exp2;`
	`  Exp rU = U.Derivation();`
	`  Exp rV = V.Derivation();`
	`  return new Plus(new Times(rU, V),new Times(U, rV)).Simplify();`
	`}`

Power	`public Exp Derivation() {`
	`  Exp U = exp1;`
	`  Exp V = exp2;`
	`  Exp rU = U.Derivation();`
	`  Exp rV = V.Derivation();`
	`  Exp A = new Times(new Divide(V, U), rU);`
	`  Exp B = new Times(rV, new Ln(U));`
	`  return`
	`    new Times(new Power(U, V), new Plus(A, B))).Simplify();`
	`}`

Divide	`public Exp Derivation() {`
	`  Exp U = exp1;`
	`  Exp V = exp2;`
	`  Exp rU = U.Derivation();`
	`  Exp rV = V.Derivation();`
	`  Exp A = new Minus(new Times(rU, V),new Times(U, rV));`
	`  Exp B = new Power(V, new Num(2));`
	`  return new Divide(A, B).Simplify();`
	`}`

Sin	`public Exp Derivation() {`
	`  Exp U = exp1;`
	`  Exp rU = U.Derivation();`
	`  return new Times(rU, new Cos(U)).Simplify();`
	`}`

Var	`public Exp Derivation() {`
	`  return new Num(1);`
	`}`

In Table 59 the result of the derivation is not the final one, therefore, a simplification is required. For this reason, we return the simplified form of the derivation by calling the *Simplify()* method of *Derivation()*.

Example: Table 60 shows the derivation of expression " 2^x".

Table 60 –The *derivation()* method to get the derivation of an object

Expression	Object View
2^x	`Power(Num(2), Var.X())`

2^xLn2	`U = 2`
	`V = x`
	`rU = 0`
	`rV = 1`
	`A = Times(Divide(2, x), 0)      # 0`
	`B = Times(1, Ln(2))             # ln(2)`
	`Time(Power(2, x), Plus(A, B))   # 2^x(0+ln(2)) = 2^xLn(2)`

The general notation of using AST for derivation enables it to solve various mathematical problems, such as Taylor series, Maclaurin Series, etc. We will discuss Taylor series here to show an application of derivations using AST.

Example: Taylor series is a polynomial representation of a function as an infinite sum of terms that are calculated from the values of function's derivation at a single point. For *f(x)*,

$$f(x_0) + \frac{f^{(1)}(x_0)}{1!}(x - x_0) + \frac{f^{(2)}(x_0)}{2!}(x - x_0)^2 + \frac{f^{(3)}(x_0)}{3!}(x - x_0)^3 + \cdots \qquad (9)$$

Where $f^{i)}$ is the i^{th} derivation of *f(x)*. To calculate Taylor series of *f(x)* a system of the following components is required:

- *A method to calculate the derivation of expressions (Section 4.11.4)*
- *A method to simplify expressions (Section 0)*
- *A method to calculate the value of an expression at a given point (Section 4.11.1.1)*

Having all required components, it is easy to calculate the Taylor series for any expression. One can create an array and collect desired number of terms of this series into it. The calculation of $f^{i)}$ and Factorial of iterator *i* are the main part of the algorithm. The implementation of calculating Taylor series is provided in Appendix D.

4.12. Discussion

In this section we introduced our proposed methodology as a step-by-step solutions for mathematical expressions. An extended-BNF grammar is used to accept algebraic expressions. Using a compiler-compiler, we converted our grammar into a parser. Because of strong support for object orientation, independent from any particular operation system, and popular language in academic world, we used Java. We used JavaCC to generate a scanner and parser in Java.

JavaCC uses LL(k) grammars for input, so we prefered to use a LL(1) grammar. Default output parser of JavaCC is used to produce an Abstract Syntax Tree or AST, where it is more easy and powerful to manipulate the relations between operators and operands in a mathematical expression.

Modeling relations in an AST is helpful to analyses and implement desired operations on its nodes. Automatic simplifications are groups of translations where it replaces a complex relation between nodes with equivalent but simple relations.

Representation of AST as a data structure into something more meaningful for human is discussed. Because of tree structure and recursive property of AST, it is easy to produce proper components such as human-readable string, LaTex and MathML.

Various operations, depending on the type of input expression, is used. Therefore, a system to detect the expression type implemented. This empower the proposed system to use more complex and powerful operations such as operations on functions possible.

To show the power and usability of proposed methodology, some applications are demonstrated. Various operations on functions, solving first degree and quadratic equations, division and factoring polynomials, and derivations of expressions are discussed.

In next section, to complete the idea of solving mathematical expressions, we will present a method to generate these expressions.

5. AUTOMATIC PRODUCTION OF MATHEMATICAL EXPRESSIONS

There are situations that one needs to write various kinds of mathematic questions, such as practicing tests, school exams, and function optimization algorithms. Some systems may use a database of mathematic equations where all expressions are written once and then it is being used many times. Solutions or answers of such questions might be stored, but it will not be a dynamic system, and no variety in expressions. There are various ways for random generation of mathematic expressions. Depending on involving operators and operands, variables, different types of generators can be implemented. Unfortunately, it is not possible to control those kinds of expressions that require satisfying conditions; hence there are methods that can control limited part of generating expressions.

This chapter addresses a grammar based methodology to automatically generate mathematical expressions such as first-degree and quadratic equation, polynomials, and limits. The generating process is very closely related to the synthesis of expressions. Synthesis is obviously the reverse of analysis, as it signifies the taking the terms of which an expression is composed, and combining them together, so as to produce that expression. Since both analysis and synthesis involve description of a language via grammar rules, it is possible to use analysis grammars also for synthesis to some extent. However, these activities exhibit two different kinds of indeterminacy; analysis involves determining which of possible representations best fits a particular expression, synthesis involves determining which of possible expressions best fits a particular representation. We can see synthesis as just a matter of linearization plus grammatical realization rules. In general, compared to analysis, there is no problem of ambiguity in synthesis.

5.1. Methods for Generating Expression

The Simplify way to produce random math expressions is to use a well-formed generation algorithm on the space of operators and numbers. Table 61 shows such an algorithm.

Table 61 – An algorithm for the random generation of an arithmetic expression

```
Str ← generate a random number
while condition
    Operator ← select a random operator from {+, -, ×, /}
    Operand  ← generate a random number
    Str      ← Str . Operator . Operand
```

The algorithm generates algebraic expressions, randomly selecting an operator from the set [+, -, *, /] and combining it with the operands of numbers. An example expression would be "23+5×3-2/3+10".

A similar algorithm is given for polynomial expressions in Table 62. With an initial value of degree n, the algorithm can generate polynomials that can have up to n terms. A typical example is the polynomial "$3x^5 - 8x^2 + 9$".

Table 62 – An algorithm for the random generation of polynomial expression

```
n ← A random number as the length of polynomial
c ← Generate a random number
polyExp ← c . 'x^n'
n--
while n > 0
   c ← Generate a random number
   if c>0 : polyExp ← polyExp . '+' . c . 'x^n'
   if c == 0 ;
   if c<0 : polyExp ← polyExp . '-' . c . 'x^n'
   n--
```

These algorithms can be improved adding some other attributes such as parentheses. But, in this case, the generation process of expressions must be held under control to conduct different kinds of evaluation or interpretation.

In general, we need a special structure in accordance with the recursive nature of generating on expression. This structure must have all the properties of binary trees. Figure 58 shows the expression tree of "$((x+y)/2)\times(a+b)$-12".

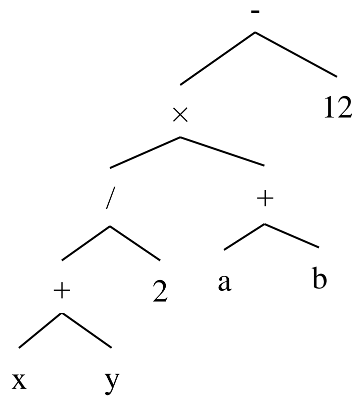


Figure 58– The binary tree of the expression "$((x+y)/2)\times(a+b)$-12"

The binary tree is a suitable data structure which supports the inclusion of algebraic operators and single parameter functions. It provides a simple way to represent operator precedence. Besides it is easy to convert the tree structure to other data formats for the requirements of document formatters. Another advantage is about the development formal grammars because each node of the tree suggests a recursive invocation among parent and child ones constructed by a grammar rule.

Figure 59 shows a block diagram of mathematical components required to create a random mathematical expression using trees.

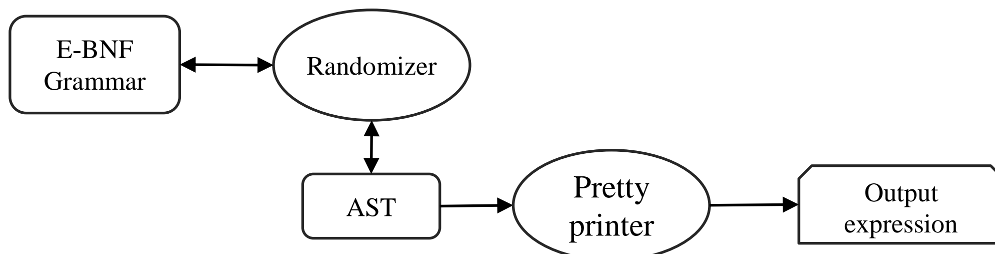


Figure 59- The methodological components for random tree-based generation of expressions

In Figure 59, The Randomizer is a rule selector that first places the beginning rule of a grammar into an AST and then recursively expands the non-terminals in that rule replacing them with the other possible rules until encountering a non-expandable rule. Table 63 shows an example of grammar rules which can be placed into an indexed table for an easy selection.

Table 63-Grammar rules for randomly generating mathematical expressions

Non-terminals	Rule
S	E
E	E+E
E	E-E
E	E*E
E	E/E
E	Sin(E)
E	Ln(E)
…	
E	D
E	V
…	
D	0|1|2|…|9

Note that the rules presented in Table 63 do not consider the precedence of operators because it is a matter of the evaluation phase. Table 64 displays the main algorithm of the methodology performed by the Randomizer component of Figure 59.

Table 64 – The random expression generating algorithm

```
AST ← Start symbol
While a Non-Terminal exists in AST
   E    ← Select a non-Terminal in AST
   sub  ← Expand E using a random rule
   AST  ← Replace E with sub in AST
In-order traverse the AST and print mathematical expression
```

In Table 64, it is possible to invoke a sequence of the non-terminals, thus rules, in the order of S, E and D. For example, resulting AST and expression would be Num(7) and "7", respectively.

However, the expression of the rules like "$S \rightarrow E \rightarrow E+E \rightarrow E+E+E \rightarrow E+E+E+E$..." can create an unlimited number of non-terminals. Stochastic Grammars [95] can be used to control the status of the invoked rules. Initializing an invocation probability for each rule in the algorithm, controlled through the expression steps, we can restrict the number of non-terminals expanded for an AST to a finite value. In this way, some restrictions might be imposed on the following structural parameters.

- *The length of output expression*
- *The number of nodes in AST*
- *The number of levels in AST*

These parameters determine the expression level of the AST, generating mathematical expressions of the desired characteristics. Table 65 shows an improved version of the algorithm presented in Table 64.

Table 65-An improved algorithm based on the one in Table 64

```
AST ← Start symbol
N   ← Number of maximum nodes
n   ← 0
While a Non-Terminal exists in AST
   E    ← Select a non-Terminal in AST
   if(n<N)
     sub  ← Expand E using a random rule
   else
     sub  ← Expand E using a random rule without a non-terminal
   AST  ← Replace E with sub in AST
In-order traverse the AST and print mathematical expression
```

5.2. Statistical Space Analysis

Mathematical expression generated by the provided grammar in Table 15 can be selected randomly or can be selected from infinite number of expressions. If we limit generated expressions by identified mechanisms such as maximum nodes or maximum level of tree, it is possible to convert infinite space of the problem to finite space.

Suppose that applied limit of producing mathematical expressions related to the number of nodes. For a tree with *n* nodes, one of these nodes is the root node, and the rest of $n-1$ internal nodes are children or parent of underlying nodes. Obviously, there is one way to make a binary tree with zero or one node. Equ. (10), calculates the number of binary trees with *n* nodes, shortly called Catalan Numbers [96].

$$S_n = \frac{1}{n+1} . \binom{2n}{n} = \frac{2n!}{n!\,(n+1)!} \qquad (10)$$

The total number of trees which can contain up to n nodes is calculated by:

$$S_{total} = \sum_{n=1}^{N} \frac{2n!}{n!\,(n+1)!} \tag{11}$$

Where, N denotes the maximum number of nodes. Based on the type and location of operators, operands and functions in an AST, there can arise various combinations that lead to different mathematical expressions. It is not easy to calculate the variations of AST using only the number of nodes. We can determine it in terms of the depth of the tree.

The number of possible ASTs at a single level is calculated by the following.

$$S_1 = v + d \tag{12}$$

Where v and d are the numbers of variables and integers respectively, which can construct a single leaf of the AST. Figure 60 shows possible AST for a maximum of up to 2 levels.

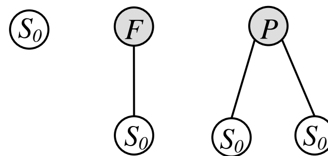


Figure 60 - Possible AST with up to 2 levels

So, the number of AST can be given as follows:

$$S_2 = S_1(1 + F + P \times S_1) \tag{13}$$

Where F is the number of functions such as *sin* and *ln,* and P is the number of operators such as "+" and "*". Figure 61 demonstrates the ASTs constructed through up to 3 levels.

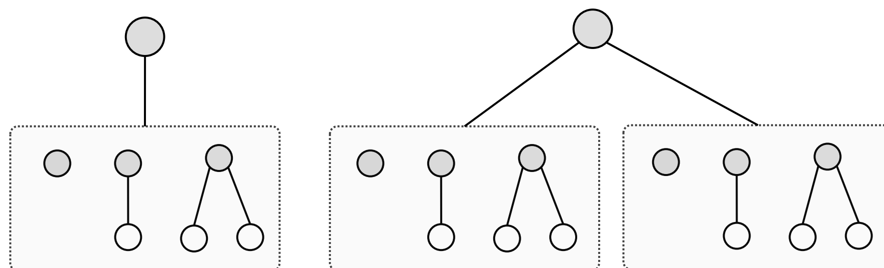


Figure 61-Possible AST with up to 3 levels

The number of such ASTs would be calculated by the relation:

$$S_3 = S_2(1 + F + P \times S_2) \quad (14)$$

In a general, we can give the following relation for the number of possible ASTs based on up to L levels.

$$S_L = \begin{cases} v + d & L = 1 \\ S_{L-1}(1 + F + P \times S_{L-1}) & L > 1 \end{cases} \quad (15)$$

Table 66 shows the number of the ASTs for some small values of L, assuming that there are P=*5* operators, $F = 8$ functions, $v = 1$ variable, and $d = 100$ digits.

