



الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Syrian Private University
Faculty of Informatics & Computer Engineering

SNA²

A Mathematical Program

Junior Project

Prepared By

Saleh Al-Hakeem
Nour Aldeen Arar
Adham Khwaldeh
Akram Lazkane

Under Supervision Of

Dr. Samir Jafar

This page left blank on purpose

Table Of Contents :

Title	Page
List of figures	4
List of tables	5
Abstract	6
Introduction	7
What is SNA ² ?	8
Chapter 1 : System Analysis	9
1.1 Actor Specification	9
1.2 Use Case Diagram	10
1.3 Use Case Description	11
Chapter 2 : System Design	15
2.1 Class Diagram	15
2.2 Flow of The Program	17
2.3 Form of Input	18
Chapter 3 : System Implementation	21
3.1 Under The Hood	21
3.2 Implementing The Core	23
3.3 Implementing The Commands	25
3.4 Supportive Functions	31
Chapter 4 : System Testing	34
Conclusion and future Visions	36
References	37

List Of Figures :

Figure	Page
Figure 1.1 : Use Case Diagram	10
Figure 2.1 : Class Diagram	16
Figure 2.2 : Flow of The Program	17
Figure 3.1 : The Overall Structure of The Boost Spirit Library	22
Figure 3.2 : Trie Representation	23
Figure 3.3 : Node Structure	24
Figure 3.4 : Tree Structure	24
Figure 3.5 : Variable Structure	25
Figure 3.6 : Scope Structure	25
Figure 3.7 : Solve Structure	27
Figure 3.8 : Activity Diagram for FormSetup();	27
Figure 3.9 : Activity Diagram for Run();	27
Figure 3.10 : Activity Diagram for RunFromForm();	28
Figure 3.11 : Activity Diagram for Run Form	28
Figure 3.12 : Form Structure	29
Figure 3.13 : Activity Diagram for Add Form to Trie	29
Figure 3.14 : Activity Diagram for Clone, Value and Print	31
Figure 3.15 : General Flow for Order	31
Figure 3.16 : Adding Node to Ordered List	32
Figure 4.1 : System Testing 1	34
Figure 4.2 : System Testing 2	35

List Of Tables :

Table	Page
Table 1.1 Actor specification	9
Table 1.2 Use Case Description Tables	11

Abstract

This project is a mathematical program called **SNA**², and it is aimed to help users in mathematical fields.

This program can solve polynomial mathematical equations from variant degrees, it can show you the steps taken to achieve the result for pedagogical and academic purposes.

This program strength points that we can input the equations in the natural mathematical form also it can learn new ways from users to use it later.

Introduction

Most of the time people find it hard to solve mathematical equations, especially if they don't know how, it also takes a long time to get hard equations solved, and the most known mathematical program MATLAB is too big on medium devices, so our goal was as Unix and Linux did, is to develop a kernel with the main operators and process inside, then you can add modules that can solve polynomial mathematical equations from variant degrees, also the kernel can show you the steps taken to achieve the result for pedagogical and academic purposes, also it can learn new ways from users to use it later.

So our work context was generally in the mathematical field.

After developing our initial idea, we now have a kernel that we will build up on in the future, with high performance calculating process.

This report is organized in five chapters starting from the system analysis through system design and implementation and ending with system testing.

What is SNA²

SNA² provides an environment which can solve various mathematical problems, and provide insight on the steps taken to achieve that solution.

The user has various commands at his disposal. Each command allows for a different result to occur within this environment, with each affecting the environment in the way they treat mathematical problems that were inputted.

Chapter 1 : System Analysis

In this chapter we are going to discuss the steps taken to analysis the system in three parts :

1. Actor Specification.
2. Use case diagram.
3. Use case description.

1.1 Actor Specification

In this section we present the actor specification table

Actor Name: User		
Type: Primary	Personality: External, Initiator & Receiver	Abstract: No
Role Description: The user interact with the system by queries for either solving problems or suppling more ways to solve, in addition the user can learn the algorithms .		
Actor Goals <ul style="list-style-type: none">• Input equations to solve.• Add new forms, algorithms or for non-present solutions for problems.		
Use Cases Involved with: <ul style="list-style-type: none">• Input Query		

Table 1.1: Actor Specification

1.2 Use Case Diagram

A **use case diagram** at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved.

A use case diagram can identify the different types of users of a system and the different use cases and will often be accompanied by other types of diagrams as well.

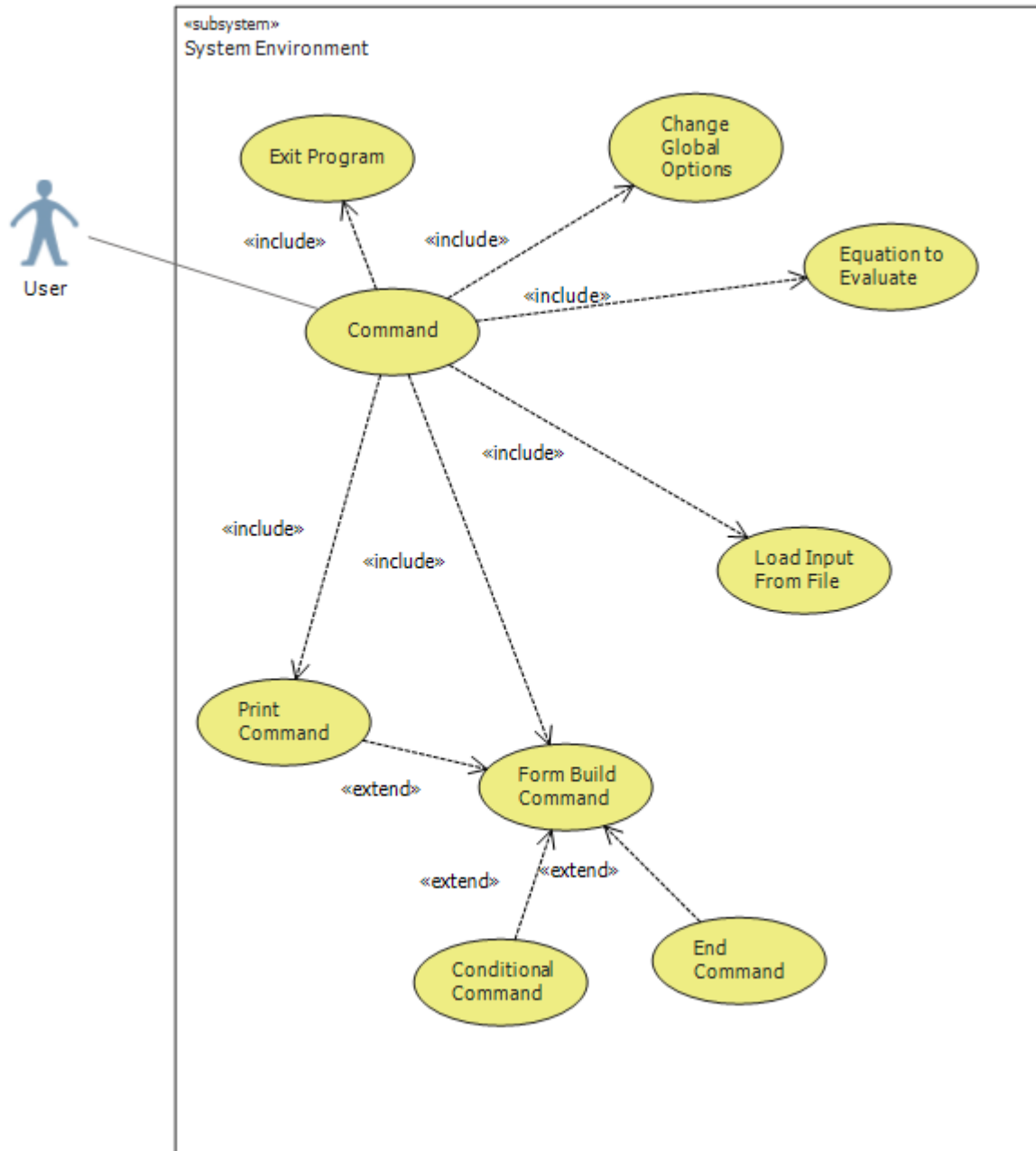


Figure 1.1: Use Case Diagram

1.3 Use Case Description

For agile development, a requirement model of many UML diagrams depicting use cases plus some textual descriptions, notes or use case briefs would be very lightweight and just enough for small or easy project use.

