CMP-6048A/7009A Advanced Programming

Project Report - Due 12 January 2026 before 15:00

Maths Interpreter software

Group members:
Mason Buckle and Anthony De Cruz.

School of Computing Sciences, University of East Anglia

Version 2.0

Abstract

This report documents both the design and implementation of a custom mathematical interpreter and graphing calculator. The primary objective was to o develop a capable maths interpreter and GUI with a user friendly interface. The development methodology adopted a hybrid programming approach, using the functional paradigm of F Sharp for the interpreter logic and the object oriented nature of C Sharp for the user interface.

The project evolved through a series of iterative stages. Early development focuses on creating a foundational GUI, this was expanded to support advanced interpreter features such as operator precedence, floating point arithmetic and variable storage using a symbol table. An external plotting library was used to enable the visualisation of linear and polynomial functions.

The final result is a fully functional mathematical software. It demonstrates successful compatibility between two languages, with the front end handling the user input and the back-end using lexical analysis and evaluation. The inclusion of the optional features: control flow and booleans represents the flexibility of the project.

Chapter 1

Introduction

1.1 Project statement

This project focuses on developing a desktop maths software solution with GUI that uses an interpreter. This project play an important role in areas such as education and research, offering a platform to test mathematical concepts easier. F# is used for the interpreter and C# (WPF) is used for the GUI. The software has been developed over a period of 4 months and was split into sprints (see Development History 3). We accomplished this via modular design and testing each part at every stage. Git was used for version control and Github's kanban board feature was used to break down tasks as well as the platforms workflow feature for automated test pipelines. The final deliverable is a capable desktop maths software solution with a GUI that successfully links a maths interpreter with a user-friendly interface.

1.2 Aims and objectives

The main overarching goal of the project is to develop a capable maths interpreter and GUI with a user friendly interface, this is broken down further into the main project objectives below:

Project Objectives

---

1. To implement a interpreter capable of correctly interpreting and executing arithmetic expressions, managing variable assignment and executing control flow loops.
2. To create a responsive GUI that is capable of accepting user commands, displaying results and displaying errors
3. To create a plotting section in the GUI to plot both linear and polynomial functions and have interactive capabilities such as zooming in and out.

Table 1.1: (Functional) MoSCoW

Priority	Task	Comments
Must	To implement a interpreter capable of correctly parsing and executing arithmetic expressions	Is the most essential task within the brief
	To implement variable management allowing assignment of values to variables and to use variables in expressions	An important feature needed to allow polynomials later in the project
	To develop a basic GUI that has a command prompt for user input and a text field for displaying results or errors	Essential for user interaction and error reporting
	To be able to plot both linear and polynomial functions within the GUI	The main visualization requirement, needed to visualise mathematical functions.
Should	To extend the interpreter with control flow	Implementing for loops
	To implement interactive plotting features E.g. zooming in and out	Enhances the user experience by allowing the user to explore the plane
Could	To visualize the parse tree	Helpful for debugging the parsing logic
	To implement GPU acceleration	To optimise the rendering of the grid line during real-time interaction
Should not	To implement a compiler/transpiler	Overly ambitious given the development time.
	To implement advanced mathematical features (differentiation/integration)	We wanted to ensure the core interpreter was robust and also dropped to the development time.

Chapter 2

Background

The development of maths platforms and plotting software has a lot of history, ranging from early command line maths tools to web based educational tools. Desmos is one of such tools that represents a modern approach to mathematical softwate. Desmos was launched in 2011, it is a web-based graphing calculator built mainly in Javascript [Desmos Studio PBC, 2023]. It mainly focuses on graphing and visualization allowing a user to input complex equations instantly in a browser, furthermore it is a great educational tool allowing students to get instant feedback making it more engaging to learn.

Matlab is a programming and numeric computing platform used to analyze data, develop algorithms and create models [MathWorks ®, 2023]. Matlab was originally released in the late 1970's by Cleve Moler, it was designed to be a interactive matrix calculator [Wikipedia contributors, 2026]. Its launch as a commercial product in 1984 by MathWorks gave matlab a significant redesign transforming it into what it is today allowing not only matrix operations but also plotting capabilities and algorithm implementation.