Table 66–The numbers of possible ASTs with some different levels

Level	Number of Trees
1	102
2	52938
3	14,012,635,662
…	
7	6.743220307892116e+172
8	2.273551006038432e+346

The amount of ASTs which can be created at 8 levels is incredibly high. It means that the method can produce many different ASTs. However, some ASTs would contain expressions that are not worth constructing. Whit this aspect, it works an algorithm that generate expressions without using ASTs. On the other hand, we need some control rather than the number of nodes or levels in a generated expression. For example, the rule-invoking steps must be ended in a reasonable amount of time, without waiting for the generation of a particular type of expressions.

5.3. Type-specific Grammars for Expressions

The generating and evaluating methods of expressions vary widely from a subject of mathematics to another. So, it is clear that a different kind of expressions would require a different grammar. In this section, we first focus on the linear (first-degree) and quadratic (second degree) equations and then introduce a new grammar called a rule-iterated context free grammar.

5.3.1. Grammars for Polynomials

In the first-degree equation, the left and right hand sides of the symbol "=" continue a polynomial of degree 1 in a variable *x*. A first degree polynomial can be constructed in a few ways. Table 67 presents a CFG grammar that considers all possible ways of generating polynomials of degree 1. For example, the expression $\frac{x-1}{1-x} = 1$ generated by the grammar would correspond to the first degree equation "x-1 = 1-x".

Table 67 – A CFG grammar for first-degree equations

```
G={N,T,P,S}
N={E, E',S, var, number, digit}⊆ Σ
T={x,+,-,*}⊆Σ
Productions:
S  ⟶ E "=" E | E "/" E "=" E'
E  ⟶ E "+" E | E "-" E
     | number | var
E  ⟶ E  "*" E' | E'  "*" E
E' ⟶ E' "+" E'|E' "-" E'
     | E' "*" E' |  Number
Var ⟶ x
Number ⟶ "-"? [digit] + ["." [digit]+]?
digit ⟶ ["0" - "9"]
```

Note that the alternative production rule of E with the division operator (/) represents an equation with a constant number on the right hand side. Another grammar is given for quadratic equations in Table 68.

Table 68 – A CFG grammar for quadratic equations

```
G={N,T,P,S}
N={E,E',E'',S,var,number,digit}
T={x,+,-,*,/,^}
P:
S  ⟶ E "=" E
E  ⟶ E "+" E | E "-" E
      |number | var |var^2
E  ⟶ E' "*" E' |E "*" E'' |E'' "*" E
E' ⟶ E' "+" E' |E'  "-" E'
      |E' "*" E'' |E'' "*" E'
      |number | var
E'' ⟶ E''  "+" E'' |E'' "-" E
      | E''  "*" E'' |  Number
var ⟶ x
number ⟶ "-"? (digit)+ ["." (digit)+]?
digit ⟶ ["0"-"9"]
```

The equations generated by the grammars in Table 67 and Table 68 contain structural control on neither the length of the output expression nor the number of the operators. The restrictions discussed in the previous section must be considered to hold the expression generation under control. The next section, proposes a new grammar in which the select controlled.

5.4. Rule-Iterated Context-Free Grammar

In order to meet the user-defined restrictions, we introduce some specific control structures, manipulating the current set of the grammar rules. The rule-based grammars, such as a CFG, do not have convenient structures to limit the number of rule invocations to some certain value. Such a grammar is given in Table 69.

Table 69-A sample grammar with no specific control structures

```
E → E + E | D
D → 0|1|..|9
```

The grammars in Table 67 or Table 68 have the potential to produce an unlimited number of equations. The production process must be terminated via a grammar which would be deterministic in terms of rule invocations. Therefore, we propose a rule-iterated context free grammar, shortly called RI-CFG.

A RI-CFG is a system that G = <T, N, P, S>, where *T* and *N* are disjoint finite non-empty sets of terminals and non-terminals, respectively, S∈N is the start symbol, and *P* is a non-empty finite set of rules. Each set of rules in P has the form of *"u → v, n"*, where $u \in N$, $v \in (N \cup T)^*$, and *n* is a positive integer which indicates the number of rule selections in a non-terminal expression.

Let us denote by n the number of rule-specific selections. In this case, for n=1, the system behaves like a CFG. Table 70 shows a small example of a RI-CFG grammar for addition expressions.

Table 70 - A sample RI-CFG grammar

```
S → DT,1
T → +D,2
D → 0|1|..|9, 2
```

Using the grammar in Table 70, the expression "12+63+47" can be derives as seen in Table 71.

Table 71 – A sample derivation with the grammar in Table 70

Production	Rule
S	-
DT	S → DT,1
D + D + D	T → +D,2
12 + D + D	D → 0|1|..|9, 2
12 + 63 + D	D → 0|1|..|9, 2
12 + 63 + 47	D → 0|1|..|9, 2

The derivation of the resulting expression in Table 71 is demonstrated in the form of a generating tree in Figure 62.

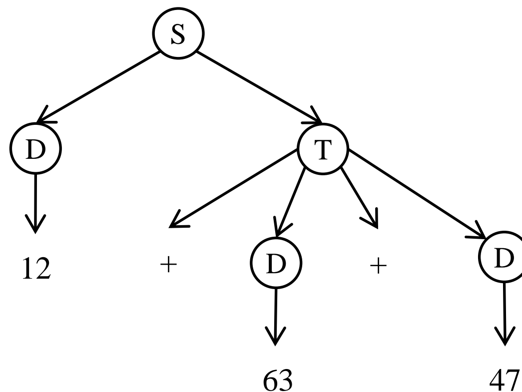


Figure 62 – The generating tree for the expression "12+63+47"

The use of RI-CFG grammars facilitates the control of the generating process. The only disadvantage is that CFG grammars must be transformed into an iterative form. Through iterations of the rules, it is possible to generate mathematical expressions of desired types.

5.4.1. Grammar Manipulation

As in the parsing process of expressions, the generating one must also be controlled via functions that can be individually defined for each non-terminal of a grammar. Each of these functions is typically implemented as a loop structure having a certain number of iterations. Table 72 shows a typical implementation of such a function, called repeater().

Table 72–An implementation for a generating function called repeater().

```
public Rule repeater(Rule r)
  out ← []
  for i = 1 to r.n
    out.push(r.v)
  return out
```

The iterations of grammar rules make it more customizable to generate expressions associated with a value assigned for each rule. For example, assign some value for x, we can yield different types of expressions, such as first-degree and quadratic equations. A particular type of expressions is represented by a class in which a value is set for the number of iterations of grammar rules. It is possible to define various classes that represent distinct types of expressions. A new grammar with modified rules of the grammar shown in Table 73 is given in Table 65.

Table 73 – An enhanced version of the grammar in Table 65

```
AST ← Start Symbol
N   ← Number of maximum nodes
n   ← 0
While a Non-Terminal exists in AST
   E    ← Select a non-Terminal in AST
   if(n<N)
     sub  ← Expand E using a random rule
   else
     sub  ← Expand E using a random rule without a non-terminal
   sub  ← repeater(sub, E.n)  # Using Table 72
   AST  ← Replace E with sub in AST
In-order traverse the AST and print mathematical expression
```

5.4.2. CFG versus RI-CFG

In this section we introduce comparative examples of CFG an RI-CFG grammars for some expressions. In some cases the enhanced grammar is almost identical to CFG, however, the noticeable difference occurs for complex expressions. Table 74 shows an example of CFG and RI-CFG grammars that produce the sentences of the same language. Note that RI-CFG rules have a controlled number of iterations.

Table 74 – CFG and RI-CFG grammar for the language $a^n b^n$

CFG	RI-CFG
S->aSb S->λ	S->aSb, 1 S->λ, 1

Table 75 shows another comparative example.

Table 75 – CFG and RI-CFG grammars for the language $a^{3n} b^{3n}$

CFG	RI-CFG
S->aaaSbbb S->λ	S->ASB, 1 A->a, 3 B->b, 3 S->λ, 1

As seen in Table 75, the RI-CFG grammar contains additional non-terminals of *A* and *B*. Although, this gives rise to the number of non-terminals, however, it would be flexible and easy to change the grammar to the grammar of another similar language such as $a^n b^{2n}$, etc. Another important point for such grammars is that during their implementations, the number of iterations of a rule can change dynamically. Thus, an extension of the grammar can always be constructed without enhancing similar ones.

As discussed in Section 0, writing a CFG grammar with specific restrictions requires complex structures and various controls over a generated string. However, it is easy to apply such restrictions via the proposed type of grammar. Table 76 shows the grammar of first-degree equations in CFG and RI-CFG, notations.

Table 76 – The grammar of first-degree equations in the CFG and RI-CFG notations

CFG	RI-CFG
S ⟶ E=E E ⟶ E + E | E - E | Number | Var E ⟶ E * F| F * E F ⟶ F + F|F - F | F * F | Number Var ⟶ x Number⟶-?[digit]$^+$[.[digit]$^+$]? digit ⟶ [0 - 9]	S ⟶ TE=TE, 1 E ⟶ [+T | -T], n T ⟶ Number(x)?, 1 Number⟶-?[digit]$^+$[.[digit]$^+$]?,1 digit ⟶ [0 - 9],1

The grammar in Table 76 produces first-degree equations of n terms. As mentioned before, integer n can be entered during the implementation or even at run-time to change the number of terms. Besides, one can select a random value for n from a range of integers n_a through n_b.

With some rule additions, modifications or replacements, we can easily transform the grammar in Table 76 into another. For example, the grammar would start to yield quadratic equations with the addition of a rule "$T \longrightarrow Dx^2$".

5.5. A Methodology for Expression Generation

The importance of generating random mathematical expressions is discussed in previous sections. This section presents a grammar-based methodology which uses expression templates to produce different types of expressions.

5.5.1. RI-CFG-Based Production of Expressions

The methodology manages the expression generating into which a considerable kind of expression templates can easily be embedded.

Figure 63 shows the steps of the proposed methodology.

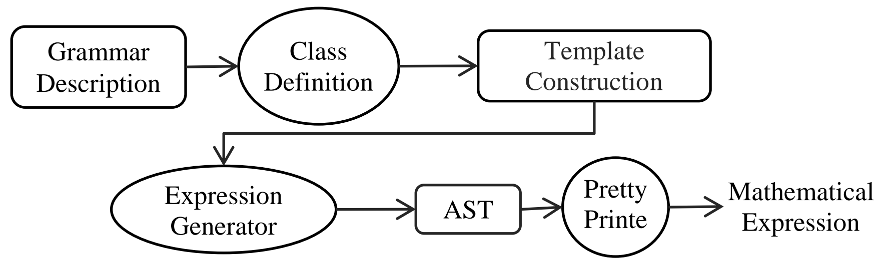


Figure 63 - Components of the proposed methodology

In

Figure 63, the grammar pool contains various *RI-CFG* grammars for templates, for first-degree equations, quadratic equations, polynomials, trigonometric, etc. We have developed a class called *RICFG* which has the core implementations of Rule, Grammar and other classes. This RICFG class is extended by every expression template.

5.5.1.1. Grammar Development

The first step of the methodology is the development of a RI-CFG grammar special to target mathematical expressions. Table 77 shows a typical example of such a grammar.

Table 77 - An example of RI-CFG grammar

```
S ⟶  DT, 1
T ⟶  [+D | -D], @n1
D ⟶  [0 | 1 | 2 ... | 9], @n2
```

In Table 77, $@n_i$ is an attribute of the grammar which controls the number of generated terms. This attribute repeat the number of iterations for the related rules, set by a particular integer at the time of implementing expression templates. The other components of the methodology are also described via the grammar in Table 77 in the following sections.

5.5.1.2. Class Definition

As in the analysis process of expressions, each grammar rule would be represented with a class that can be defined using any object-oriented programming language. We code the rules in the Java language, extending the RI-CFG class. The superclass RI-CFG contains many useful methods such as *hasNonTerminal()*, *SelectRandomRule()*, and *generate()*. Here, we give one of its methods, *generate*(), in Table 78.

Table 78 – A method, *generate(),* of *RICFG* class

```
public Exp generate(){
    ArrayList<Object> list = new ArrayList<>();
    list.add(START);
    while(hasNonTerminal(list)){
      Rule nt = getFirstNonTerminal(list);
      replaceOnce(list, SelectRandomRule(nt).repeat());
    }
    return getAST(list);
  }
}
```

Given the grammar rules in Table 78, the non-terminals T and D are represented by the class NonTerminal. The rules, together with their alternatives, are implemented as the objects of the class Rule. Table 79 demonstrates the class implementations of the rules

Table 79 - Class definitions of templates

```
// S ⟶  DT, 1
// T ⟶  [+D | -D], @n1
// D ⟶  [0|1|2..|9], @n2

public class ExampleGrammar extends RICFG {
  public ExampleGrammar(Range n1, Range n2){
    NonTerminal T = NonTerminal.get("T");
    NonTerminal D = NonTerminal.get("D");

    pool = new ArrayList<>()
    pool.add(new Rule(START, new Object[]{D, T}, 1));
    pool.add(new Rule(T, new Object[]{"+", D}, n1));
    pool.add(new Rule(T, new Object[]{"-", D}, n1));
    pool.add(new Rule(D, "0", n2));
    pool.add(new Rule(D, "1", n2));
    pool.add(new Rule(D, "2", n2));
    ...
    pool.add(new Rule(D, "9", n2));
  }
}
```

In Table 79, the class ExampleGrammar can be supported with additional methods for simplicity and customization.

5.5.1.3. Template Construction

Expression templates are in fact implemented as objects of the related classes of RI-CFGs in which the attributes are set to proper values. It is adequate to manipulate the object attributes with the aim of generating various forms of expressions. A collection of templates can be packaged as a framework to support various forms of mathematical expressions. Table 80 displays some expression templates for the rules defined in Table 79

Table 80 – The definitions of some expression templates

```
// expressions with two digits
ExampleGrammar term1 =
     new ExampleGrammar(new Range(1, 5), new Range(2, 2));

// expressions with 3 to 10 terms
ExampleGrammar term2 =
     new ExampleGrammar (new Range(3, 10), new Range(1, 4));
```

5.5.1.4. Production of Expressions

The method *generate()*listed in Table 78 produces an AST. Working with AST is easy as it can provide useful methods to deal with mathematical expressions. For example, we can define specific methods on ASTs to its content into human-readable strings, MathMl, and LaTex. Table 81 shows the outputs of the templates given in Table 80.

Table 81 - The outputs of the templates in Table 80

```
// output of term1
12+49-30
51-25
73-59+02+11

// output of term2
782+63+4-5120+8+1-576+23
7+163-2+87
15-39
```

5.6. Applications

A class that implements a template serves as a generator for a certain type of expressions represented by that template. An expression template can use a set of other templates. In this section, we will demonstrate some applications of the methodology using polynomial and limits templates. The Java notation is used for code demonstration.