Number	1
Name	<i>Input Query</i>
Summary	<i>The user inputs a query</i>
Actors	<i>Primary: User</i>
Pre-Conditions	-
Scenario	<ol style="list-style-type: none">1. <i>Write a Query</i>2. <i>Press Enter</i>
Exceptions	<ul style="list-style-type: none">• <i>The user inputted an empty query</i>• <i>The user inputted a non valid query</i>
Post-Conditions	<i>The user receives an output based on his query</i>

Number	2
Name	<i>Global Option Change</i>
Summary	<i>The user can change options that affect the program</i>
Actors	<i>Primary : User</i>
Pre-Conditions	-
Scenario	<ol style="list-style-type: none">1. <i>Check option if exists</i>2. <i>Toggle option</i>
Exceptions	<ul style="list-style-type: none">• <i>Option doesn't exists</i>
Post-Conditions	<i>The system altered.</i>

Number	3
Name	<i>Equation to evaluate</i>
Summary	<i>The equation is evaluated and a result based on the evaluation is made.</i>
Actors	<i>Primary : User</i>
Pre-Conditions	-
Scenario	<ol style="list-style-type: none"> 1. <i>Evaluate Expression</i> 2. <i>Refine</i> 3. <i>Find Related Form</i> 4. <i>Solve</i>
Exceptions	<ul style="list-style-type: none"> • <i>No form found.</i> • <i>Invalid equation</i>
Post-Conditions	<i>The equation is solved</i>

Number	4
Name	<i>Exit Program</i>
Summary	<i>The Program shuts down</i>
Actors	<i>Primary : User</i>
Pre-Conditions	-
Scenario	<ol style="list-style-type: none"> 1. <i>Shut down the program</i>
Exceptions	-
Post-Conditions	<i>Program has exited</i>

Number	5
Name	<i>Load Input from File</i>
Summary	<i>The file gets loaded and all commands within are parsed.</i>
Actors	<i>Primary : User</i>
Pre-Conditions	<i>File exists</i>
Scenario	<ol style="list-style-type: none"> 1. <i>Check if File Exists</i> 2. <i>Open To read from</i> 3. <i>Parse all available commands from within.</i>
Exceptions	<ul style="list-style-type: none"> • <i>File doesn't exists</i> • <i>All exceptions from other cases apply here</i>
Post-Conditions	<i>The whole file is parsed</i>

Number	6
Name	<i>Print Command</i>
Summary	<i>Prints Whatever was held along the Print command</i>
Actors	<i>Primary : User</i>
Pre-Conditions	-
Scenario	<ol style="list-style-type: none"> 1. <i>Print</i>
Exceptions	<ul style="list-style-type: none"> • <i>Invalid input</i>
Post-Conditions	<i>Printed on screen</i>

Number	7
Name	<i>Form Build</i>
Summary	<i>A new Form is Added by the user to the environment.</i>
Actors	<i>Primary : User</i>
Pre-Conditions	-
Scenario	<ol style="list-style-type: none"> 1. <i>Check command</i> 2. <i>Evaluate Form</i> 3. <i>Setup Procedures</i> 4. <i>Add to list of Forms</i>
Exceptions	<ul style="list-style-type: none"> ● <i>Invalid input or form</i>
Post-Conditions	<i>Equations of the new form are now solvable</i>

Number	8
Name	<i>Conditional command</i>
Summary	<i>Allows there to be a conditional branching in forms</i>
Actors	<i>Primary : User</i>
Pre-Conditions	<i>Used within the form do statement</i>
Scenario	<ol style="list-style-type: none"> 1. <i>Setup condition</i> 2. <i>Setup children</i>
Exceptions	<ul style="list-style-type: none"> ● <i>Invalid input or condition</i>
Post-Conditions	<i>A conditional branch within the form</i>

Number	9
Name	<i>End command</i>
Summary	<i>Tells the Form that when met, no more procedures are to be executed.</i>
Actors	<i>Primary : User</i>
Pre-Conditions	-
Scenario	-
Exceptions	-
Post-Conditions	<i>The Form Procedure running exits on hit from command</i>

Chapter 2 : System Design

In this chapter we are going to discuss the steps taken to design the system in one part :

1. Class diagram.
2. Flow of the program.
3. Form of input.

2.1 Class Diagram

A **class diagram** is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

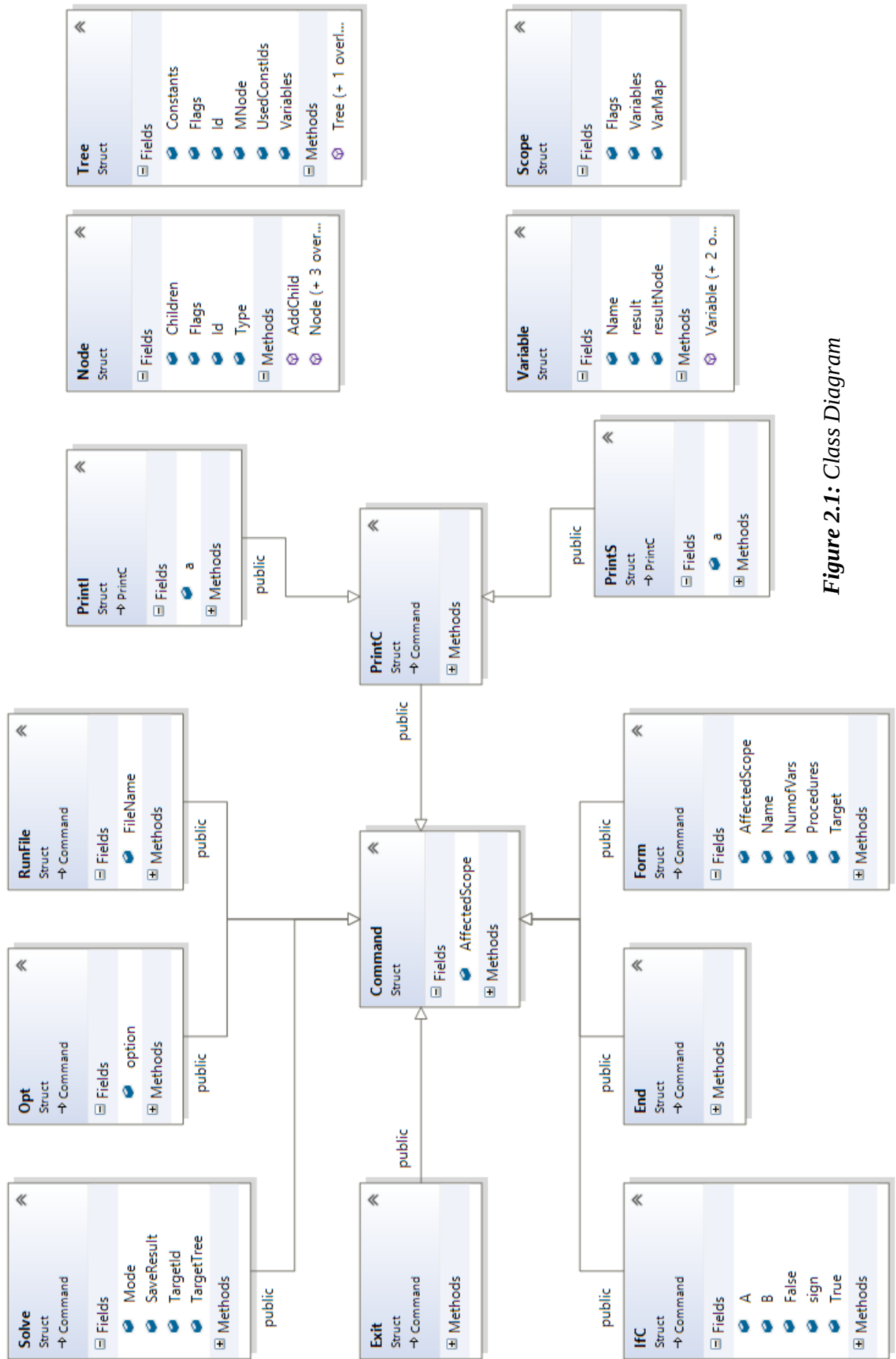


Figure 2.1: Class Diagram

2.2 Flow of The Program

The Program handles the input in the following way:

- Waits for user input.
- Parses the input and extracts the needed data in the form of a [*Command*] and other types of data.
- Depending on what type of command was inputted, the environment applies it and returns a result to the user.
- The program returns to wait for the user input.