5.6.1. Polynomials

Polynomials are composed of different terms or monomials added or subtracted from each other. Each monomial, as discussed in Section 4.11.3, has a well-known structure. Polynomial expressions can be defined by the RI-CFG grammar given in Table 82.

Table 82 – A RI-CFG grammar for polynomial expressions

```
S ⟶  MT, 1
T ⟶  [+M | -M], @n1
M ⟶ DP, 1
P ⟶ [xR | x], 1
R ⟶ [*x], @n2
D ⟶ [0|1|2..|9], @n2
```

In Table 82 three different attributes are used to control the production process of polynomial expressions where *N* is the number of monomials, *P* is the maximum exponential, and *C* is the range of values for coefficients.

A class named TPolynomial is defined to implement the grammar of polynomials as in Table 83.

Table 83 - Code implementation of Table 82

```
public class TPolynomial extends RICFG {
  public Range count = new Range();
  public Range exponential = new Range();
  public Range coefficient = new Range();
  public TPolynomial(){
    NonTerminal T = NonTerminal.get("T");
    NonTerminal M = NonTerminal.get("M");
    NonTerminal P = NonTerminal.get("P");
    NonTerminal R = NonTerminal.get("R");
    NonTerminal D = NonTerminal.get("D");
    pool = new ArrayList<>()
    pool.add(new Rule(START, new Object[]{M, T}, 1));
    pool.add(new Rule(T, new Object[]{"+", M}, n1));
```

```
    pool.add(new Rule(T, new Object[]{"-", M}, n1));
    pool.add(new Rule(M, new Object[]{D, P}, 1));
    pool.add(new Rule(P, new Object[]{"x", R}, 1));
    pool.add(new Rule(P, "x", 1));
    pool.add(new Rule(R, new Object[]{"*", "x"}, n2));
    pool.add(new Rule(D, null, n2));
  }
}
```

For example, let us create a polynomial template with the attributes listed in Table 84.

Table 84 - A sample template for polynomials

Attribute	Value
Number of monomials (N)	(3, 5)
Exponentials (P)	(2, 6)
Coefficients (C)	(-5, 10)

Table 85 shows a method which produces random polynomial expressions using the attributes in Table 84.

Table 85 – A random polynomial generator using templates

```
TPolynomial poly = new TPolynomial();
poly.count.set(3, 5);
poly.exponential.set(2, 6);
poly.coefficient.set(-5, 10);
Exp ast = poly.generate();
```

The examples of re-usability of template classes using Table 85 are shown below:

Example 1: Generate first-degree and quadratic expressions.

To do so, *poly.exponential* has to be set to *(0, 1)* and *(0, 2)* respectively. The related equations must be constructed based on the structure of each equation. Table 86 shows an example of constructing quadratic equations.

Table 86 - Random quadratic equation using Table 85

```
poly.exponential.set(0, 2);
Exp left  = poly.generate();
Exp right = poly.generate();
Exp quad  = new Equ(left, right);
```

Example 2: Generate dividable polynomials, such as "$P(x)/Q(x)$", where $P(x)$ and $Q(x)$ are polynomials. For $P(x) = a_2x^2+a_1x+a_0$, $Q(x)=k\times b_1x+k\times b_0$ where P(x) is dividable with "b_1x+b_0" and k is an arbitrary integer.

To do so, two polynomials S_1 and S_2 is generated *where* $deg(S_1)=deg(S_2)=1$. Simplifying S_1 and S_2 will be resulted in $S_1=c_1x+c_0$ and $S_2=b_1x+b_0$. Assuming "$P(x) = S_1\times S_2$" and "$Q = k\times S_2$" where k=[-7, 7]. Table 87 shows an implementation of this example.

Table 87 - An example of dividable polynomials using templates

```
poly.exponential.set(0, 1);
Exp S1 = poly.generate();
Exp S2 = poly.generate();
Exp K  = new Rand(-7, 7).get();
Exp P  = new Times(S1, S2).Simplify();
Exp Q  = new Times(k, S1).Simplify();
Exp div = new Div(P, Q).Simplify();
```

Example 3: Generate expressions which "$f(x) = a_3k^3x^3+a_2k^2x^2+a_1kx+a_0$".

To do so, *F(x)* and *G(x)* is generated, where $F(x) = a_3x^3+a_2x^2+a_1x+a_0$, $G(x) = kx$, and k is an arbitrary integer. Simplifying $F(x)$ and then creating $(FoG)(x)$ will produce expressions in required pattern. Table 88 shows an implementation of this example, supposing that k = [-7, 7].

Table 88 - An example of polynomials using templates

```
Poly.exponential.set(0, 3);
Exp F = poly.generate().Simplify();
Exp K  = new Rand(-7, 7).get();
Exp G = new Times(K, Var.X());
Exp div = F.eval(G).Simplify();
```

5.6.1. Indeterminate Limits

Expressions such as "$\lim_{x\to v} f(x)$" can be produced using Exp class which is discussed in Section 0, where lim is an extended class of Exp and accepts v and $f(x)$ as parameters. $F(x)$ is generated using other templates associated with what kind of expression one may need. Table 89 shows an implementation of limit where $f(x)$ is a polynomial expression.

Table 89 – An example of the limit using Table 88

```
Exp F = poly.generate().Simplify();
Exp v  = new Rand(-6, 8).get();
Exp lim = new Limit(v, F);
```

The common indeterminate form are denoted 0/0, ∞/∞, 0*∞, 0^0 and ∞^0. Of these forms, we will handle the ratio of two functions which both tend to zero in limit, referred to as "the form 0/0". To generate the expressions of this form, the following steps can be used for the limit of $f(x)$ as x approaches v.

1. Define an expression "(x - v)" where v is a number.
2. Generate $P(x)$ and $Q(x)$, where P and Q are two polynomials
3. Multiply "(x - v)" into P and Q.
4. Replace the resulting P and Q by their simplifications.
5. Set $f(x)$ to $P(x)$ over $Q(x)$

Table 90 shows an implementation of these steps.

Table 90 - An example of an indeterminate limit expression

```
Exp T = new Minus(Var.X(), v);
Exp P = poly.generate().Simplify();
Exp Q = poly.generate().Simplify();
P  = new Times(T, P).Simplify();
Q  = new Times(T, Q).Simplify();
Exp fx = new Div(P, Q).Simplify();
```

6. CONCLUTION

In this thesis, we propose two grammar-based methodologies for the automatic production and step-by-step solving of mathematical. The methodologies are basically designed to handle algebraic expressions that can be evaluated or simplified into a more concise form. From this perspective, on the contrary of numerical computation methods, we have aimed to produce the exact values of mathematical expressions, using symbolic computation approaches. There are many algebra-related topics such as simplification, factorization, distribution and substitution that can easily be covered by the solution methodology. Our methodologies currently support univariate operations on these algebra topics, but can also be adapted to operate on multivariate expressions.

In order to parse input string of an algebraic expression and understand an algebraic identity, computer systems require some special algorithms that verify, separate and identify these expressions. In this way, we present new algorithms based on context-free grammars that can achieve the operations such as expression separation and identification.

The form of expressions is represented via LL(1) grammars that exhibit an unambiguous aspect. Using the JavaCC tool, we develop LL(1) parsers which always follow a unique derivation for every kind of mathematical expressions. The implementation of such parsers is relatively easy as it involves the mapping of the grammar rules into corresponding methods. We illustrate the use of the tool through the implementation of the different parts of the solution methodology.

We model an input expression by using Abstract Syntax Tree (AST) which has a hierarchical structure. The tree represents the precedence and associativity of operators, and thus establishes the semantic-based relations among its nodes. It is quite possible to make the different interpretations of an AST, however, we use it to evaluate simplify and optimize the expressions. The well-known applications such as LISP and Maxima use a similar tree structure for manipulation of expressions.

Using AST, the step-by-step solutions for mathematical expression are proposed. The simplification ways of equations and similar expressions are discussed. Simplifying a node in an AST simply depends on the type of that node. Different occurrences of an operator or function node and its children confront different simplifications or

transformations. Each transformation resolves a part of the complexity that exists in a node.

AST supports the execution of the evaluating operations in a recursive manner. In this way, the simplification process goes down in the tree until it encounters the leaves that contain numbers or variables and terminates when transformation can further be applied to the tree. However, there are some special cases where we need to define control flags, for example, to decide how to use algebraic identities (i.e., product and factoring formulas). This is to prevent those formulas from triggering several times, ensuring that no infinite loops occur.

A different type of mathematical expressions requires a different style of simplification. We introduce some basic and sophisticated transformations on AST nodes, but they can be extended to construct new ones operating on other states of nodes. The examples of basic transformations are commutative and power and the examples of sophisticated transformations are linearization of nested operators; fraction, power and radical simplifications; and radical removal from fractions.

In term of the step-by-step solution, these transformations of the simplification process are considered as the steps of solving the problem. The result of each transformation is a new AST; therefore this operation advances the solution one step towards the final one.

Various documents or reports can be produced to print the applied transformations and the resulting expression. We prefer to use a document formatting system in which content of AST is printed. The differences between two consecutive prints of the output show the progress of solving the given input through a symbolic computation system.

The evaluation of AST is quite simple. The tree contains variables and numbers, therefore by initializing variables such as x, a, and b into some values, the numerical result can be calculated.

We propose a methodology for producing mathematical expressions which can be used to generate questions for exams and practicing exercises to improve students' skill in mathematics. Different forms of mathematical problems need have different sets of grammar rules and symbols. First degree, and quadratic equations, polynomials, and other mathematical expressions are represented by a formal language which can be modeled by context free grammars (CFG). In general, the expressions randomly generated in this language are expected to correspond randomly with mathematical questions.

Unfortunately, CFGs cannot control the generation of expression which meet various requirements in math problems.

The generating methodology imposes some restrictions on the expressions, including the number of terms, the degree of variables and the range of coefficients. To control these restrictions, we modify the rule structure of CFG grammars, calling it RI-CFG. Unlike CFG grammars, RI-CFG ones have an interactive structure where the grammar rules iterate in a way directed by the related restrictions during the generation of expressions.

The RI-CFG rules, implemented as classes in the Java programming language, serve as expression templates, inheriting from the class of an expression template, we can develop new templates. The behavior of expression templates can be modified by setting the class attributes that are obtained from RI-CFG rules. Each attribute provides a different way of producing random mathematical expressions.

For example, a template of polynomial expressions can be used in various ways as a well-contained one. By limiting the maximum degree of variable x to 1 or 2, first degree or quadratic equations can be generated respectively. A similar process can be also repeated done for other mathematical expressions to produce other RI-CFG grammars and related templates.

By combining the two methodologies into a package, a framework can be created. In this framework, powerful methods can be implemented to provide a leading environment for future developments. Besides, using the basic features of framework, one can produce random math questions and then get their step-by-step solutions. The applications of the framework are beyond the scope of this thesis. However, for the sake of a presentation and competition, we have developed an application in Java.

Several experiments are conducted to compare the benefits and drawbacks of the proposed methodologies for solving and producing mathematical expressions. We have used Matlab Symbolic Computation Toolbox and a real person as our test competitors. The results are listed in Table 91.

Table 91 - Symbolic calculation of some math questions

Expression	Matlab	A real person	Our System
x+3x+2	4x+2	4x+2	4x+2
2x(x^2+2x+1)	2x(x+1)^2	2x^3+4x^2+2x	2x^3+4x^2+2x
(2x-1)(3x+1)	(2x-1)(3x+1)	6x^2-x-1	6x^2-x-1
$((2^a)^b)^3$	$(2^a)^{3b}$	2^{3ab}	2^{3ab}
$2^{(a^{(b^3)})}$	$2^{(a^{(b^3)})}$	$2^{(a^{(b^3)})}$	$2^{(a^{(b^3)})}$
$\frac{x^2-2x+1}{x-1}$	x-1	x-1	x-1

For further analysis of the solution methodology, we have conducted some experiments for time complexity. The results are listed in Table 92.

Table 92 - Time and space complexity

Expression	Number of terms	Step-by-step solution		Fast solution	
		Recalls	Average	Recalls	Average
x+3x+2	3	5	1.67	3	1
2x(x^2+2x+1)	4	18	4.5	13	3.25
(2x-1)(3x+1)	4	16	4	11	2.75
$((2^a)^b)^3$	3	5	1.67	3	1
$2^{(a^{(b^3)})}$	3	3	1	3	1
$\frac{x^2+2x-1}{x-1}$	5	9	1.8	4	0.8

Also to investigate the system performance, produced a certain number of mathematical expressions by the system. Generated expressions evaluated by the math teachers. These statements, placed in four categories in accordance with the demands of the individuals. The results are shown in Table 93.

In the methodology proposed for solving math problems, a new recursive model based on AST is used. With object-oriented features in programming languages, it is easy to implement various transformations suggested for simplifications. The step-by-step solutions can be reported using each applied transformation.

In the methodology proposed for producing math problems, a new type of grammar is introduced. The rules are modified to carry a new attribute which is used to control the type of produced expressions. The classes defined for these grammars allow their sub classes to modify the attributes to generate different types of expressions.

Table 93 – Evaluation of produced expressions

Product Type	Total	Excellent	Good	Average	Failing
Polynomial	100	65	25	10	0
First-Degree Equation	100	80	18	2	0
Quadratic Equation	100	73	20	7	0
Indeterminate Limit	100	57	29	11	3

We are planning to produce applications for computers and smart-phones to ensure that schools and students can find solutions for their mathematical problems and also enjoy an unlimited number of various math questions to develop their math skills. Also additional methods for improving the performance of implementations are being developed which will support used in future applications and publications.