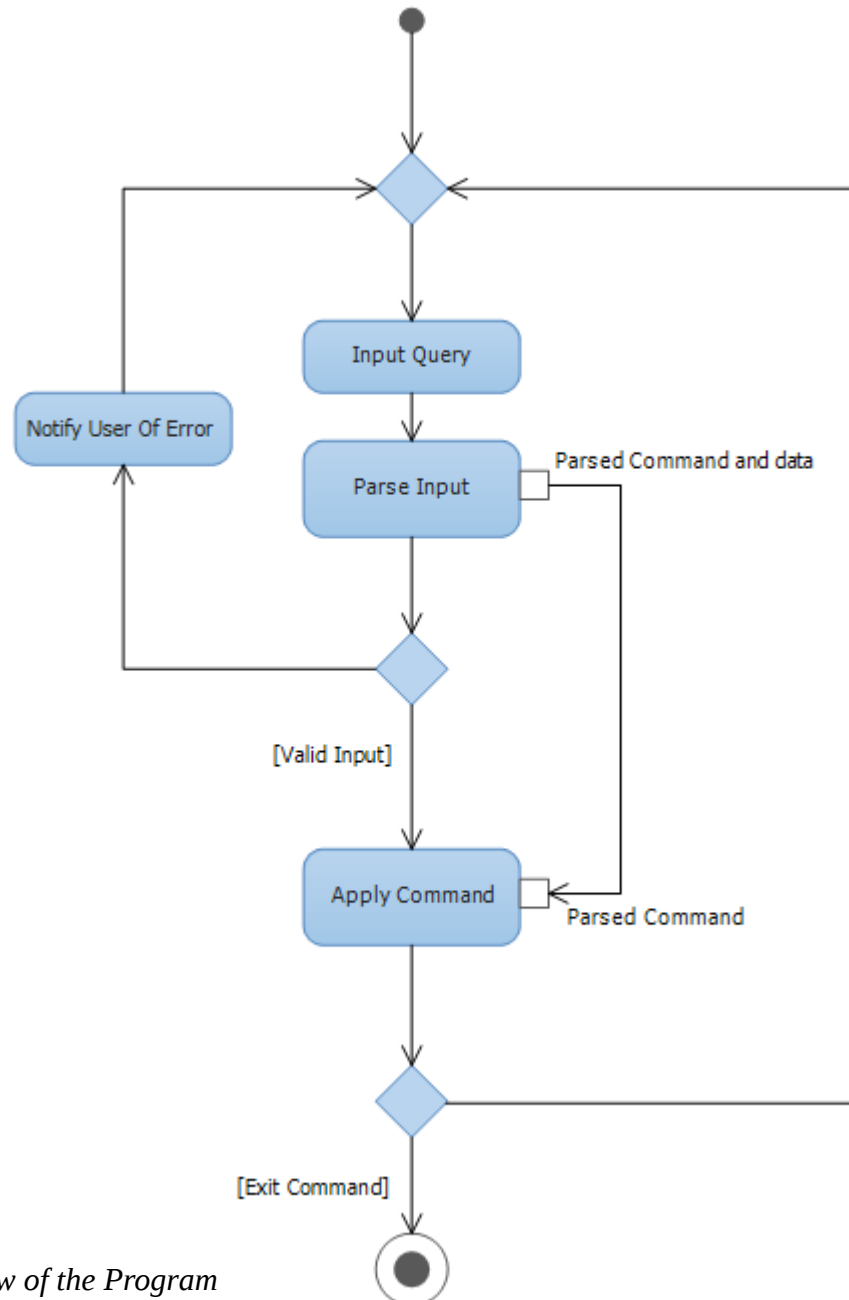


Figure 2.2: Flow of the Program

2.3 Form of Input

As it receives the input, **SNA**² starts its parsing cycle, trying to make sense of it. There is a definite form that the user has to adhere to for the program to recognize what has been inputted.

The equation uses the following operators:

Sign	Purpose	Example
+	Addition - Positive	a+b OR +a
-	Subtraction or Negative	a-b OR -a
*	Multiplication	a*b
/	Division	a/b
^	Exponents	a^b which represents a ^b

The user has to adhere to these rules when handling equations, otherwise the program would not be able to recognize it.

For example : $1 + 2x + 3x^2$ should be written as $1 + 2*x + 3*x^2$

The user can also use a string of characters denoting a variable. However that string should be compromised of letters and the underscore.

As for commands, There Forms that the user should abide to will be mentioned in the following section.

➤ *Available Commands*

The user has various commands at his disposal, each following a different form of input.

But most of them follow the general form of:

[Command] : [Respective_Input]

We now providing the following commands, and along with their form a brief description of what each of them is.

- **Solve command:**

- The Solve command, which is stated as default, takes the respective input of an equation.

- It would have the following form:

[Equation] OR Solve: [Equation]

- The Solve command, when applied, will take the Equation given and try to find a solution for it, and based on the solution, the user should receive an output.

- If the user wishes to associate the [Equation] with a certain identifier, it should take the following form:

[identifier] = [Equation]

- **Print command:**

Print "String" OR Print [identifier]

- The Print command is as its name describes, a command used to print a particular thing to the user.

- Printing the string will just output the string, while printing the Identifier will print the identifier itself, and what was saved to it as a result.

- **Option command:**

Toggle [Option]

- The Option Command is used to toggle options that affect the environment and the behavior of the algorithm used within the program.

- One of the options, Trace, allows the user to see the set of procedures applied to reach the solution.

- **Load File command:**

Load "Filename"

- The Load File command when applied takes the input from a file and parse as if it was user input, the solution to each parsed command in the File will be displayed on the console of the program.

- **Form command:**

```
Form:{  
  name: "String"  
  variable: [identifier], ...  
  form: [Equation]  
  do: [List of Commands]  
}
```

- The Form command is the corner stone of the program. The user can define a new form for the program to look for, and a set of commands (procedures) to apply if the form is met.

- The form should follow that general look, but the name, and variable lines are optional, and give the form extra information to work with. Note that if the form has a variable, it should be stated in the variable line, otherwise all identifiers will be regarded as unknown constants.

- **If command:**

```
If: [Condition]  
  { [List of Commands] }  
else  
  { [List of Commands] }
```

- The If command, which can be only used within the Form do statement, is a command set to give the user more flexibility on how to deal with the various forms that could be meant. If the condition is met, it will apply the first list of commands, but if it were false, it would apply the second.

- The else statement, and the 2nd list of commands are optional, and if the program couldn't match the Condition, it would just move on and apply nothing.

- The Condition is of the form:

[Equation] [Comparison Sign] [Equation]

- **End command:**

End

- The End command, which can be only used within the Form do statement, is a command that is used to tell the program that once it hits that command to stop applying procedures.

- **Exit command:**

Exit

- The Exit command, tells the program that it has finished its job, and that it should exit. The User should use this when they want to exit the program safely, encase anything should be saved.

Chapter 3 : System Implementation

In this chapter we are going to discuss the steps taken to implement the system and the techniques used for it.

3.1 Under The Hood

This project has been written using C++ Language, Boost library and Trie

➤ *C++*

C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation.

C++ is standardized by the International Organization for Standardization(ISO), with the latest (and current) standard version ratified and published by ISO in December 2014 as ISO/IEC 14882:2014 (informally known as C++14).



The C++ programming language was initially standardized in 1998 as ISO/IEC 14882:1998 .

➤ *Boost Library*

Boost is a set of libraries for the C++programming language that provide support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing .It contains over eighty individual libraries.



Most of the Boost libraries are licensed under the Boost Software License, designed to allow Boost to be used with both free and proprietary software projects. Many of Boost's founders are on the C++ standards committee, and several Boost libraries have been accepted for incorporation into both Technical Report 1and the C++11 standard.

Boost::Spirit is an object-oriented parser and output generation library for C++. It allows you to write grammars and format descriptions using a format similar to Extended Backus Naur Form (EBNF) directly in C++. These inline grammar specifications can mix freely with other C++ code and, thanks to the generative power of C++ templates, are immediately executable. In retrospect, conventional compiler-compilers or parser-generators have to perform an additional translation step from the source EBNF code to C or C++ code.

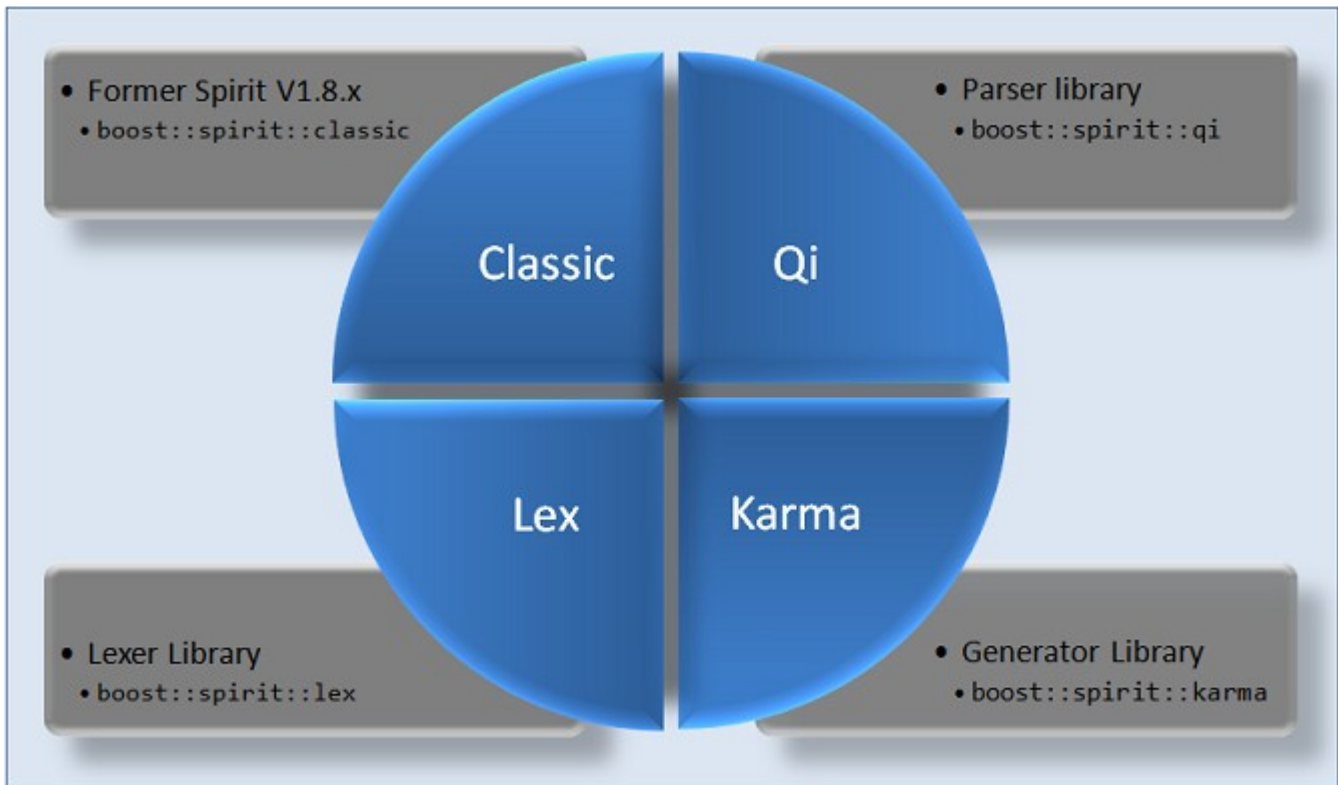


Figure 3.1: The overall structure of the Boost Spirit library

Boost::Spirit::Qi This is the parser library allowing you to build recursive descent parsers.

The exposed domain-specific language can be used to describe the grammars to implement, and the rules for storing the parsed information, so it is a Spirit's sub-library dealing with generating parsers based on a given target grammar (essentially a format description of the input data to read).

➤ The Trie

The trie is an information retrieval data structure, of which their search complexity would take up about $O(m)$ where m is the length of input.

The only downside is the space complexity of the data structure.

But with the way we designed our program, that won't be much of an issue.

Within the tree, the inner nodes consist of a value which has to be returned, and a list of pointers to the next couple of nodes. In our program the data structure used to link to other nodes is a map.

When trying to match an input, the trie is traversed, if it stays within the bounds of the trie, it would return the value of the last node it ends up on. Otherwise it would return a default value.

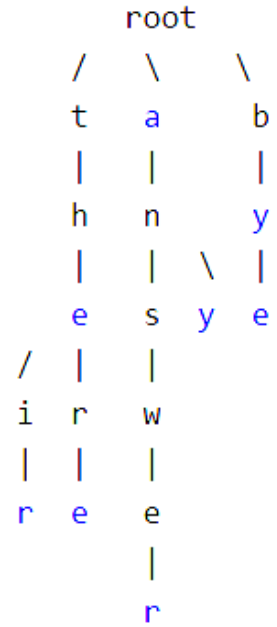


Figure 3.2: Trie Representation

3.2 Implementing The Core

We talked about the general flow of the program, the exact form the user should abide to, and the parser that was used in the program.

Now we talk about the internal workings of the program.

➤ *The Node:*

The program revolves around solving mathematical equations, so we had to represent it a purposeful manner.

We wanted a general structure that can describe the many forms that could be used, such as binary operators, constants, variables, etc...

We decided on the following structure, a Node with:

- **Type (int):** Denotes what kind of Node the current one is.
- **Id (int):** Which gives extra information on the Node that differs in meaning depending on the type.
- **Flags (int):** Allows the Node to hold extra information on the current state the Node is in.
- **Children (vector<Node*>):** which points to the children this current Node has.

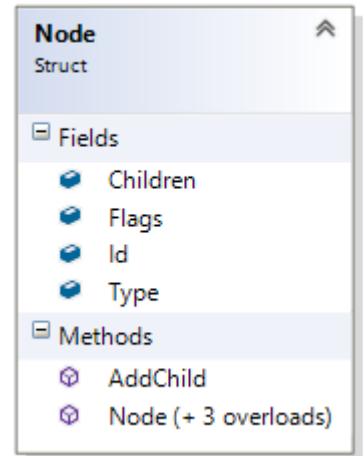


Figure 3.3: Node Structure

➤ *The Tree:*

The Tree is an extra structure that holds extra information on the tree made of nodes as a whole.

Of the most important fields in this structure:

- **Constants (vector<int>):** which holds the constants that the Tree uses, a Node of type Constant accesses this information.
- **Variables (vector<int>):** holds the Variable ID that is used within the tree.
- **MNode (Node*):** a pointer that points towards the top node of the tree.

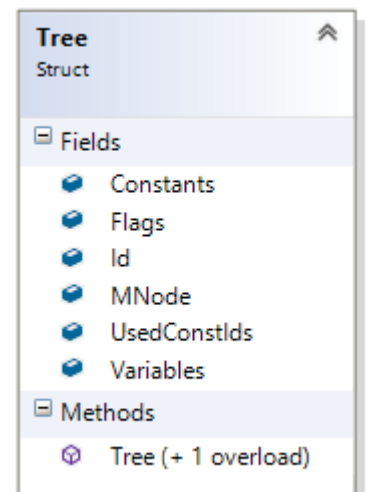


Figure 3.4: Tree Structure

➤ *The Variable:*

Holds most information the Variable needs to function.

- **Name (string):** which represents the name(identifier) that is used to access the variable
- **Result (tree*):** which represents the result that is saved on it. Points to the tree in question.
- **resultNode(Node*):** represents the Node that the variable points to, the node can be different from the tree main node, meaning it can be pointing towards a sub tree inside, with all the necessary information saved inside the tree.

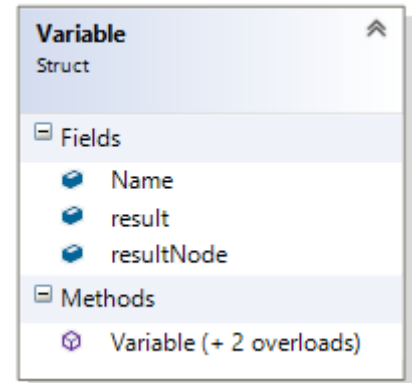


Figure 3.5: Variable Structure

➤ *The Scope:*

Holds all the information about what is occurring within that scope of action.

Separate Scopes are initialized when using a command such as form or when loading a file. The Scopes are used to separate the entities that need not be affected by outside information.

Important fields:

- **Variables (vector<Variable>):** which holds all the variable information of the variables used within this scope.
- **VarMap (map<string,int>):** which holds the ID that is assigned to the identifier, so that the program know what each identifier it passes through is.

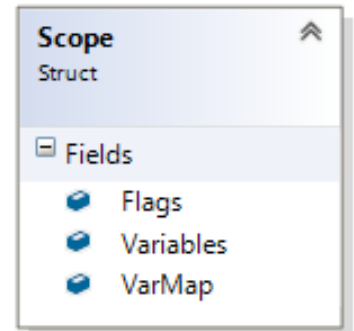


Figure 3.6: Scope Structure

3.3 Implementing the Commands

During the parsing cycle, the equations are dissected into the core structures that were presented. In that cycle, any odd information that it passes by is recorded into the flags of the right structures.

Which gets us to the command preparation cycle in which all the needed information is prepared for running the parsed command.

Lets start with the easiest of all commands.

- **Exit command:**

The keyword is the only thing that needs to be mentioned, and the program will exit.

- **Option command:**

By using Toggle keyword, the string that follows is stored with the command.

Once the command is run, it will check whether the string related to any global option, If it does, then it alters it, otherwise it would tell the user it was invalid.