7. REFERENCE

1. Dunham, P.H. and Dick, T.P., Connecting Research to Teaching: Research on Graphing Calculators, Mathematics teacher, 87, 6 (1994) 440-45.
2. https://www.coursera.org Coursera. 10 May 2013.
3. https://edventure.ntu.edu.sg. Edventure. 10 May 2013.
4. Gulwani, S., Korthikanti, V.A. and Tiwari, A., Synthesizing geometry constructions, ACM SIGPLAN Notices, 46 (2011) 50–61.
5. Singh, R., Gulwani, S. and Rajamani, S., Automatically generating algebra problems, AAAI, (2012).
6. Rasila, A., Harjula, M. and Zenger, K., Automatic assessment of mathematics exercises: Experiences and future prospects, ReflekTori Symposium of Engineering Education, (2007) 70–80.
7. Rasila, A., Havola, L., Majander, H. and Malinen, J., Automatic assessment in engineering mathematics: evaluation of the impact, ReflekTori Symposium of Engineering Education, (2010) 37–45.
8. Cohen, J.S., Computer algebra and symbolic computation: Mathematical methods, Denver Universities Press, A K Peters, Natick, MA, (2003).
9. Von Zur Gathen, J. and Gerhard, J., Modern computer algebra. Cambridge University Press, (2003).
10. http://www.wolfram.com/mathematica/, Mathematica, 1% July 2011.
11. http://www.maplesoft.com/, Mapple Version 15, 15 July 2011.
12. Buchberger, B. and Loos, R., Algebraic simplification. Computer Algebra-Symbolic and Algebraic Computation, (1982) 11–43.
13. Moses, J., Algebraic simplification a guide for the perplexed, In Proceedings of the second ACM symposium on Symbolic and algebraic manipulation, (1971) 282–304.
14. Shatnawi, M. and Youssef, A., Equivalence detection using parse-tree normalization for math search, 2nd International Conference on Digital Information Management, ICDIM, 2 (2007) 643–648.
15. Youssef, A. and Shatnawi, M., Math searches with equivalence detection using parse-tree normalization, 4th International Conference on Computer Science and Information Technology, (2006).
16. Apostolico, A. and Galil, Z., Pattern matching algorithms, Oxford University Press, USA, (1997).
17. Zhang, K. and Shasha, D., Simple fast algorithms for the editing distance between trees and related problems, SIAM journal on computing, 18,6 (1989) 1245–1262.
18. Demana, F., Calculators in mathematics teaching and learning, Learning mathematics for a new century, (2000) 51.
19. Runde, D.C., The Effect of Using the TI-92 on Basic College Algebra Students, Ability To Solve Word Problems, (1997).
20. Palmiter, J.R., Effects of computer algebra systems on concept and skill acquisition in calculus, Journal for research in mathematics education, (1991) 151-156.
21. Judson, P.T., Effects of modified sequencing of skills and applications in introductory calculus, University of Texas at Austin, (1988).

22. Judson, P.T., Elementary business calculus with computer algebra, The Journal of Mathematical Behavior, (1990).
23. Drijvers, P., Assessment and new technologies: Different policies in different countries, International Journal of Computer Algebra in Mathematics Education, 5 (1998) 81-94.
24. Herget, W., Heugl, H.K., Kutzler, B., and Lehmann, E., Indispensable manual calculation skills in a CAS environment, Exam Questions & Basic Skills in Technology-Supported Mathematics Teaching, bk teachware, Hagenberg, (2000).
25. Brown, R., Computer algebra systems in junior high school, in Proceedings from the 3rd international Derive and TI-92 Conference, (1998).
26. Heid, M.K., The technological revolution and the reform of school mathematics, American Journal of Education, (1997) 5-61.
27. Day, R.P., Algebra and Technology, Journal of Computers in Mathematics and Science Teaching, 12, 1 (1993) 29-36.
28. Bennett, G., Calculus for general education in a computer classroom, International DERIVE Journal, 2 (1995) 3-11.
29. Tall, D., Functions and calculus, in International handbook of mathematics education, Springer, (1996) 289-325.
30. Heid, M.K., Resequencing skills and concepts in applied calculus using the computer as a tool. Journal for research in mathematics education, (1988) 3-25.
31. Repo, S., Understanding and reflective abstraction: Learning the concept of derivative in a computer environment, International DERIVE Journal, 1, 1 (1994) 97-113.
32. Small, D.B. and Hosack, J.M., Computer algebra system tools for reforming calculus instruction, Toward a lean and lively calculus, (1986) 143-155.
33. Hearst, M. A., The debate on automated essay grading, IEEE Intelligent Systems and their Applications, 15, 5 (2000) 22–37.
34. Mohler, M., Bunescu, M. A. and Mihalcea, R., Learning to grade short answer questions using semantic similarity measures and dependency graph alignments, In Proceedings of Association for ComputationalLinguistics, (2011) 752–762.
35. Farnsworth, C.C., Using computer simulations in problem-based learning, Document Resume, (1994) 145.
36. Kashy, E., Sherrill B. M., Tasi, Y., Thaler, D., Weinshank D., Engelmann E. and Morrissey, D. J. , CAPA-An integrated computer-assisted personalized assignment system, American Journal of Physics, 61, 12 (1993) 1124-1130.
37. Gutierrez-Santos, S., Geraniou, E., Pearce-Lazard, D. and Poulovassilis, A., Design of teacher assistance tools in an exploratory learning environment for algebraic generalization, IEEE Transactions on Learning Technologies, 5, 4 (2012) 366-376.
38. Dragon, T., Mavrikis, M., McLaren, B.M., Harrer, A., Kynigos, C., Wegerif, R. and Yang Yang, Metafora: A web-based platform for learning to learn together in science and mathematics, IEEE Transactions on Learning Technologies, 6, 3 (2013) 197-207.
39. Barker, D.S., CHARLIE: A computer-managed homework, assignment and response, learning and instruction environment, in Frontiers in Education Conference, 27th Annual Conference. Teaching and Learning in an Era of Change. Proceedings. (1997), IEEE.
40. Bridgeman, S., Goodrich, M.T., Kobourov, S.G. and Tamassia, R., PILOT: An interactive tool for learning and grading, in ACM SIGCSE Bulletin, (2000).
41. Baldwin, D., Three years' experience with gateway labs. ACM SIGCSE Bulletin, 28 (1996) 6-7.
42. Brooker, R. A., The solution of algebraic equations on the EDSAC, Cambridge Phil. Soc, Proc, 48 (1952) 255-270.

43. Hans J., Zur Iterativen Auflösung Algebraischer Gleichungen, Z. Angew. Math Physik, 5 (1954) 260-263.
44. Butler, G. and Cannon, J. , The design of cayley - a language for modern algebra, Springer, (1990).
45. Schonert, M., GAP 3.4 Manual (Groups, algorithms, and Programming), RWTH Aachen, (1994).
46. Capani, A. and Niesi, G., CoCoA User's Manual (v. 3.0 b). Dept. of Mathematics, University of Genova, (1996).
47. Van Leeuwen, M., a software package for Lie group computations, Euromath Bull, 1, 2 (1994) 83-94.
48. Kurz, T.L., Middleton, J.A. and Yanik, H.B., A Taxonomy of Technological Tools for Mathematics Instruction, Contemporary Issues in Technology and Teacher Education, 5**,** 2 (2005) 123-137.
49. Handal, B. and Herrington, A., Mathematics teachers' beliefs and curriculum reform, Mathematics Education Research Journal, 15, 1 (2003) 59-69.
50. McRoy, S.W., Channarukul, S. and Ali, S.S., YAG: A template-based generator for real-time systems, Proceedings of the first international conference on Natural language generation, 14. (2000). Association for Computational Linguistics.
51. Theune, M., Klabbers, E., Odijk, J., Pijper , J. R. and Krahmer , E., From data to speech: a general approach, Natural Language Engineering, **7**, 01 (2001) 47-86.
52. White, M. and Caldwell, T., A practical, extensible framework for dynamic text generation, in Proceedings of the Ninth International Workshop on Natural Language Generation (INLG). (1998).
53. Stenzhorn, H., XtraGen: a natural language generation system using XML-and Java-technologies, in Proceedings of the 2nd workshop on NLP and XML, 17 (2002).
54. Schellekens, D., Preneel, B. and Verbauwhede, I., FPGA vendor agnostic true random number generator, in Field Programmable Logic and Applications, FPL'06, International Conference on. (2006). IEEE.
55. Tkacik, T.E., A hardware random number generator, in Cryptographic Hardware and Embedded Systems-CHES, Springer. (2002) 450-453.
56. Jun, B. and Kocher, P., The Intel random number generator, Cryptography Research Inc. white paper, 1999.
57. Cohn, C.E., RANDOM NUMBER GENERATOR, Google Patents, 1972.
58. Jakimoski, G. and Kocarev, L., Chaos and cryptography: block encryption ciphers based on chaotic maps, IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, 48, 2 (2001) 163-169.
59. Kocarev, L., Chaos-based cryptography: a brief overview, Circuits and Systems Magazine, IEEE, 1, 3 (2001) 6-21.
60. Langkilde, I. and Knight, K., Generation that exploits corpus-based statistical knowledge, in Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, 1. (1998).
61. Coates, R.F., Janacek, G.J. and Lever, K.V., Monte Carlo simulation and random number generation, IEEE Journal on Selected Areas in Communications, 6, 1 (1988) 58-66.
62. Ferrenberg, A.M., Landau, D. and Wong, Y.J., Monte Carlo simulations: Hidden errors from ‘‘good’’random number generators, Physical Review Letters, 69, 23 (1992) 3382.
63. Schaeffer, L., Application of random regression models in animal breeding, Livestock Production Science, 86, 1 (2004) 35-45.
64. Stojanovski, T. and Kocarev, L., Chaos-based random number generators-part I: analysis [cryptography], IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, 48, 3 (2001) 281-288.

65. Michael, E., Laszlo, H., Raymond, K., Martin, R. and Hao, Z., Design and implementation of a true random number generator based on digital circuit artifacts, in Cryptographic Hardware and Embedded Systems-CHES, Springer, (2003) 152-165.
66. Gilbert, E.N., Random graphs. The Annals of Mathematical Statistics, (1959) 1141-1144.
67. Van Deemter, K., Krahmer, E. and Theune, M., Real versus template-based natural language generation: A false opposition? Computational Linguistics, 31, 1 (2005) 15-24.
68. Becker, T., Practical, template-based natural language generation with TAG, in Proceedings of TAG. (2002).
69. Kumar, A.N., Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors, Technology, Instruction, Cognition and Learning (TICL) Journal, 4, 1 (2006).
70. Yoshikawa, T., Shimura, K. and Ozawa, T., Random program generator for Java JIT compiler test system, in Quality Software, Proceedings. Third International Conference on, (2003).
71. Melis, E., Eric, A., Jochen, B., Adrian, F., Erica, M., George, G., Paul, L., Martin P. and Carsten U., ActiveMath: A generic and adaptive web-based learning environment, International Journal of Artificial Intelligence in Education (IJAIED), 12 (2001) 385-407.
72. Hoffman, D., Wang, H.Y., Chang, M., Ly-Gagnon, D., Sobotkiewicz, D. and Strooper, P., Two case studies in grammar-based test generation, Journal of Systems and Software, 83, 12 (2010) 2369-2378.
73. Klai, S., Kolokolnikov, T. and Van den Bergh, N., Using Maple and the web to grade mathematics tests, in Advanced Learning Technologies, IWALT, (2000).
74. Almeida, J.J., Araujo, I., Brito, I., Carvalho, N., Machado, G.J., Pereira, R.M.S. and Smirnov, G., Math exercise generation and smart assessment, in Information Systems and Technologies (CISTI), (2013).
75. Tomás, A.P. and Leal, J.P., A CLP-based tool for computer aided generation and solving of maths exercises, in Practical Aspects of Declarative Languages, Springer, (2003) 223-240.
76. *http://en.wikipedia.org/wiki/Algebraic_expression*, Algebraic expression. N.d. In Wikipedia, The Free Encyclopedia. Retrieved January 23, 2015,
77. Pulman, S. G. , Basic Parsing Techniques: an introductory survey, (1991).
78. Jacobs, C. J. and Grune, D., Parsing Techniques-A Practical Guide, (1990).
79. Sippu, S., and Soisalon-Soininen, E., Parsing Theory: LR (k) and LL (k) Parsing, 20 (1990).
80. Johnson, S.C., Yacc: Yet another compiler-compiler, 32 (1975).
81. Donnelly, C. and Stallman, R., Bison the YACC-compatible Parser Generator, (2004).
82. Earley, J., An efficient context-free parsing algorithm, Communications of the ACM, 13, 2 (1970) 94-102.
83. Gill, A. and Marlow, S., Happy: the parser generator for Haskell. University of Glasgow, (1995).
84. Graver, J.O., The evolution of an object-oriented compiler framework, Software: Practice and Experience, 22, 7 (1992) 519-535.
85. Hudson, S., JAVACUP: LALR parser generator for Java, (1999).
86. Kodaganallur, V., Incorporating language processing into java applications: A JavaCC tutorial, Software IEEE, 21, 4 (2004) 70-77.
87. Viswanadha, S. and Sankar, S., Java compiler compiler (JavaCC)-The java parser generator,(2009).
88. Lundberg, J., Getting started with JavaCC, Vaxjo University, (2006).
89. Levine, J.R., Mason, T. and Brown, D., Lex & yacc, O'Reilly Media Inc, (1992).

90. Ryan, C., O'Neill, M. and Collins, J., Grammatical evolution: solving trigonometric identities. Proceedings of Mendel, (1998) Citeseer.

91. Jones, J., Abstract syntax tree implementation idioms, in Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003), (2003).

92. Gavin, H. P., A brief LATEX tutorial, (2002).

93. Ausbrooks, R., Buswell, S., Carlisle, D., Dalmas, S., Devitt, S., Diaz, A. and Watt, S., Mathematical Markup Language (MathML) version 2.0, W3C recommendation World Wide Web Consortium, (2003).

94. Brzozowski, J.A., Derivatives of regular expressions, Journal of the ACM (JACM), 11, 4 (1964) 481-494.

95. Newmeyer, F.J., Grammar is grammar and usage is usage, Language, (2003) 682-707.

96. Hilton, P. and Pedersen, J., Catalan numbers their generalization and their uses, The Mathematical Intelligencer, 13, 2 (1991) 64-75.