- **Run command:**

Which only saves the filename. Once it is run, it checks whether the file exists or not. If it does, it constructs a new scope and moves on to handle parsing the commands within that file.

- **Print command:**

The Print command has two forms. The Print Command accepts a string, or an identifier, for both, the information is saved in a string that we called "a". If a string was found, when the command is run, it will just print out the string stored within the "a" field.

If an identifier was found, then it would print that identifier, and print any result that might be attached to it.

Now for the bit complicated commands that get executed.

- **End command:**

When met, the program knows that this is the end of the path in the set of procedures that it is applying.

- **If command:**

A conditional command, that when run, it applies the condition and chooses which set of procedures to continue executing.

- **Solve command:**

First off is the main information the Solve command needs.

- **Mode (int):** which determines mode of solution, currently set up and reserved for future releases.
- **SaveResult (bool):** which determines whether the result is going to be saved into the variable.
- **TargetId (int):** which determines which variable it is going to save to incase SaveResult has been activated.
- **TargetTree (int):** is the tree, equation basically, that is going to be the center of the command.

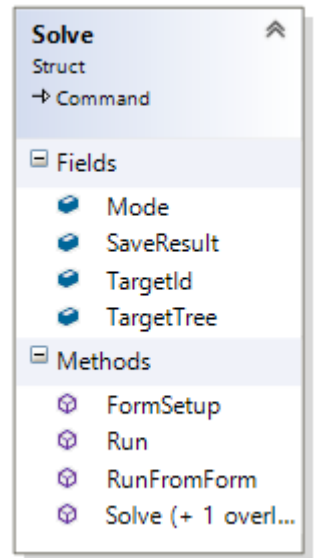


Figure 3.7: Solve Structure

The Solve command has the three main functions to work with **Run() , Formsetup(), RunFromForm()**.

When a solve command is issued from within a form, any identifier that is not present inside the variable list determined in the form will be considered an unknown constant.

- **FormSetup();**

All it does is setup everything needed for the command to work or to ease up performance when *RunFromForm()* is called.

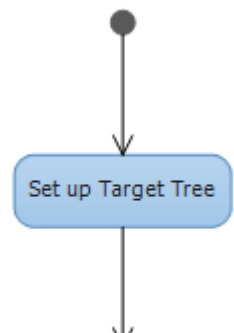


Figure 3.8: Activity Diagram for FormSetup();

- **Run();**

The main frame is simple,

It sets up the tree its going to work with.

Finds the Form it is related to, and depends on the results the command acts to notify the user.

And its fairly simple to find the form and uses the same way it does when its adding a form.

divides the tree into bases,

and goes through the trie to find its respective form.

The only difference is that in this instance it is not allowed to change the way the tree is setup.

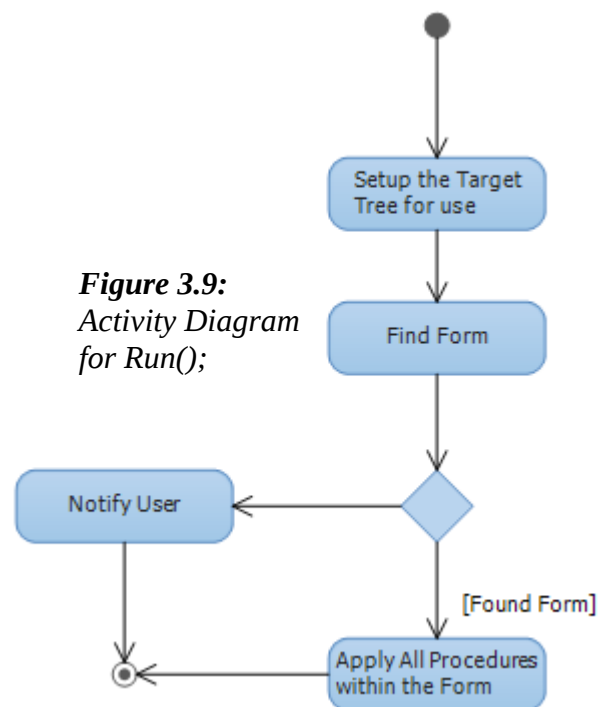


Figure 3.9: Activity Diagram for Run();

- **RunFromForm();**

It starts off by taking in the list of unknown constants with a result that has been assigned to them, Then makes a clone of the target tree with all the unknown constants replaced with their results, from that clone it checks for whether the result should be saved or not.

If it does then it saves it to the assigned identifier, which

could be an unknown constant or a variable.

On the other hand it would just apply the equation and get an answer, then dump the answer away.

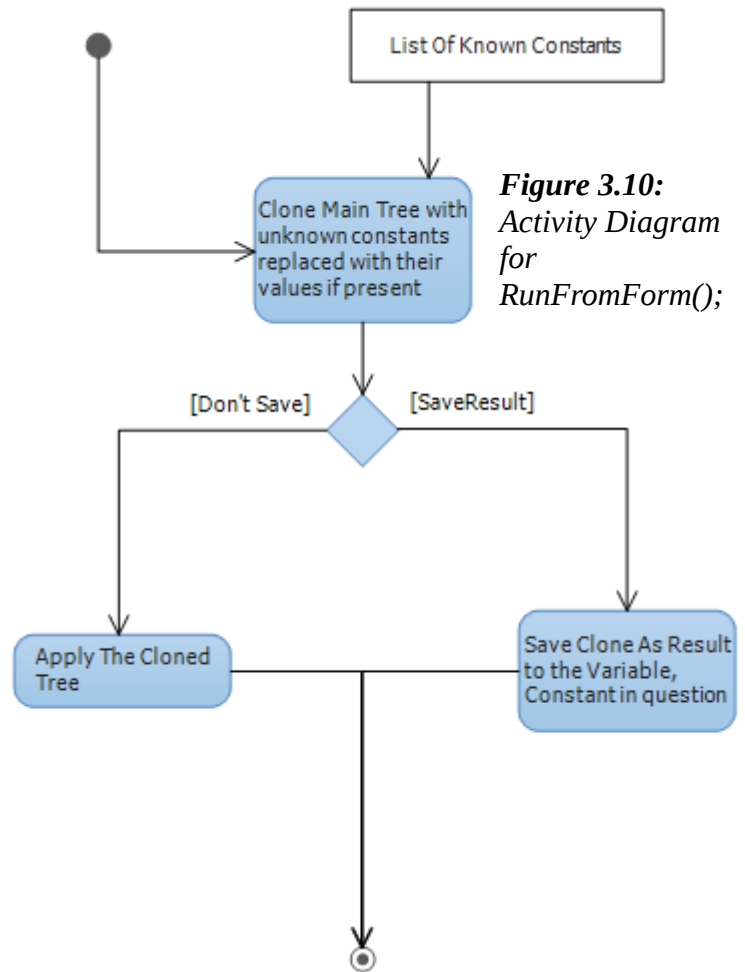


Figure 3.10: Activity Diagram for RunFromForm();

- **Form command:**

When the parser takes in four main statements that the Form command needs to execute.

The name given to the form, the variables present within the form, and the form itself, followed by the set of procedures the form has to run once it is found.

As the whole command gets structured, and ready for deployment, A set of functions are run to include the new form into the number of forms in the database.

The main flow of the command is pretty straight forward.

At first it works on the inputted data, ordering the tree, seeing if any extra work needs to be done on it before it moves on.

After that, a new form is constructed which holds all the necessary information that it would need function properly.

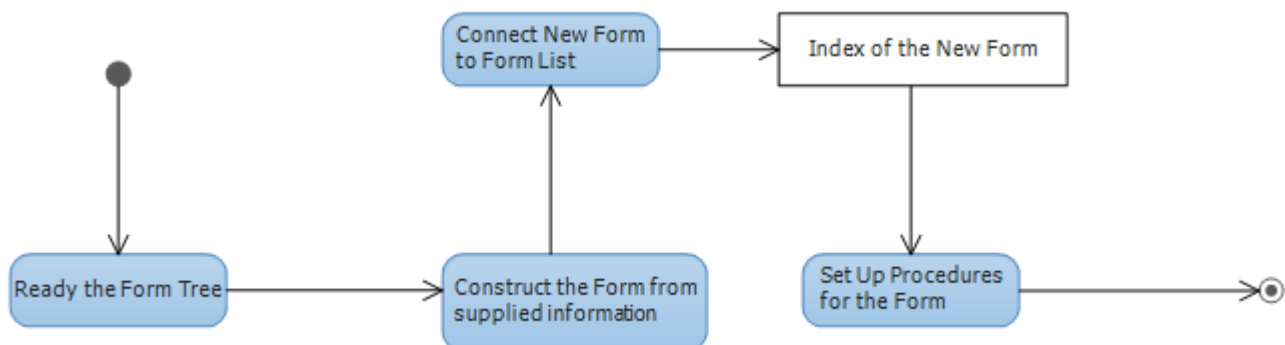
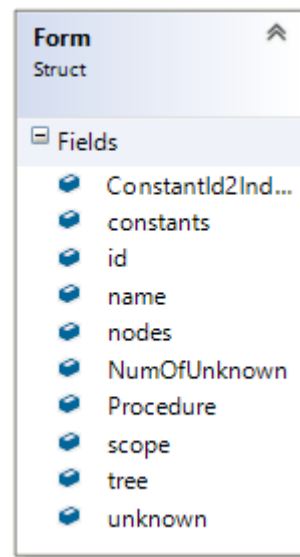


Figure 3.11: Activity Diagram for Run Form

- **Scope (Scope*)**: Which points to the scope that all the procedures within will work with once the form is found and run.
- **Tree (Tree*)**: Holds the main tree information.
- **Procedure (vector<Command*>)**: Which holds all the commands to be run as procedures when the form is found and run.
- **Name (string)**: Is the name given to the form.



Constants, unknown, nodes and *constantId2Index* are all important extra information that are used to identify which form later on during the Solve command.

Figure 3.12: Form Structure

After the form is constructed and ready to be saved, it has to be added into the list of forms and a trie like structure.

The main tree that defines the form is taken and separated into what we call bases.

Based on those bases the trie is constructed.

During the addition of a form to the trie, it would start going through the nodes one by one until it reaches the end of the list of nodes.

Along the way, if a node does not exist, a new node is constructed.

Once they reach the end of the list, the last node it falls on determines which set of forms this new form belongs to.

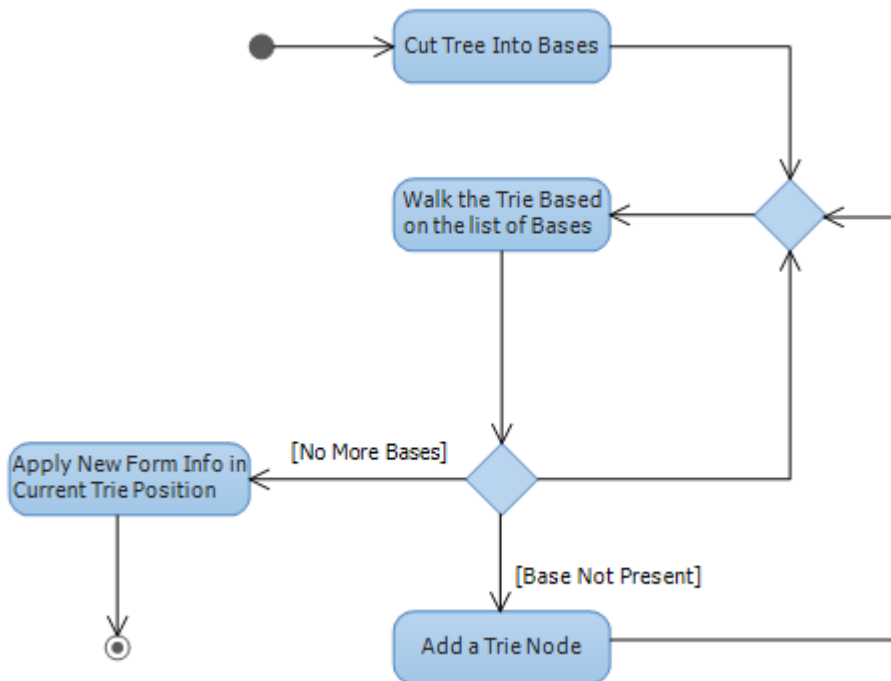


Figure 3.13: Activity Diagram for Add Form to Trie

The new form is then added to the list and now is ready to be used.

The main tree in the form is cut to bases. These bases show information on the current blocks in the tree.

They are just basically an ID, bases follow the same design that the Form follows.

The base ID is based on the multiplication block, the block is cut, then moved through the trie in the same manner the form does.

But instead of relying on just an integer, the base relies on a pair of type, and ID based on the nodes that were cut, the ID that rests on the final node it is on is then returned to the form trie.

Returning To Form, after the Form is correctly placed in the list of forms, and its place in the trie, the procedures get attached into the form and set up, and thus the form is now ready to be used.

3.4 Supportive Functions

During the setup of a tree and output, multiple functions were made to provide varied functionalities, most of them follow a similar design of which there is one exception.

The Following hold the same pattern of design, with only difference being with what it executes within, Print, Clone, and Value.

The Print outputs a designated tree depending on where it has been called, and Clone outputs a replica of a tree while Value which outputs a double which is the equivalent of the answer to the tree. The value function will throw an exception if the equation has an unknown constant or variable.

The three follow the same flow of program, which is very dependent on the current Node it is dealing with.

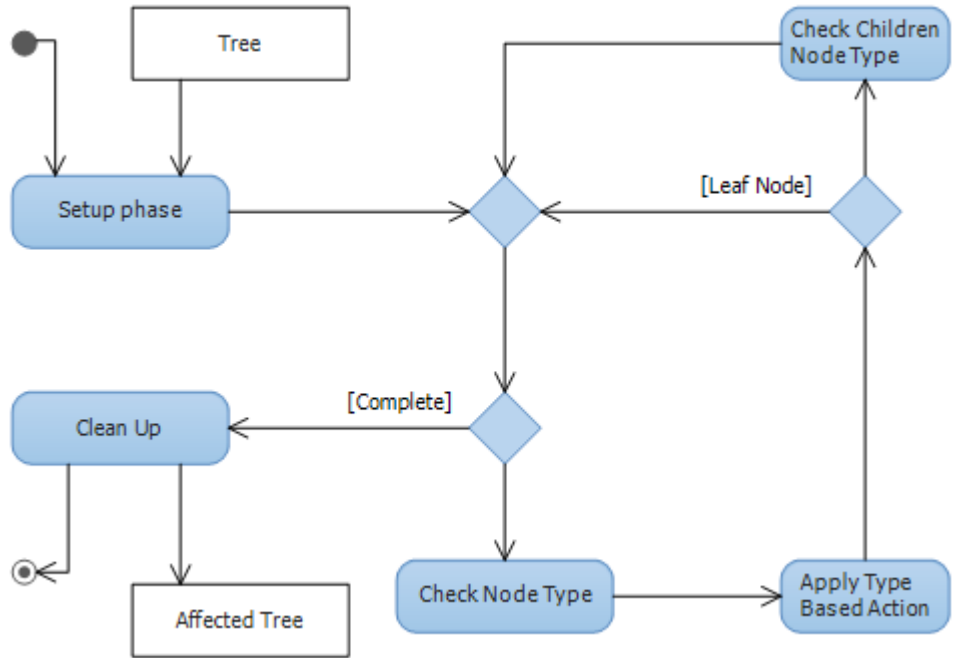


Figure 3.14: Activity Diagram for Clone, Value and Print

They start out by preparing anything they need to work with, and they go through the tree Node by node, applying whatever they need to be doing based on what type of node it exactly is, Once they finish up it cleans up anything that's left, and moves on to return their results.

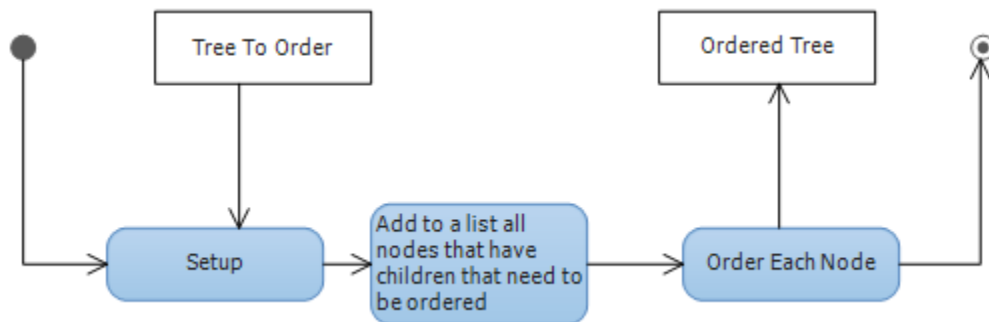


Figure 3.15: General Flow of Order

As for ordering the Elements, It follows a different approach, the general flow of the operation is simple, it takes a tree, takes all the nodes that need to be ordered and put them into the list.

After all of it is done, each node is ordered, Once done it would return the ordered tree.