8. APPENDIIX

Appendix A – Grammar and JavaCC codes

- *LL(1) Grammar*

```
G={Σ, T, V, P, S}
V={start, equ, func, expr, element, term, unary, power, func, number,
digit}⊆ Σ
T={x, Sin, Cos, Tan, Log, Exp, Sqrt, ( , ), +, -, *, /, ^}⊆Σ
Σ=T ∪V
S={start}
Productions
<start> ⟶ <equ> ("," <equ>)?
<equ>   ⟶ <expr> ( "=" <expr>)? | <func> ( "=" <expr>)?
<func> ⟶ "f(" <expr> ")=" <expr>
<expr> ⟶ <term> [("+"|"-") <term> ]*
<term> ⟶ <unary> [("*"|"/") <unary>]*
<unary> ⟶ ("+"|"-") ? <power>
<power> ⟶ <element> ("^"<power>)?
<element> ⟶ <func> (<expr>) | <number> | "x"
<func> ⟶ "Sin " | "Cos" | "Tan" | "Log" | "Exp" | "Sqrt"
<number> ⟶ "-"? <digit> + ("."<digit>+)?
<digit> ⟶ [" 0"-"9"]
```

- *JavaCC* Code

```
options {
    STATIC = false; // make parser methods static
    LOOKAHEAD = 1;
}
PARSER_BEGIN(EvalParse)
public class EvalParse {}
PARSER_END(EvalParse)
SKIP : { " " | "\t" | "\r" }
TOKEN : { < NUMBER : (["0"-"9"])+ > }
TOKEN : { < EOL : "\n" > }
TOKEN : { /* OPERATORS */
    < PLUS: "+" > | < MINUS: "-" > | < TIMES: "*" > | < DIVIDE: "/" >
  | < POWER: "^" >| < SQRT: "sqrt" >
  | < SIN : "sin" > | < COS : "cos" > | < TAN : "tan" > | < COT : "cot" >
  | < COM: "," >| < EQU: "=" >
  | < LPR: "(" > | < RPR: ")" > | < PI: "pi" >
  | < X: "x" > | < Y: "y" > | < A: "a" >| < B: "b" > | < C: "c" >
  | < D: "d" > | < Q: "q" > | < M: "m" >| < N: "n" > | < F: "f" >
  | < G: "g" > | < H: "h" >
}
Comma parse() : { Equ e1,e2=null;}
{
  e1=equ()  (< COM > e2= equ())?  (<EOF> | <EOL>)  { return new
Comma(e1,e2);}
```

```
}
Func func() : { Exp a,b;}
{
        < F >< LPR > a=expr() <RPR> <EQU> b=expr()
                  { return new Func("f",a,b);   }
      | < G >< LPR > a=expr() <RPR> <EQU> b=expr()
                  { return new Func("g",a,b);   }
      | < H >< LPR > a=expr() <RPR> <EQU> b=expr()
                  { return new Func("h",a,b);   }
}
//*****************************************************
Equ equ(): { Exp a,b=null; Func c; }
{
        a = expr() (< EQU > b= expr())?  { return new Equ(a,b); }
      | c= func()  (< EQU > b= expr())?  { return new Equ(c,b); }
}
//*****************************************************
Exp expr() : { Exp a, b;}
{
  a = term() (
      <PLUS>  b = term()    { a = new Plus(a, b); }
    | <MINUS> b = term()    { a = new Minus(a, b); }
  )*
  { return a; }
}
//****************************************************
Exp term() : { Exp a, b;}
{
  a = unary() (
      <TIMES>  b = unary()  { a = new Times(a, b); }
    | <DIVIDE> b = unary()  { a = new Divide(a, b); }
  )*
  { return a; }
}
//****************************************************
Exp unary() : { Exp a; }
{
    <PLUS>  a = power()     { return a; }
  | <MINUS> a = power()     { return (new Times(new Num(-1,1), a)); }
  | a = power()             { return a; }
}
//****************************************************
Exp power() : { Exp a, b;}
{
  a = element() (
      <POWER> b = power()   { a = new Power(a, b); }
  )?
  { return a; }
}
//****************************************************
Exp element() : { Token t; Exp a, b;}
{
   t = <NUMBER>(
      <X>   {
    return new Times(new Num(Integer.parseInt(t.image),1), new Var("x"));
}
  |   <Y>   {
    return new Times(new Num(Integer.parseInt(t.image),1), new Var("y"));
}
  |  <A>   {
```

```
    return new Times(new Num(Integer.parseInt(t.image),1), new Var("a"));
}
  | <B>    {
   return new Times(new Num(Integer.parseInt(t.image),1), new Var("b"));}
  | <Q>    {
   return new Times(new Num(Integer.parseInt(t.image),1), new Var("q"));}
  | <C>    {
   return new Times(new Num(Integer.parseInt(t.image),1), new Var("c"));}
  | <D>   {
   return new Times(new Num(Integer.parseInt(t.image),1), new Var("d"));}
  | <M>   {
   return new Times(new Num(Integer.parseInt(t.image),1), new Var("m"));}
  | <N>   {
   return new Times(new Num(Integer.parseInt(t.image),1), new Var("n"));}
  | <PI>  {
   return new Times(new Num(Integer.parseInt(t.image),1), new Var("pi"));}
    ) ?   { return new Num(Integer.parseInt(t.image),1); }
  |<X> ( t = <NUMBER>
   {return new Times(new Num(Integer.parseInt(t.image),1),new Var("x"));})?
   { return new Var("x");}
  |<Y> ( t = <NUMBER>
   {return new Times(new Num(Integer.parseInt(t.image),1),new Var("y"));})?
   { return new Var("y");}
  |<A> ( t = <NUMBER>
   {return new Times(new Num(Integer.parseInt(t.image),1),new Var("a"));})?
   { return new Var("a");}
  |<B> ( t = <NUMBER>
   {return new Times(new Num(Integer.parseInt(t.image),1),new Var("b"));})?
   { return new Var("b");}
  |<C> ( t = <NUMBER>
   {return new Times(new Num(Integer.parseInt(t.image),1),new Var("c"));})?
   { return new Var("c");}
  |<D> ( t = <NUMBER>
   {return new Times(new Num(Integer.parseInt(t.image),1),new Var("d"));})?
   { return new Var("d");}
  |<Q> ( t = <NUMBER>
   {return new Times(new Num(Integer.parseInt(t.image),1),new Var("q"));})?
   { return new Var("q");}
  |<M> ( t = <NUMBER>
   {return new Times(new Num(Integer.parseInt(t.image),1),new Var("m"));})?
   { return new Var("m");}
  |<N> ( t = <NUMBER>
   {return new Times(new Num(Integer.parseInt(t.image),1),new Var("n"));})?
   { return new Var("n");}
  |<PI> ( t = <NUMBER>
   {return new Times(new Num(Integer.parseInt(t.image),1),new Var("pi"));})?
   { return new Var("pi");}
  | < SIN > <LPR> a = expr() <RPR>           { return new Sin(a); }
  | < COS > <LPR> a = expr() <RPR>           { return new Cos(a); }
  | < TAN > <LPR> a = expr() <RPR>           { return new Tan(a); }
  | < COT > <LPR> a = expr() <RPR>           { return new Cot(a); }
  | <LPR> a = expr() <RPR>           { return a; }
  | <SQRT> <LPR> a = expr()
      {b= new Num(2,1);} (< COM > b=expr())?<RPR> { return (new Sqt(a,b));}
}
```

Appendix B – *Simplify* and *fog* methods in AST

```
package SymbolicApp;

import java.util.ArrayList;

//****************************************************************
class Equ {

    Exp exp1 = null, exp2;
    public Func equ1 = null;

    public Equ(Func e1, Exp e2) {
        equ1 = e1;
        exp2 = e2;
    }

    public Equ(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }
    …
}
//****************************************************************
class Func {

    Exp exp1, exp2;
    public String f;

    public Func(String e, Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
        f = e;
    }
    …
}
//****************************************************************

class Comma {

    Equ exp1, exp2;

    public Comma(Equ e1, Equ e2) {
        exp1 = e1;
        exp2 = e2;
    }
    …
}
//*****************************************************************

abstract class Exp {
    …
    public abstract Exp Simplify (String str);
    public abstract Exp fog(Exp g);
}
```

```
class Plus extends Exp {

    Exp exp1, exp2, tmp;

    public Plus(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }

    public Exp fog(Exp g) {

        return new Plus(exp1.fog(g), exp2.fog(g));
    }

    public Exp Simplify (String str) {
        ArrayList<Exp> rightPlus = new ArrayList<Exp>();
        ArrayList<Exp> leftPlus = new ArrayList<Exp>();
        ArrayList<Exp> arrPlus = new ArrayList<Exp>();

        // special cases
        if (exp1.eval().equals("0")) {
            GeneralFunc.chkChng = 1;
            return exp2;
        }
        if (exp2.eval().equals("0")) {
            GeneralFunc.chkChng = 1;
            return exp1;
        }
        if (exp1 instanceof Plus || exp1 instanceof Minus) {
            rightPlus = exp1.OpRepeate(str);
        } else {
            rightPlus.add(exp1);
        }
        if (exp2 instanceof Plus || exp2 instanceof Minus) {
            leftPlus = exp2.OpRepeate(str);
        } else {
            leftPlus.add(exp2);
        }
        arrPlus = GeneralFunc.addTwoList(rightPlus, leftPlus);

        // Return Simplify of List
        return GeneralFunc.SimplifyPlusList(arrPlus, str);
    }
    …
}

class Minus extends Exp {

    Exp exp1, exp2;
    Data val2;
    Exp tmp;

    public Minus(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }


    public Exp fog(Exp g) {
```

```
        return new Minus(exp1.fog(g), exp2.fog(g));
    }

    public Exp Simplify(String str) {
        ArrayList<Exp> rightPlus = new ArrayList<Exp>();
        ArrayList<Exp> leftPlus = new ArrayList<Exp>();
        ArrayList<Exp> arrMinus = new ArrayList<Exp>();

        // special cases
        if (exp2.eval().equals("0")) {
            GeneralFunc.chkChng = 1;
            return exp1;
        }
        if (exp1.eval().equals("0")) {
            GeneralFunc.chkChng = 1;
            return new Times(new Num(-1, 1), exp2);
        }
        if (exp1 instanceof Plus || exp1 instanceof Minus) {
            rightPlus = exp1.OpRepeate(str);
        } else {
            rightPlus.add(exp1);
        }
        if (exp2 instanceof Plus || exp2 instanceof Minus) {
            leftPlus = exp2.OpRepeate(str);
        } else {
            leftPlus.add(exp2);
        }
        leftPlus = GeneralFunc.complementList(leftPlus, str);
        arrMinus = GeneralFunc.addTwoList(rightPlus, leftPlus);

        // Simple list Separate
        if (GeneralFunc.chkChng != 0) {
            return GeneralFunc.expPlusArrayList(arrMinus, str);
        }
        // Return Simplify of List
        return GeneralFunc.SimplifyPlusList(arrMinus, str);
    }
    …
}

class Times extends Exp {

    Exp exp1, exp2, tmp;
    Data val2;

    public Times(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }

    public Exp fog(Exp g) {

        return new Times(exp1.fog(g), exp2.fog(g));
    }

    public Exp Simplify(String str) {
        ArrayList<Exp> rightTimes = new ArrayList<Exp>();
        ArrayList<Exp> leftTimes = new ArrayList<Exp>();
        ArrayList<Exp> TimesArrDown = new ArrayList<Exp>();
        ArrayList<Exp> TimesArrUp = new ArrayList<Exp>();
```

```
        Exp chkTmp;
        // special cases
        if (exp1.eval().equals("0")) {
            GeneralFunc.chkChng = 1;
            return new Num(0, 1);
        }
        if (exp2.eval().equals("0")) {
            GeneralFunc.chkChng = 1;
            return new Num(0, 1);
        }
        if (exp1.eval().equals("1")) {
            GeneralFunc.chkChng = 1;
            return exp2;
        }
        if (exp2.eval().equals("1")) {
            GeneralFunc.chkChng = 1;
            return exp1;
        }
        if (exp1 instanceof Sqt && exp2 instanceof Sqt && ((Sqt)
exp1).exp2.eval().equals(((Sqt) exp2).exp2.eval())) {
            GeneralFunc.chkChng = 1;
            return new Sqt(new Times(((Sqt) exp1).exp1, ((Sqt)
exp2).exp1), ((Sqt) exp1).exp2);
        }

        // Create Up (and Down) List(s)
        if (exp1 instanceof Times || exp1 instanceof Divide) {
            rightTimes = exp1.OpRepeate(str);
        } else {
            rightTimes.add(exp1);
        }
        if (exp2 instanceof Times || exp2 instanceof Divide) {
            leftTimes = exp2.OpRepeate(str);
        } else {
            leftTimes.add(exp2);
        }
        Exp re = rightTimes.get(rightTimes.size() - 1);
        Exp le = leftTimes.get(leftTimes.size() - 1);
        if (rightTimes.size() > 0 && re instanceof Divide &&
            ((Divide) re).exp1.eval().equals("1")) {
            TimesArrDown.add(((Divide) re).exp2);
            rightTimes.remove(rightTimes.size() - 1);
        }
        if (leftTimes.size() > 0 && le instanceof Divide &&
            ((Divide) le).exp1.eval().equals("1")) {
            TimesArrDown.add(((Divide) le).exp2);
            leftTimes.remove(leftTimes.size() - 1);
        }
        TimesArrUp = GeneralFunc.addTwoList(rightTimes, leftTimes);

        if (TimesArrDown.size() > 0) {
            chkTmp = GeneralFunc.expTimesArrayList(TimesArrDown, str);
            TimesArrDown.clear();
            if (chkTmp instanceof Times || chkTmp instanceof Divide) {
                TimesArrDown = chkTmp.OpRepeate(str);
            } else {
                TimesArrDown.add(chkTmp);
            }
        }
```

```
        // Return Simplify of List
        return   GeneralFunc.SimplifyTimesList(TimesArrUp,   TimesArrDown
str);
        // Simple same element in up and down
        // Simple Up (and Down) list(s) Separate
        // Simple Union's element in up or down
        // Simple GCD's element in up and down
        // Remove Sqrt in down list
        // Simple Up (and Down) list(s)
        // Etc.
    }
    …
}

class Divide extends Exp {

    Exp exp1, exp2, tmp1, tmp2;

    public Divide(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }

    public Exp fog(Exp g) {

        return new Divide(exp1.fog(g), exp2.fog(g));
    }


    public Exp Simplify(String str) {
        int i, j;
        Exp chkTmp;
        ArrayList<Exp> rightTimes = new ArrayList<Exp>();
        ArrayList<Exp> TimesArrDown = new ArrayList<Exp>();
        ArrayList<Exp> TimesArrUp = new ArrayList<Exp>();

        // special cases
        if (exp1.eval().equals("0")) {
            GeneralFunc.chkChng = 1;
            return new Num(0, 1);
        }
        if (exp2.eval().equals("0")) {
            GeneralFunc.chkChng = -1;
            return new Num(0, 1);
        }
        if (exp2.eval().equals("1")) {
            GeneralFunc.chkChng = 1;
            return exp1;
        }

        if (exp1 instanceof Num && exp2 instanceof Num) {
            Data d =
            GeneralFunc.DevideOp(exp1.getNumValue(), exp2.getNumValue());
            return new Num(d.num1, d.num2);
        }

        // Create Up (and Down) List(s)
        if (exp1 instanceof Times || exp1 instanceof Divide) {
            TimesArrUp = exp1.OpRepeate(str);
        } else {
```

```
            TimesArrUp.add(exp1);
        }
        Exp TU = TimesArrUp.get(TimesArrUp.size() - 1);
        Exp TD = TimesArrDown.get(TimesArrDown.size() - 1);
        if (TimesArrUp.size() > 0 && TU instanceof Divide &&
            ((Divide) TU).exp1.eval().equals("1")) {
            exp2 = new Times(exp2, ((Divide) TU).exp2);
            TimesArrUp.remove(TimesArrUp.size() - 1);
        }

        if (exp2 instanceof Times || exp2 instanceof Divide) {
            TimesArrDown = exp2.OpRepeate(str);
        } else {
            TimesArrDown.add(exp2);
        }

        // Return Simplify of List
        return   GeneralFunc.SimplifyDividList(TimesArrUp,   TimesArrDown
str);
        // Simple same element in up and down
        // Simple Up (and Down) list(s) factoring
        // Simple Union's element in up or down
        // Simple GCD's element in up and down
        // Remove Sqrt in down list
        // Simple Up (and Down) list(s)
        // Etc.

    }
  …

}

class Power extends Exp {

    Exp exp1, exp2;

    public Power(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }

    public Exp fog(Exp g) {

        return new Power(exp1.fog(g), exp2.fog(g));
    }

    public Exp Simplify(String str) {
        Data val1;
        ArrayList<Exp> rightPower = new ArrayList<Exp>();
        ArrayList<Exp> leftPower = new ArrayList<Exp>();
        ArrayList<Exp> arrPower = new ArrayList<Exp>();
        if (exp2.eval().equals("1")) {
            return exp1;
        }
        if (GeneralFunc.numPowChk == 1 && exp1 instanceof Num
            && exp2 instanceof Num) {
            double a = exp1.getNumValue().num1 / exp1.getNumValue().num2;
            double b = exp2.getNumValue().num1 / exp2.getNumValue().num2;

            double c = Math.pow(a, b);
```

```
            int d = ((int) c);
            return new Num(d, 1);
        }
        if (exp2.eval().substring(0, 1).equals("-")) {
            exp2 = GeneralFunc.completeSimplify(
                            new Times(new Num(-1, 1), exp2), str);
            Exp t = null;
            if (exp2.eval().equals("1")) {
                t = new Divide(new Num(1, 1), exp1);
            } else {
                t = new Divide(new Num(1, 1), new Power(exp1, exp2));
            }
            return t;
        }
        if (exp1.eval().equals("1")) {
            GeneralFunc.chkChng = 1;
            return new Num(1, 1);
        }
        if (exp1.eval().equals("0")) {
            GeneralFunc.chkChng = 1;
            return exp1;
        }
        if (exp2.eval().equals("1")) {
            GeneralFunc.chkChng = 1;
            return exp1;
        }
        if (exp2.eval().equals("0")) {
            GeneralFunc.chkChng = 1;
            return new Num(1, 1);
        }
        if (exp1 instanceof Sqt) {
            GeneralFunc.chkChng = 1;
            return new Sqt(
                 new Power(((Sqt) exp1).exp1, exp2), ((Sqt) exp1).exp2);
        }
        if (exp1 instanceof Power) {
            rightPower = exp1.OpRepeate(str);
        } else {
            rightPower.add(exp1);
        }
        if (exp2 instanceof Power) {
            leftPower = exp2.OpRepeate(str);
        } else {
            leftPower.add(exp2);
        }

        arrPower = GeneralFunc.addTwoList(rightPower, leftPower);

        // Return Simplify of List
        return GeneralFunc.SimplifyPowerList(ArrPower, str);
        // Simple list Separate
        // Simplify multipowered of List
        // Simplify multipowered of List
        // Simplify multipoly based
        // Simplify Unions
        // Etc.

    }
    …
}
```

```
class Sqt extends Exp {

    Exp exp1, exp2;

    public Sqt(Exp e) {
        exp1 = e;
        exp2 = new Num(2, 1);
    }

    public Sqt(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }

    public Exp fog(Exp g) {

        return new Sqt(exp1.fog(g), exp2.fog(g));
    }

    public Exp Simplify(String str) {
        Exp retTmpOut = null, retTmpIn = null;
        ArrayList<Exp> arrTimes = new ArrayList<Exp>();
        ArrayList<Exp> arrDivide = new ArrayList<Exp>();
        ArrayList<Exp> tmpArr = new ArrayList<Exp>();

        // Special cases
        if (exp1.eval().equals("1") || exp1.eval().equals("0")) {
            GeneralFunc.chkChng = 1;
            return exp1;
        }

        exp1 = exp1.Simplify(str);
        if (GeneralFunc.chkChng != 0) {
            return new Sqt(exp1, exp2);
        }
        exp2 = exp2.Simplify(str);
        if (GeneralFunc.chkChng != 0) {
            return new Sqt(exp1, exp2);
        }

        // Create Up(arrTimes) & Down(arrDivide) List
        if (exp1 instanceof Times || exp1 instanceof Divide) {
            arrTimes = exp1.OpRepeate(str);
        } else {
            arrTimes.add(exp1);
        }
        Exp aT =  arrTimes.get(arrTimes.size() - 1);
        while (arrTimes.size() > 0 && aT instanceof Divide &&
               ((Divide) aT).exp1.eval().equals("1")) {
            Exp   chkTmp   =   ((Divide)   arrTimes.get(arrTimes.size()   -
1)).exp2;
            arrTimes.remove(arrTimes.size() - 1);
            if (chkTmp instanceof Times || chkTmp instanceof Divide) {
                tmpArr = chkTmp.OpRepeate(str);
            } else {
                tmpArr.add(chkTmp);
            }
            arrDivide = GeneralFunc.addTwoList(arrDivide, tmpArr);
```

```
        // Return Simplify of List
        return GeneralFunc.SimplifyPowerList(ArrPower, str);
        // Simple Up (and Down) list(s) Separate
        // Simple Up
        // Etc.

    }
    …
}

class Sin extends Exp {

    Exp exp;

    public Sin(Exp e) {
        exp = e;
    }

    public Exp fog(Exp g) {

        return new Sin(exp.fog(g));
    }

    public Exp Simplify(String str) {
        String s = exp.eval();
        if (s.equals("0") || s.equals("Ï€") ||
            s.equals("2Ï€") || s.equals("180") || s.equals("360")) {
            return new Num(0, 1);
        } else if (exp.eval().equals("Ï€/2")) {
            return new Num(1, 1);
        } else if (exp.eval().equals("Ï€/4")) {
            return new Divide(
                 new Sqt(new Num(2, 1), new Num(2, 1)), new Num(2, 1));
        } else if (exp.eval().equals("3Ï€/2")) {
            return new Num(-1, 1);
        }

        exp = exp.Simplify(str);
        return new Sin(exp);
    }
    …
}

class Cos extends Exp {

    Exp exp;
    public Cos(Exp e) {
        exp = e;
    }

    public Exp fog(Exp g) {

        return new Cos(exp.fog(g));
    }

    public Exp Simplify(String str) {
         ⋮
    }
    ⋮
}
```

```
⁞

class Num extends Exp {

    int num1, num2;

    public Num(int n1, int n2) {
        num1 = n1;
        num2 = n2;
    }

    public Exp fog(Exp g) {

        return new Num(num1, num2);
    }

    public Exp Simplify(String str) {
        int GCD = GeneralFunc.findNumGCD(num1, num2);
        if (GCD != 1) {
            num1 /= GCD;
            num2 /= GCD;
            if (num2 < 0) {
                num2 *= -1;
                num1 *= -1;
            } else {
                GeneralFunc.chkChng = 1;
            }
        }
        return new Num(num1, num2);
    }
    ⁞

}

class Var extends Exp {

    String s;

    public Var(String str) {
        s = str;
    }

    public Exp fog(Exp g) {

        return g;
    }

    public Exp Simplify(String str) {
        return new Var(s);
    }
    ⁞

}
```