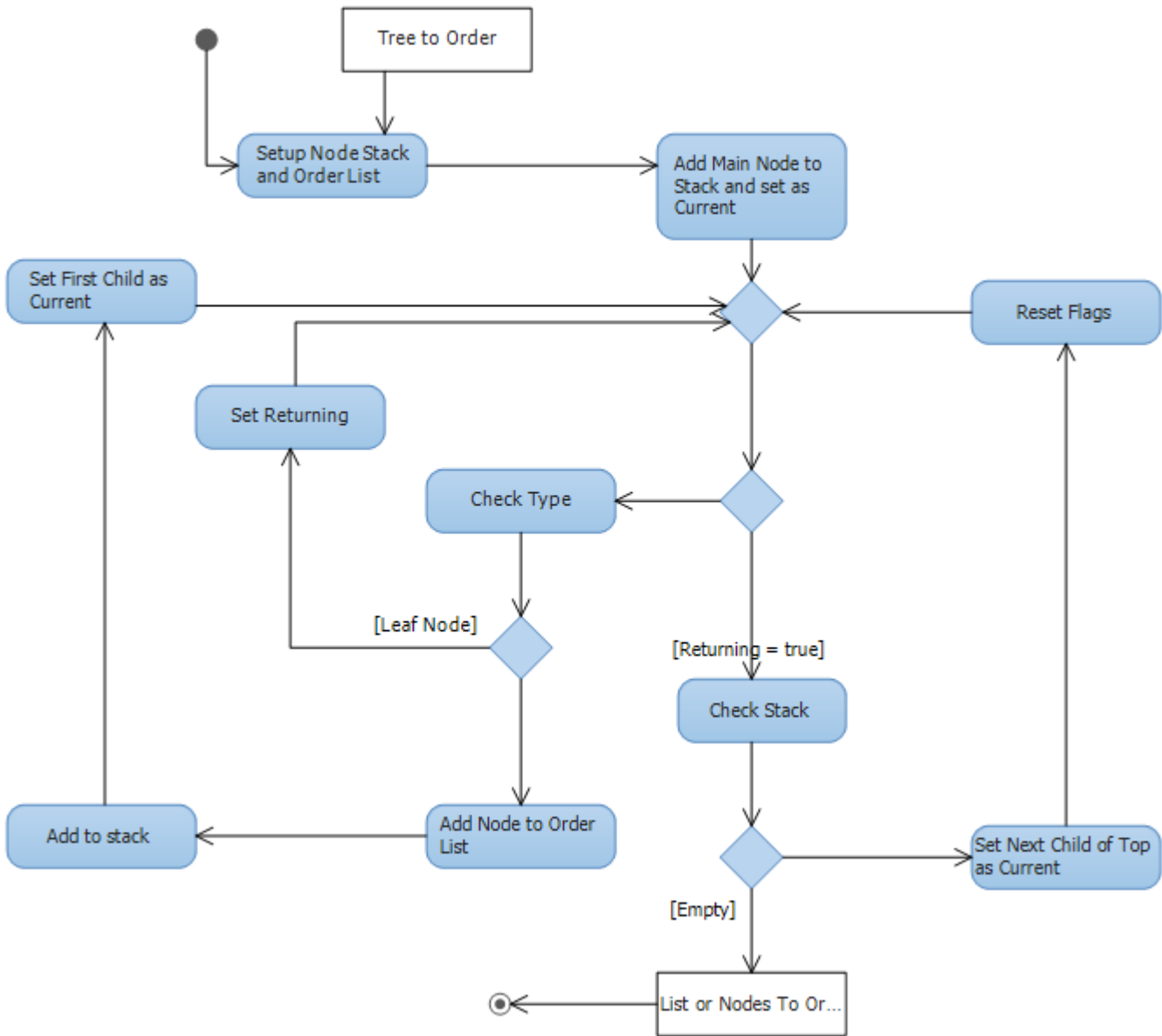


Figure 3.16: Adding Nodes to Ordered List

At the start after it sets up everything it needs, it starts a loop to input all the needed nodes to order into a list.

It sets up a stack to record where it currently is and a list to return containing the nodes to order and a returning flag, then it takes the main node in the tree as the current node to work with, checks whether the returning flag has been triggered, if it hasn't, it checks what type of node it is, and depending on that type it would commit a change.

Mainly, if it is a leaf node, it would trigger the returning flag and start the loop anew. While if it was the other way around, it would add the node to the stack, and to the list. Take the first son and sets it up as the current node. And then repeat the loop.

Once the returning flag is triggered, the stack is checked to see whether it is empty or not.

If it wasn't, it would check which son the top node is currently on, and makes the next son the current node. Resets the returning flag, and loops.

Otherwise it would return that list for the ordering to begin.

Next we go through all the nodes in the list, the program orders the children depending on the type of the node. Since the behavior of the ordering in addition differs from the behavior in multiplication.

In Addition, the children are ordered depending on their types, numbers come first, then variables, then mixed bases.

When two numbers meet, their order doesn't change. When two variables meet, their order depends on who appeared in the program first.

If the two of the same variable meet, then their exponent decides which comes first. if they have the same exponent, they are kept as is.

Mixed bases come after the two, and when compared with other mixed bases, they are compared from left to right, the elements are compared.

In multiplication, the elements are ordered as numbers, then variables.

The variables if the same aren't altered while two different variables are switched based on their appearance in the program.

Once all the elements are setup in an ordered fashion, then the program starts to refine the node depending on its type.

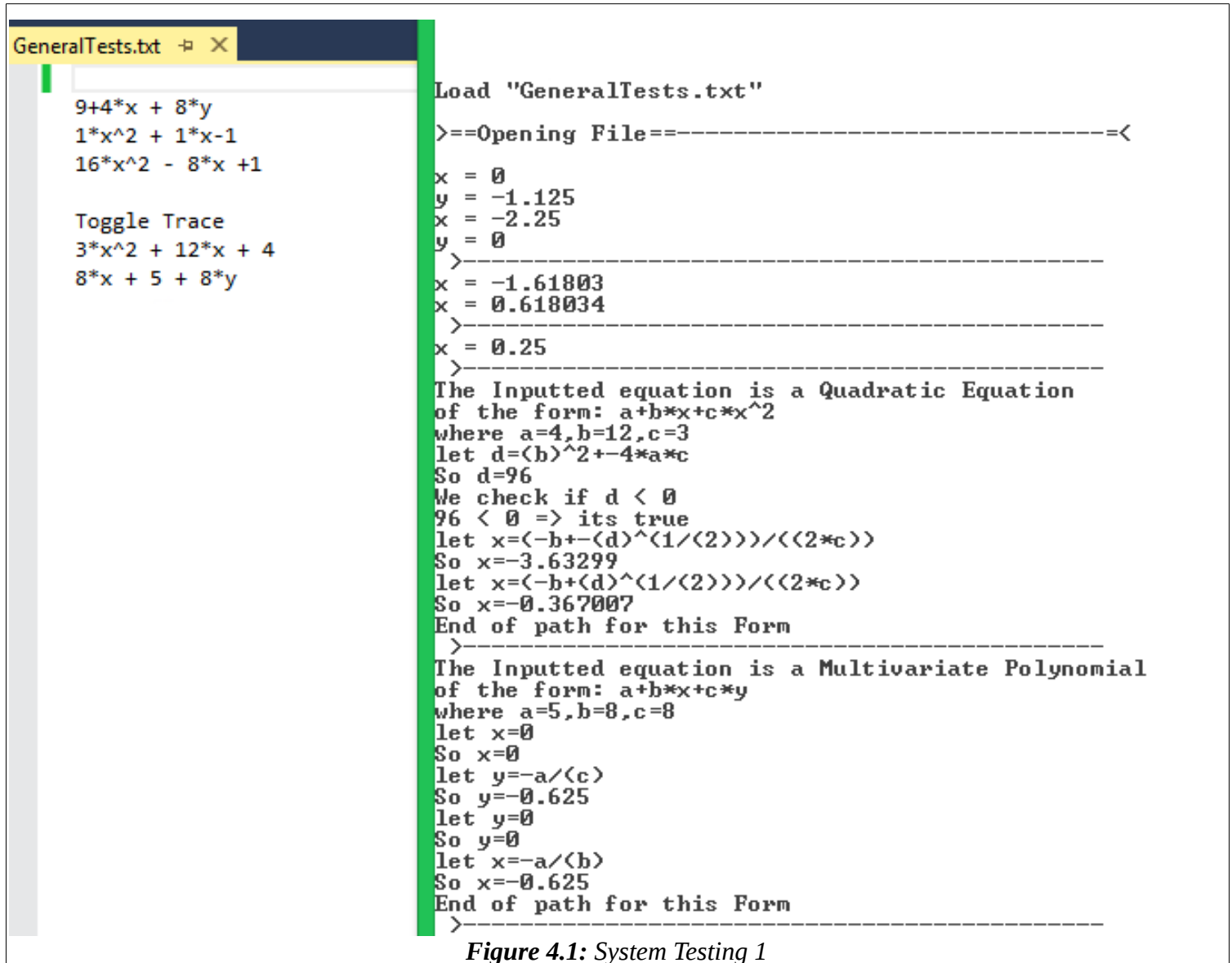
In mathematics, during addition, two blocks consisting of the same base have their coefficient summed, while in multiplication, the same elements have their exponents summed.

And as such our program compares through the children and applies those two rules on them, resulting with a new tree that's been redefined and ready to work with.

Chapter IV : System Testing

In this chapter we will provide some screenshots from the program while it's running in different cases.

Testing *Load* and *Toggle* commands



The screenshot displays a program window titled 'GeneralTests.txt'. The left pane shows the test cases: $9+4*x + 8*y$, $1*x^2 + 1*x-1$, $16*x^2 - 8*x + 1$, 'Toggle Trace', $3*x^2 + 12*x + 4$, and $8*x + 5 + 8*y$. The right pane shows the execution output, which includes file loading, variable assignments, and detailed calculations for quadratic and multivariate polynomial equations.

```
Load "GeneralTests.txt"
>==Opening File==-----=<
x = 0
y = -1.125
x = -2.25
y = 0
-----
x = -1.61803
x = 0.618034
-----
x = 0.25
-----
The Inputted equation is a Quadratic Equation
of the form: a+b*x+c*x^2
where a=4,b=12,c=3
let d=(b)^2+-4*a*c
So d=96
We check if d < 0
96 < 0 => its true
let x=(-b+-(d)^(1/(2)))/((2*c))
So x=-3.63299
let x=(-b+(d)^(1/(2)))/((2*c))
So x=-0.367007
End of path for this Form
-----
The Inputted equation is a Multivariate Polynomial
of the form: a+b*x+c*y
where a=5,b=8,c=8
let x=0
So x=0
let y=-a/(c)
So y=-0.625
let y=0
So y=0
let x=-a/(b)
So x=-0.625
End of path for this Form
-----
```

Figure 4.1: System Testing 1

Testing adding Forms (*If command – End command – Form command*) :

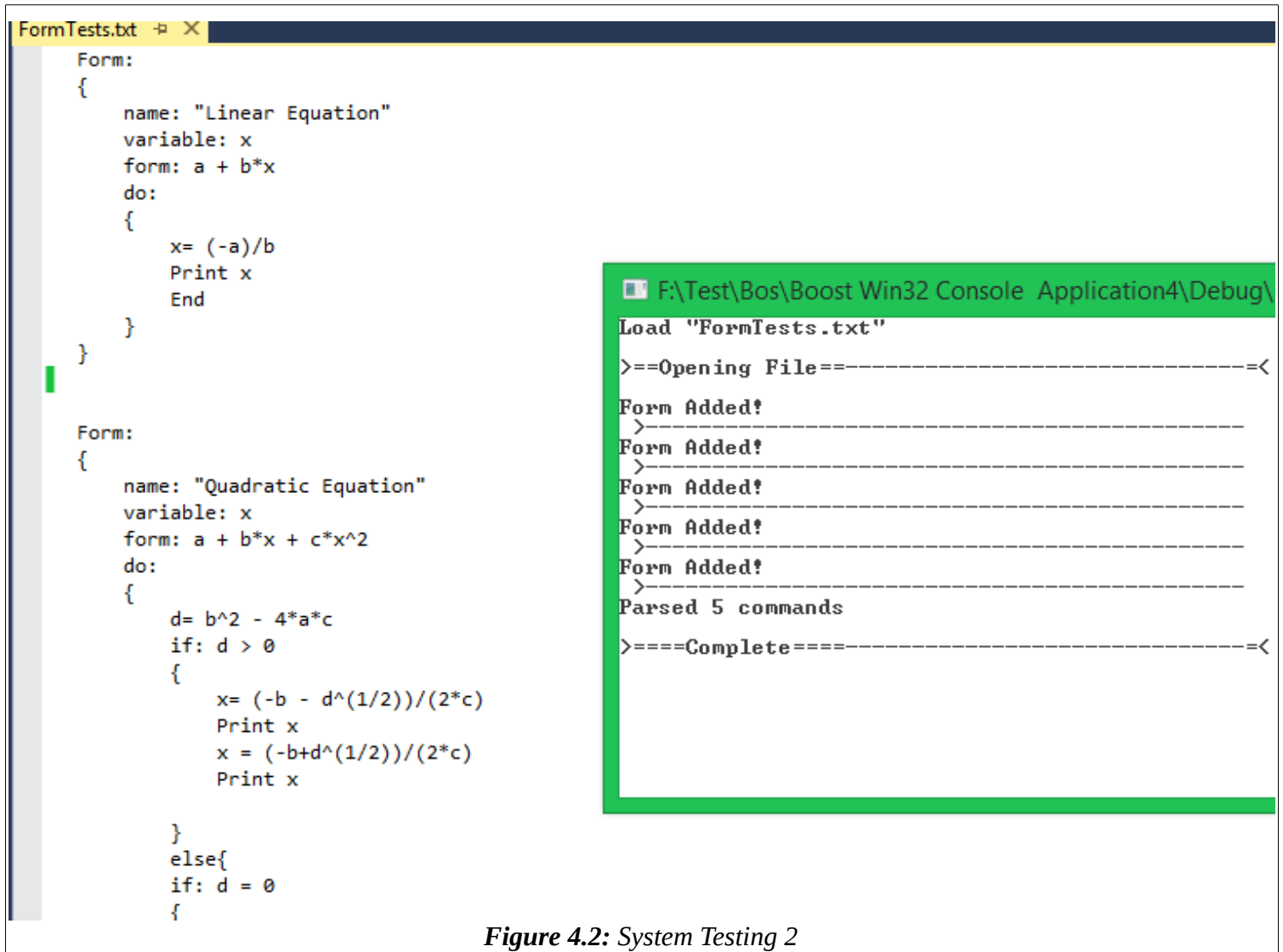


Figure 4.2: System Testing 2

Conclusion and Future Visions

➤ *Conclusion*

At the end, this program can solve polynomial mathematical equations from variant degrees, it can show you the steps taken to achieve the result for pedagogical and academic purposes.

This program strength points that we can input the equations in the natural mathematical form also it can learn new ways from users to use it later.

As you see, we have a great idea alongside huge ambitions.

We thank everyone who stood by us to help our vision to come to life and bare a sapling that no doubt will grow into a beautiful tree.

➤ *Future Visions*

As we now have a stable core to build up on, we have set up great plans for the future of the product.

- Our foremost important goal is to include support for every type of mathematical input out there, as well as the support of any form such as trigonometric functions or complex numbers.
- Alongside our main goal, it is important that we improve the error detection in a way to help the user understand more of what an error could be.
- We plan to spread the support of the program onto multiple platforms, including Android and iOS devices, so that people could enjoy our software everywhere they go, alongside image processing where the user needs only to take a picture of a mathematical problem.
- Not to forget mentioning the improvement to the performance, and memory management or the software.
- We will also improve on the interface, by improving the input to accommodate a style similar to what mathematicians would use in their everyday life, as well as develop an interactive, responsive, and dynamic (GUI) Graphical User Interface that would be able to simplify the many aspects of our software.

References

1. Stroustrup, Bjarne (1997). "1". **The C++ Programming Language** (Third ed.). ISBN 0-201-88954-4.
2. Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D. (1986). **Compilers: Principles, Techniques, and Tools** (1st ed.). Addison-Wesley. ISBN 9780201100884.
3. Allen, Randy; Kennedy, Ken (2001). **Optimizing Compilers for Modern Architectures**. Morgan Kaufmann Publishers. ISBN 1-55860-286-0.
4. Bell, C. Gordon and Newell, Allen (1971), **Computer Structures: Readings and Examples**, McGraw-Hill Book Company, New York. ISBN 0-07-004357-4.
5. Protters & Morrey: "**Calculus and Analytic Geometry. First Course**".
6. P. Aubry, M. Moreno Maza, **Triangular Sets for Solving Polynomial Systems: a Comparative Implementation of Four Methods**. J. Symb. Comput.
7. Songxin Liang, J. Gerhard, D.J. Jeffrey, G. Moroz, **A Package for Solving Parametric Polynomial Systems. Communications in Computer Algebra** (2009).
8. <http://www.boost.org/> (Boost official website)
9. <http://boost-spirit.com/> (Boost Spirit official website)