Appendix C – Some General Functions

```
package SymbolicApp;
import java.util.ArrayList;

    static int n = 14;
    public static int chkChng, numPowChk = 1;
    public static boolean PlusNumber = true, PlusSameValue = true,
                          PlusFraction = true;
    public static boolean RemoveDownSqrt = true, DivideGCD = true,
                          DivideUnion = true, DivideFactoring = false;
    public static Boolean TimesPlusMinusValue = true, DividePlusMinusOpen = true;
    public static boolean aboveSamePower = false;
    public static String[] opStrings = {"Sadelestirme", "Turev",
                          "1.Dereceden Denklem","2.Dereceden Denklem",
                          "f(g(x)) den f(x) i bulma","fonksiyon ters alma",
                          "Polinomial Bolme", "Polinomial GCD", ")) hesaplama"};

public class OperationClasses {

    // Simplification of the equation and display it in a list box
    // ------------------------------------------------------------
    public static Equ simplestEquShowListBox(
                ResultHolder obj, Exp e1, Exp e2, String str) {
        Equ e;
        String ex1, ex2;
        while (true) {
            GeneralFunc.chkChng = 0;
            ex1 = e1.eval();
            ex2 = e2.eval();
            obj.add(ex1 + "=" + ex2);
            e1 = e1.Simplest(str);
            if (GeneralFunc.chkChng == -1) {
                obj.add("!Devide by Zero!");
                break;
            }
            GeneralFunc.chkChng = 0;
            e2 = e2.Simplest(str);
            if (GeneralFunc.chkChng == -1) {
                obj.add("Devide by Zero!");
                break;
            }
            if (e1.eval().equals(ex1) && e2.eval().equals(ex2)) {
                break;
            }
        }
        e = new Equ(e1, e2);
        return e;
    }

    // Simplification of the expression and display it in a list box
    // ------------------------------------------------------------
    public static Exp simplestExpShowListBox(
                              ResultHolder obj, Exp e1, String str) {
        String ex1;
        int i = 0;
        while (true) {
            GeneralFunc.chkChng = 0;
```

```
            ex1 = e1.eval();
            if (i == 0) {
                obj.add(ex1);
                i = 1;
            } else {
                obj.add("=!" + ex1);
            }
            e1 = e1.Simplest(str);
            if (GeneralFunc.chkChng == -1) {
                obj.add("Devide by Zero!");
                break;
            }
            if (e1.eval().equals(ex1)) {
                break;
            }
        }
        return e1;
    }

    // Solve of the first-degree equation and display it in a list box
    // --------------------------------------------------------------

    public static Equ onestDegreeEquShowList(
                ResultHolder obj, Exp exp1, Exp exp2, String str) {
        Equ eqTmp;
        Exp expTmp;
        ArrayList<Exp> arrPlus1 = new ArrayList<Exp>();
        ArrayList<Exp> arrPlus2 = new ArrayList<Exp>();

        eqTmp = OpClasses.simplestEquShowListBox(obj, exp1, exp2, str);
        exp1 = ((Equ) eqTmp).exp1;
        exp2 = ((Equ) eqTmp).exp2;

        if (exp1 instanceof Plus || exp1 instanceof Minus) {
            arrPlus1 = exp1.OpRepeate(str);
        } else {
            arrPlus1.add(exp1);
        }
        if (exp2 instanceof Plus || exp2 instanceof Minus) {
            arrPlus2 = exp2.OpRepeate(str);
        } else {
            arrPlus2.add(exp2);
        }
        for (int k = 0; k < arrPlus1.size(); k++) {
            if (arrPlus1.get(k) instanceof Num) {
                exp1 = new Minus(exp1, arrPlus1.get(k));
                exp2 = new Minus(exp2, arrPlus1.get(k));
            }
        }
        for (int k = 0; k < arrPlus2.size(); k++) {
            if (!(arrPlus2.get(k) instanceof Num)) {
                exp1 = new Minus(exp1, arrPlus2.get(k));
                exp2 = new Minus(exp2, arrPlus2.get(k));
            }
        }

        eqTmp = OpClasses.simplestEquShowListBox(obj, exp1, exp2, str);
        exp1 = ((Equ) eqTmp).exp1;
        exp2 = ((Equ) eqTmp).exp2;
```

```
        expTmp = GeneralFunc.NumExp(exp1, str);
        exp1 = new Divide(exp1, expTmp);
        exp2 = new Divide(exp2, expTmp);

        eqTmp = OpClasses.simplestEquShowListBox(obj, exp1, exp2, str);

        return eqTmp;
    }

    // Solve of quadratic equation an show it in a list box
    // -------------------------------------------------------------
    public static Equ twesttDegreeEquShowList(
                    ResultHolder obj, Exp exp1, Exp exp2, String str) {
        Equ eqTmp;
        Exp expTmp;
        ArrayList<Exp> arrPlus = new ArrayList<Exp>();

        eqTmp = OpClasses.simplestEquShowListBox(obj, exp1, exp2, str);
        exp1 = ((Equ) eqTmp).exp1;
        exp2 = ((Equ) eqTmp).exp2;

        exp1 = new Minus(exp1, exp2);
        exp2 = new Num(0, 1);
        eqTmp = OpClasses.simplestEquShowListBox(obj, exp1, exp2, str);

        String sTmp;
        exp1 = ((Equ) eqTmp).exp1;
        sTmp = exp1.eval();
        if (exp1 instanceof Plus || exp1 instanceof Minus) {
            arrPlus = exp1.OpRepeate(str);
        } else {
            arrPlus.add(exp1);
        }
        Exp a = new Num(0, 1), b = new Num(0, 1), c = new Num(0, 1);
        Exp delta;
        double d;
        for (int k = 0; k < arrPlus.size(); k++) {
            d = GeneralFunc.Degree(arrPlus.get(k), str);
            if (d == 2.0) {
                a = GeneralFunc.NumExp(arrPlus.get(k), str);
            }
            if (d == 1.0) {
                b = GeneralFunc.NumExp(arrPlus.get(k), str);
            }
            if (d == 0.0) {
                c = GeneralFunc.NumExp(arrPlus.get(k), str);
            }
        }
        exp1 = new Plus(new Plus(new Times(a, new Power(new Var(str),
                        new Num(2, 1))), new Times(b, new Var(str))), c);
        if (!exp1.eval().equals(sTmp)) {
            obj.add(exp1.eval() + "=0");
        }

        delta = new Minus(new Power(b, new Num(2, 1)),
                new Times(new Num(4, 1), new Times(a, c))); // b^2-4ac
        obj.add(lineStr);
        obj.add("a = " + a.eval());
        obj.add("b = " + b.eval());
        obj.add("c = " + c.eval());
```

```
        obj.add(lineStr);
        String deltaStr = delta.eval();
        int chk = GeneralFunc.numPowChk;
        GeneralFunc.numPowChk = 1;
        delta = GeneralFunc.completeSimplest(delta, str);
        obj.add("d = " + deltaStr);
        obj.add("d = " + delta.eval());
        obj.add(lineStr);

        if (delta.getNumValue().num1 < 0) {
            obj.add("#Delta 0 dan kucuk ve bu denklemin real koku yok");
        } else {
            Exp xExp1, xExp2;
            String xStr1, xStr2;

            xExp1 = new Divide(new Plus(new Times(new Num(-1, 1), b),
                    new Sqt(delta, new Num(2, 1))),
                    new Times(new Num(2, 1), a));
                    // x1= (-b + sqrt(delta))/2a
            xExp2 = new Divide(new Minus(new Times(new Num(-1, 1), b),
                    new Sqt(delta, new Num(2, 1))),
                    new Times(new Num(2, 1), a));
                    // x1= (-b - sqrt(delta))/2a
            xStr1 = xExp1.eval();
            xStr2 = xExp2.eval();
            xExp1 = GeneralFunc.completeSimplest(xExp1, str);
            xExp2 = GeneralFunc.completeSimplest(xExp2, str);

            String x1 = xExp1.eval();
            String x2 = xExp2.eval();
            obj.add("x = " + x1);
            if (!x1.equals(x2)) {
                obj.add("x = " + x2);
            }
        }
        GeneralFunc.numPowChk = chk;
        return eqTmp;
    }

    // Divide two polynomials an display it in a list box
    // -------------------------------------------------------------
    public static devideData dividePolynomialsShowList(
      ResultHolder obj1, ResultHolder obj2, Exp exp1, Exp exp2, String
str)
      {
        devideData retTmp = new devideData();

        exp1 = GeneralFunc.completeSimplest(exp1, str);
        exp2 = GeneralFunc.completeSimplest(exp2, str);

        exp1= GeneralFunc.completeSimplest(GeneralFunc.sortExp(exp1, str), str);
       exp2 = GeneralFunc.completeSimplest(GeneralFunc.sortExp(exp2, str), str);
        obj1.clear();
        obj2.clear();
        obj1.add(exp1.eval());
        obj2.add(exp2.eval());
        Exp exp1Tmp = exp1;
        Exp tmpQ = GeneralFunc.quotientExps(exp1, exp2, str);
        if (tmpQ == null) {
            tmpQ = new Num(0, 1);
```

```
    }
    Boolean firstChk = false;
    Exp exp3 = new Num(0, 1), exp4 = new Num(1, 1);
    while ((!tmpQ.eval().equals("0") && !exp4.eval().equals("0")) ||
            firstChk == false) {

        firstChk = true;
        if (exp3.eval().equals("0")) {
            exp3 = tmpQ;
        } else {
            exp3 = new Plus(exp3, tmpQ);
        }
        Exp exp5 = new Times(tmpQ, exp2).clone();
        exp4 = GeneralFunc.completeSimplest(exp5, str);
        GeneralFunc.PlusFraction = !GeneralFunc.PlusFraction;
        exp4 = GeneralFunc.completeSimplest(exp4, str);
        GeneralFunc.PlusFraction = !GeneralFunc.PlusFraction;

        exp4 = GeneralFunc.sortExp(exp4.clone(), str);
        Exp ttt = new Minus(exp1Tmp, exp4);
        Exp tmpexp = GeneralFunc.completeSimplest(ttt.clone(), str);

        GeneralFunc.PlusFraction = !GeneralFunc.PlusFraction;
        tmpexp = GeneralFunc.completeSimplest(tmpexp, str);
        GeneralFunc.PlusFraction = !GeneralFunc.PlusFraction;

        exp1Tmp = GeneralFunc.sortExp(tmpexp.clone(), str);
        obj2.add(exp3.eval());
        obj1.add("-" + exp4.eval());
        headStr1 += "    ";
        obj1.add(" " + exp1Tmp.eval());

        if (exp1.eval().equals("0")) {
            break;
        }
        tmpQ = GeneralFunc.quotientExps(exp1Tmp, exp2, str);

    }

    String s = "";
    s = s + new Times(exp3, exp2).laTex();
    if (!exp1.eval().equals("0")) {
        s = s + "+" + exp1.laTex();
    }
    s = s + "=" + exp1Tmp.laTex();

    retTmp.Result = exp3;
    retTmp.Remind = exp1;

    return retTmp;
}

// GCD of two polynomials an display in a list box
// ------------------------------------------------
public static Exp polynomialsGCDShowList(
            ResultHolder obj1, Exp exp1, Exp exp2, String str) {

    exp1 = GeneralFunc.completeSimplest(exp1, str);
    exp2 = GeneralFunc.completeSimplest(exp2, str);
```

```
        exp1 = GeneralFunc.sortExp(GeneralFunc.completeSimplest(exp1, str), str);
        exp2 = GeneralFunc.sortExp(GeneralFunc.completeSimplest(exp2, str), str);

         devideData   remindTmp1   =   GeneralFunc.numFactorPolynomial(exp1,
str);
         devideData   remindTmp2   =   GeneralFunc.numFactorPolynomial(exp2,
str);

         ArrayList<Exp> g = GeneralFunc.findMonomialGCD(
                              remindTmp1.Result, remindTmp2.Result, str);
         Exp g1 = g.get(0);
         exp1 = GeneralFunc.devideMonomials(exp1, g1, str).Result;
         exp2 = GeneralFunc.devideMonomials(exp2, g1, str).Result;

        exp1 = GeneralFunc.sortExp(GeneralFunc.completeSimplest(exp1, str), str);
        exp2 = GeneralFunc.sortExp(GeneralFunc.completeSimplest(exp2, str), str);

         obj1.clear();
         if (g1.eval().equals("1")) {
             obj1.add("1.Polinomial = !" + exp1.eval());
             obj1.add("2.Polinomial = !" + exp2.eval());
         } else {
             obj1.add("1.Polinomial =!("+ g1.eval()+")("+ exp1.eval()+ ")");
            obj1.add("2.Polinomial =!(" + g1.eval()+")("+exp2.eval()+ ")");
         }
         obj1.add(lineStr);
         if (exp1.eval().equals("0") || exp2.eval().equals("0")) {
             obj1.add("Error in data Entry!!");
             return null;
         }
         Exp exp1Tmp = exp1;
         devideData tmp = new devideData();
         devideData remindTmp = new devideData();

         tmp = GeneralFunc.fastDevide(exp1, exp2, str);
         tmp.Remind = GeneralFunc.completeSimplest(tmp.Remind, str);
         if (tmp.Result.eval().equals("0")) {
             Exp tt = exp1;
             exp1 = exp2;
             exp2 = tt;
             if (g1.eval().equals("1")) {
                 obj1.add("1.Polinomial = !" + exp1.eval());
                 obj1.add("2.Polinomial = !" + exp2.eval());
             } else {
                 obj1.add("1.Polinomial =
                      !(" + g1.eval() + ")(" + exp1.eval() + ")");
                 obj1.add("2.Polinomial =
                      !(" + g1.eval() + ")(" + exp2.eval() + ")");
             }
             obj1.add(lineStr);

             tmp = GeneralFunc.fastDevide(exp1, exp2, str);
             tmp.Remind = GeneralFunc.completeSimplest(tmp.Remind, str);
         }
         if (tmp.Remind == null) {
             tmp.Remind = new Num(0, 1);
         }
         while (!tmp.Remind.eval().equals("0") &&
                !tmp.Result.eval().equals("0")) {

             if (tmp.Remind instanceof Divide) {
```

```
                tmp.Remind = ((Divide) tmp.Remind).exp1;
            }
            remindTmp = GeneralFunc.numFactorPolynomial(tmp.Remind, str);
            if (remindTmp.Result.eval().equals("1")) {
                obj1.add("Devide Result = !" + tmp.Result.eval());
                obj1.add("Remain = !" + tmp.Remind.eval() + "= " +
                          remindTmp.Remind.eval());
            } else {
                obj1.add("Devide Result = !" + tmp.Result.eval());
                obj1.add("Remain = !" + tmp.Remind.eval() + "= (" +
                          remindTmp.Result.eval() + ")(" +
                          remindTmp.Remind.eval() + ")");
            }

            obj1.add(lineStr);
            exp1 = exp2;
            exp2 = remindTmp.Remind;
            obj1.add("1.Polinomial = !" + exp1.eval());
            obj1.add("2.Polinomial = !" + exp2.eval());
            obj1.add(lineStr);
            tmp = GeneralFunc.fastDevide(exp1, exp2, str);
            tmp.Remind = GeneralFunc.completeSimplest(tmp.Remind, str);
        }
        if (tmp.Result.eval().equals("0")) {
            exp2 = new Num(1, 1);
        }
        obj1.add("Devide Result = !" + tmp.Result.eval());
        obj1.add("Remain = !" + tmp.Remind.eval());
        obj1.add(lineStr);
        String s = "";
        if (exp2.eval().equals("-1")) {
            s = "1";
        } else {
            s = exp2.eval();
        }
        if (g1.eval().equals("1")) {
            obj1.add("GCD = !" + s);
        } else {
            obj1.add("GCD = !(" + g1.eval() + ")(" + s + ")");
        }

        return null;
    }


   // Polynomial factoring
   // ---------------------------------------

    public static ArrayList<Exp> factorExp(Exp e, String str) {

        ArrayList<Exp> arrPlus = new ArrayList<Exp>();
        ArrayList<Exp> GCDTmp = new ArrayList<Exp>();
        if (e instanceof Plus || e instanceof Minus) {
            arrPlus = e.OpRepeate(str);
        } else if (e instanceof Num) {
            GCDTmp.add(e);
            GCDTmp.add(new Num(1, 1));
            return GCDTmp;
        } else {
            GCDTmp.add(new Num(1, 1));
```

```
            GCDTmp.add(e);
            return GCDTmp;
        }

        Exp GCD = arrPlus.get(0);
        for (int i = 1; i < arrPlus.size(); i++) {
            Exp tmp = arrPlus.get(i);
            GCDTmp = GeneralFunc.findMonomialGCD(GCD, tmp, str);
            GCD = GCDTmp.get(0);
        }

        for (int i = 0; i < arrPlus.size(); i++) {
            Exp tmp = arrPlus.get(i);
            devideData tt = GeneralFunc.devideMonomials(tmp.clone(), GCD, str);
            arrPlus.set(i, tt.Result);
        }

        GCDTmp.clear();
        GCDTmp.add(GCD);
        GCDTmp.add(GeneralFunc.expPlusArrayList(arrPlus, str));

        return GCDTmp;
    }

// used classes
//-----------------------
class Data {

    int num1, num2;
    String S;
}

class devideData {

    Exp Result;
    Exp Remind;
}

class ArrayLists {

    ArrayList<Exp> firstArr = new ArrayList<Exp>();
    ArrayList<Exp> secondArr = new ArrayList<Exp>();
}
```

Appendix D – Derivative methods in AST and Taylor Function

```
package SymbolicApp;

import java.util.ArrayList;
⁞
abstract class Exp {
    …
    public abstract Exp Derived();
}

class Plus extends Exp {

    Exp exp1, exp2, tmp;
    public Plus(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }

    public Exp Derived() {
        return (new Plus(exp1.Derived(), exp2.Derived()));
    }
    …
}

class Minus extends Exp {

    Exp exp1, exp2;
    Data val2;
    Exp tmp;
    public Minus(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }

    public Exp Derived() {
        return (new Minus(exp1.Derived(), exp2.Derived()));
    }
    …
}

class Times extends Exp {

    Exp exp1, exp2, tmp;
    Data val2;

    public Times(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }

    public Exp Derived() {
        if (exp1 instanceof Num && exp2 instanceof Num) {
            return new Num(0, 1);
        } else if (exp1 instanceof Num) {
            return new Times(exp1, exp2.Derived());
        } else if (exp2 instanceof Num) {
            return new Times(exp1.Derived(), exp2);
        } else {
```

```
            return (new Plus(new Times(exp1.Derived(), exp2),
                      new Times(exp1, exp2.Derived())));
        }
    }
    …
}

class Divide extends Exp {

    Exp exp1, exp2, tmp1, tmp2;
    public Divide(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }

    public Exp Derived() {
        return (new Divide(new Minus(new Times(exp1.Derived(), exp2),
           new Times(exp1, exp2.Derived())), new Power(exp2, new Num(2,
1))));
    }
  …
}

class Power extends Exp {

    Exp exp1, exp2;
    public Power(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }

    public Exp Derived() {
        double s;
        s = Double.parseDouble(exp2.eval()) - 1;
        if (exp1 instanceof Var) {
            return (new Times(exp2, new Power(exp1, new Num((int) s,
1))));
        }
        return (
          new Times(exp2,
            new Times(exp1.Derived(), new Power(exp1, new Num((int) s,
1)))));
    }
    …
}

class Sqt extends Exp {

    Exp exp1, exp2;
    public Sqt(Exp e) {
        exp1 = e;
        exp2 = new Num(2, 1);
    }

    public Sqt(Exp e1, Exp e2) {
        exp1 = e1;
        exp2 = e2;
    }

    public Exp Derived() {
```

```
        return (new Divide(exp1, new Times(exp2, new Sqt(new Power(exp1,
                                 new   Minus(exp2,   new   Num(1,   1))),
exp2))));
    }
    …
}

class Sin extends Exp {

    Exp exp;
    public Sin(Exp e) {
        exp = e;
    }

    public Exp Derived() {
        return (new Times(exp.Derived(), new Cos(exp)));
    }
    …
}

class Cos extends Exp {

    Exp exp;
    public Cos(Exp e) {
        exp = e;
    }
    public Exp Derived() {
        return (new Times(new Times(new Num(-1, 1), exp.Derived()),
                new Sin(exp)));
    }
    ⁞
}
⁞
class Num extends Exp {

    int num1, num2;
    public Num(int n1, int n2) {
        num1 = n1;
        num2 = n2;
    }

    public Exp Derived() {
        return (new Num(0, 1));
    }
    ⁞
}

class Var extends Exp {

    String s;
    public Var(String str) {
        s = str;
    }

    public Exp Derived() {
        return (new Num(1, 1));
    }
   ⁞
}
```

- *Construction of the Taylor Series*

For the Taylor series expansion of an input function *f(x)*, the system calculates the successive derivatives of the function, and represents the related series through follow:

$$f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f^{(3)}(x_0)}{3!}(x - x_0)^3 + \cdots$$

The derivative of an expression is symbolically obtained by using the basic derivative rules, for example," the derivative of the sum of two expressions equals the sum of their derivatives", as it was explained in the previous section. The new expression produced in a derivation stage would possibly have a considerable number of terms, and thus followed by a simplification one. In this way, the following functions are required to design the related system.

- *Function to calculate the derivative symbolically: (was explained in the previous section)*
- *Function to perform the simplification symbolically: (will be explained in the future sections)*
- *Function to calculate the value of derived function at the x_0. (It is similar to calculating the value of a function, which is described in the previous sections.)*
- *Function to find Taylor series symbolically using the equation.*

Below we describe about how to get the Taylor series. Having the underlying function, a tree related to the Taylor expansion can conveniently be created using flowchart algorithm in Figure 64.

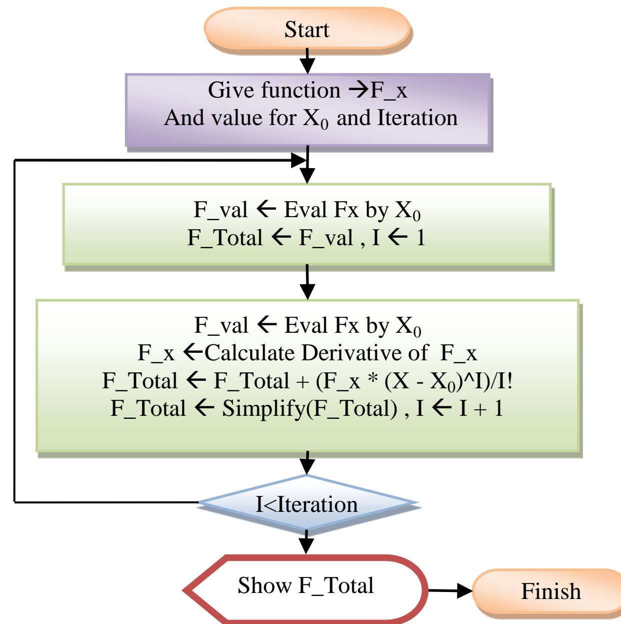
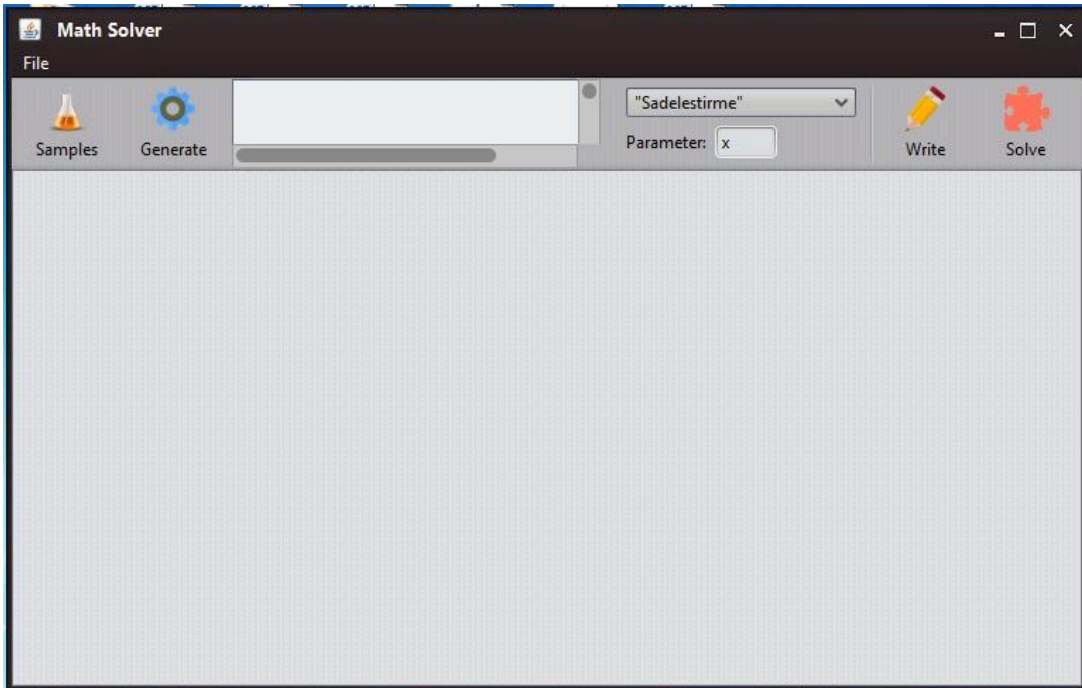


Figure 64 - Flowchart for obtaining Taylor series

According to the flowchart above, having received an expression from the user and having generated an initial parse tree, as well as considering the related equation, the trees of successive derivatives are generated. Note that for calculating the n^{th} derivative, the simplified form of the $(n-1)^{th}$ derivative is used. Every resulting derivative is a tree, which according to the equation, should connect under a single tree, known as Taylor expression tree. The number of derivatives must appropriately be received from the user. The simplification function used in the generation of the series is called after not only every differentiation, but also the final generation.

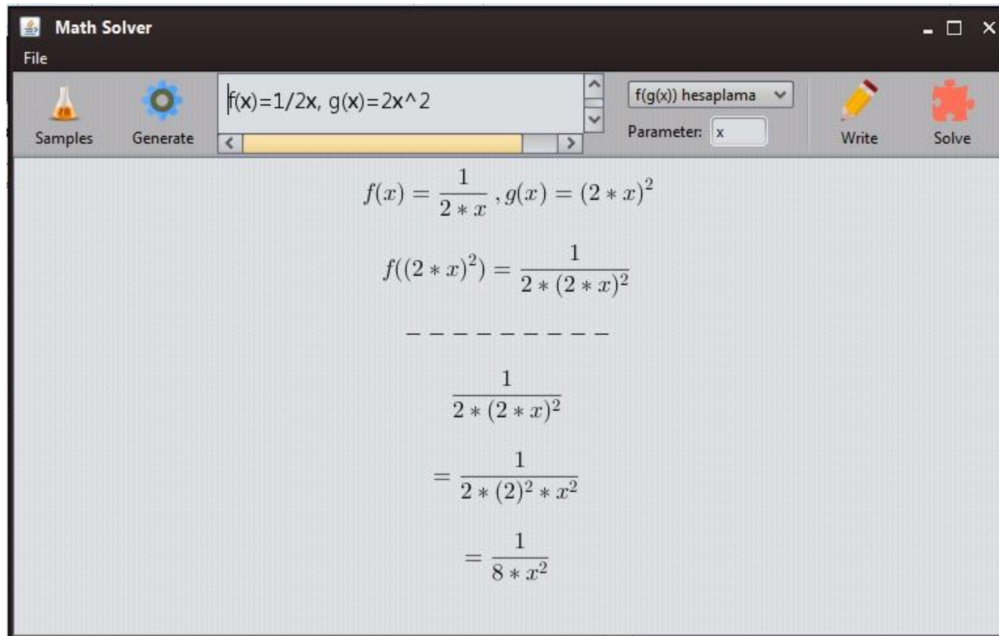
Appendix E – Screen Shots of program

- General view of the program.

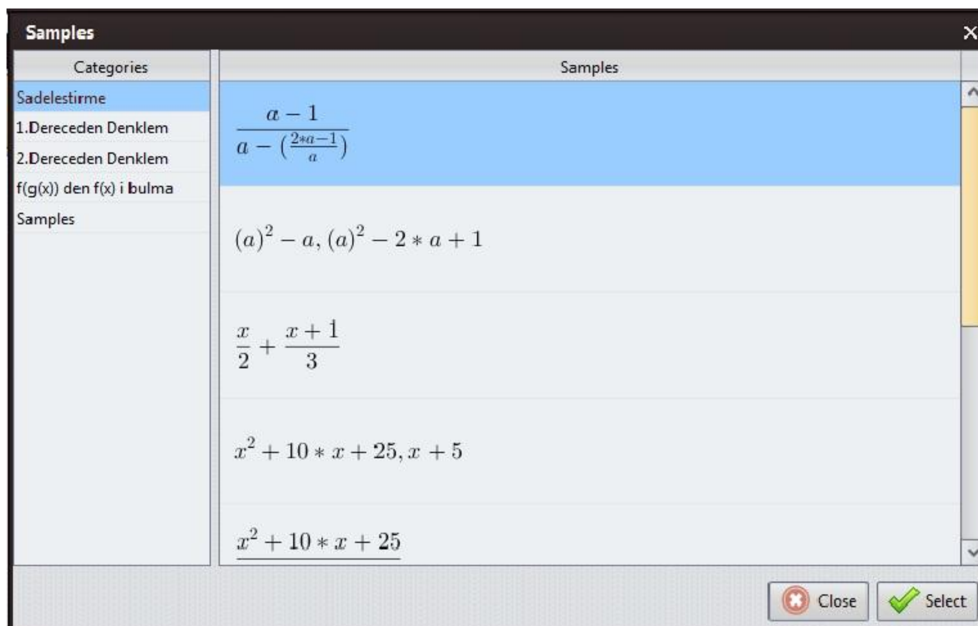


- A view of simplifying "(x^2+10x+5)/(x+5)" expression

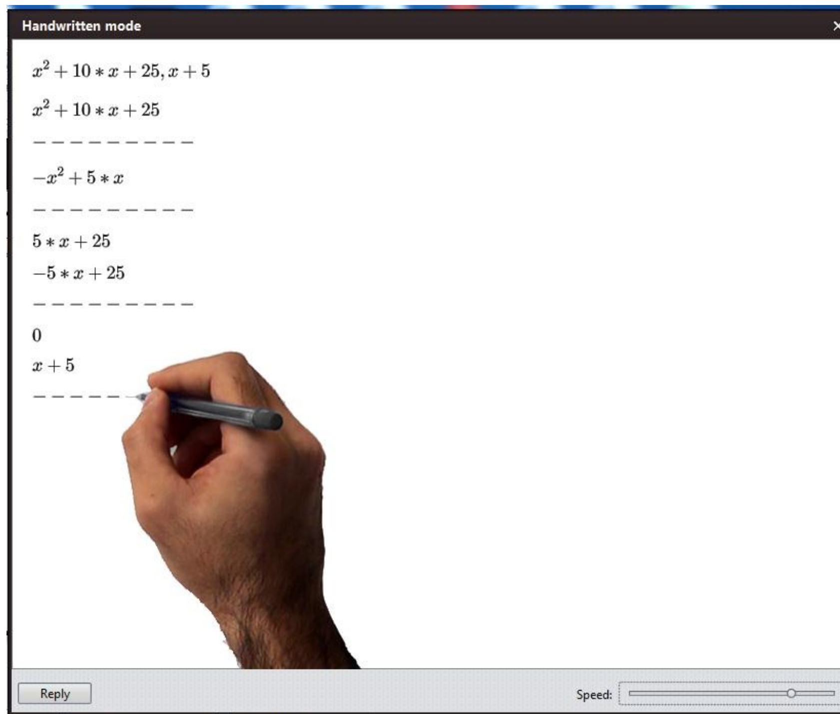
- An Example for fog operation



- Some example of solvable expressions

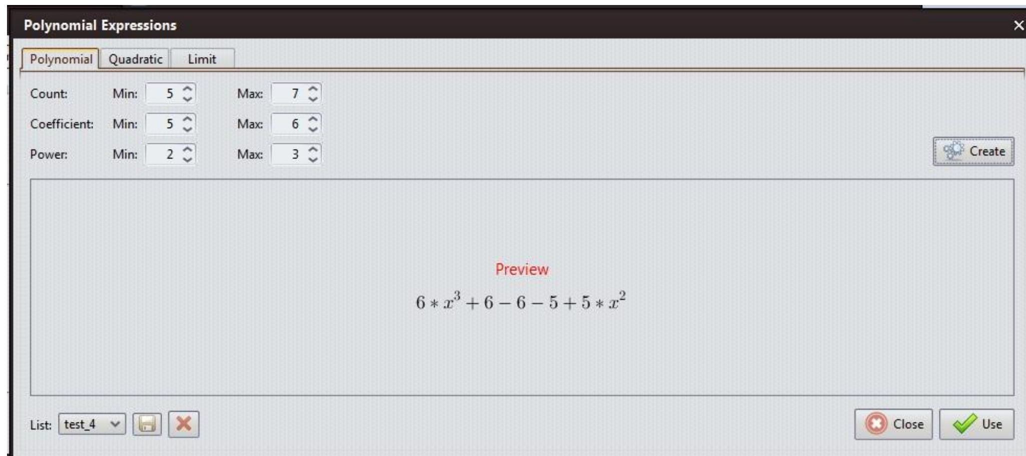


- View of animation writing for solved problems

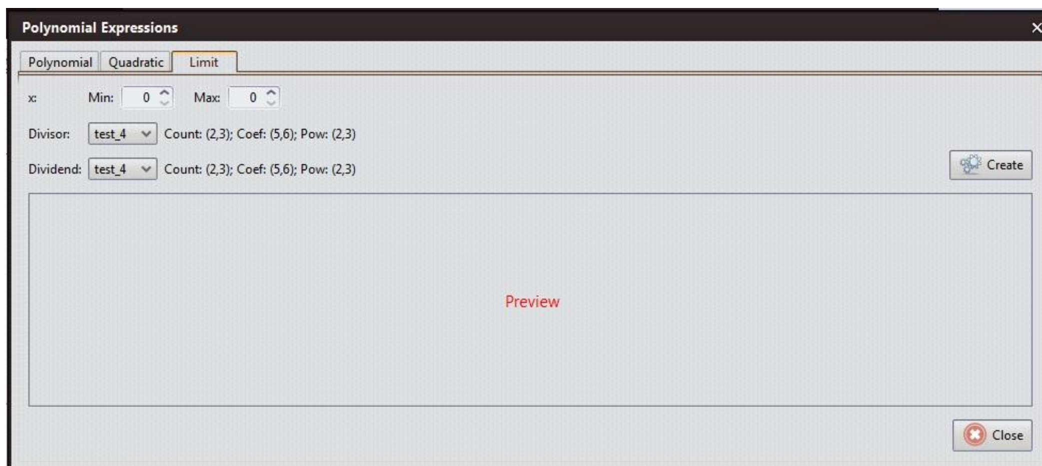


- Page for set and create polynomials randomly

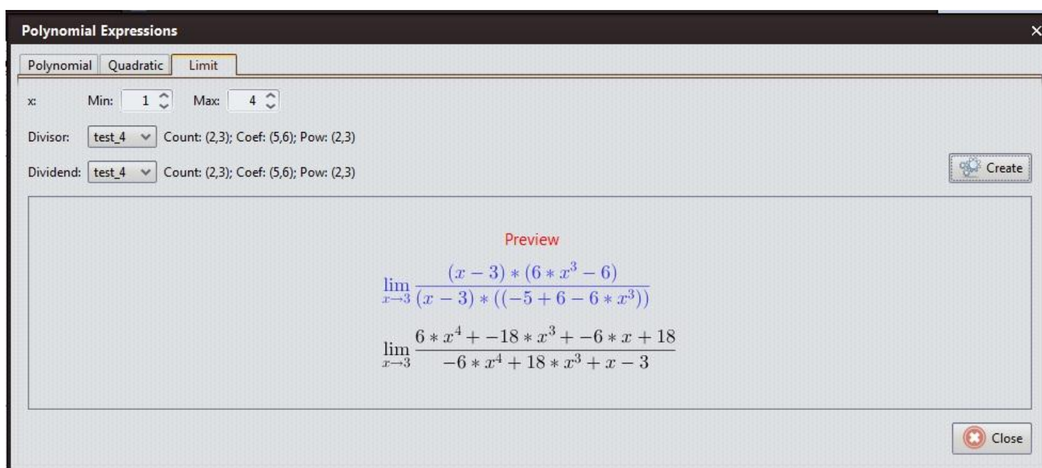
- Example view of a generated polynomial



- Page for set and create Limit randomly



- Example view of a generated Limit



- Example view of a generated Quadratic

Polynomial Expressions

Polynomial | Quadratic | Limit

Left side: test_4 Count: (2,3); Coef: (5,6); Pow: (0,2)

Right side: test_4 Count: (2,3); Coef: (5,6); Pow: (0,2)

Create

Preview

$$-6 * x + 6 * 1 = -6 + 5 + 6$$

Close Use

CURRICULM VITAE

Mir Mohammad Reza Alavi Milani was born on 1973 in Tabriz. He graduated from Taleghani High School in in 1992.He got his B.Sc.E degree in 1997 on Computer Engineering at Tehran, M.Sc.E degree in 2006 on Information Technology at Amir Kabir Technical University, Tehranö and Ph.D. degree in 2015 on Computer Engineering at Karadeniz Technical University, Trabzon. His primary research area is Symbolic Computing, Functional Programming and Cryptography. He has worked for fifteen years as teacher at technical schools. Since 2015, he has been working as an Assistant Professor at Department of Computer Engineering in Avrasya University.