The project follows some of the methodologies outlined in Crafting Interpreters by Robert Nystrom [Nystrom, 2021]. Nystrom's interpreter is implemented in java using an object oriented approach whilst this project adapts those concepts into a functional paradigm using F#. The Abstract SyntaxTree (AST) was implemented using F# Discriminated Unions. This approach allowed the data structure to mirror our Backus-Naur Form (BNF) directly.

The graphical user interface (GUI) was developed using Window Presentation Foundation (WPF) [Microsoft Learn, WPF, 2023]. The C# frontend invokes the F# lexer and parser functions directly. One of the challenges faced in this was converting functional data types such as F#List into C# arras for the UI logic.

The graphic library Oxyplot was used for the plotting capabilities. Oxyplot is an open source .NET plotting library [OxyPlot, 2023]. It handles the rendering of plot points generated by our interpreter, allowing the frontend To display graphs based on the user input.

Chapter 3

Development History

3.1 Sprint 1: Basic expressions and GUI

This was the starting point of the project, focusing on the development of a Graphic User Interface (GUI) using Window Presentation Format (WPF). A challenge encountered during this phase was the teams unfamiliarity with the WPF framework. This was overcome by consulting the technical documentation [Microsoft Learn, WPF, 2023]. The result was a basic functional frontend capable of accepting basic user input.

3.1.1 Grammar in BNF

```
<E>    ::= <T> <Eopt>
<Eopt> ::= "+" <T> <Eopt> | "-" <T> <Eopt> | <empty>
<T>    ::= <NR> <Topt>
<Topt> ::= "*" <NR> <Topt> | "/" <NR> <Topt> | <empty>
<NR>   ::= "Num" <value> | "(" <E> ")"
```

3.1.2 Basic GUI

We used WPF with C# to develop a basic GUI - see Figure 3.1.

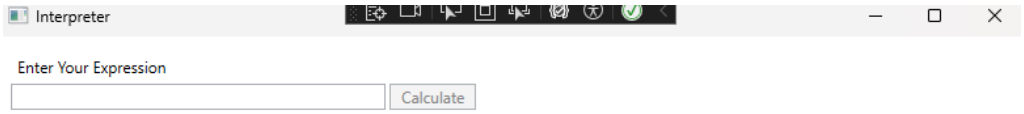

Figure 3.1: A very basic GUI!

3.1.3 Testing

A summary of the critical test cases is presented in Table 3.3. These results represent a subset of the complete test suite found in Table B.1 (Appendix B).

Expression	ResE	ResA	Status
`9 - 3 - 2`	`4`	`4`	Pass
`5 + 3`	`8`	`8`	Pass
`+ 3`	`Error`	`Syntax Error`	Pass

Table 3.1: Key testing outcomes (Subset of full data in Table **??**).

3.2 Sprint 2: Adding unary minus, powers and mod

Building upon the basic arithmetic implemented in Sprint 1, this sprint focused on expanding the interpreter's mathematical capable to support more complex expressions. The main objective was to implement exponents(^), the modulus operator (%) and unary minus. There were two major issues in this sprint which were operator precedence and unary minus representing both subtraction and negation. These were overcome by restructuring the parser to handle operator precedence correctly and changing the parser logic to distinguish between subtraction and negation.

3.2.1 BNF

```
<E>     ::= <T> <Eopt>
<Eopt>  ::= "+" <T> <Eopt> | "-" <T> <Eopt> | <empty>
<T>     ::= <U> <Topt>
<Topt>  ::= "*" <U> <Topt> | "/" <U> <Topt> | "%" <U> <Topt> | <empty>
<U>     ::= "-" <U> | <P>
<P>     ::= <NR> <Popt>
<Popt>  ::= "^" <NR> <Popt> | <empty>
<NR>    ::= "Num" <value> | "(" <E> ")"
```

3.2.2 Updated GUI

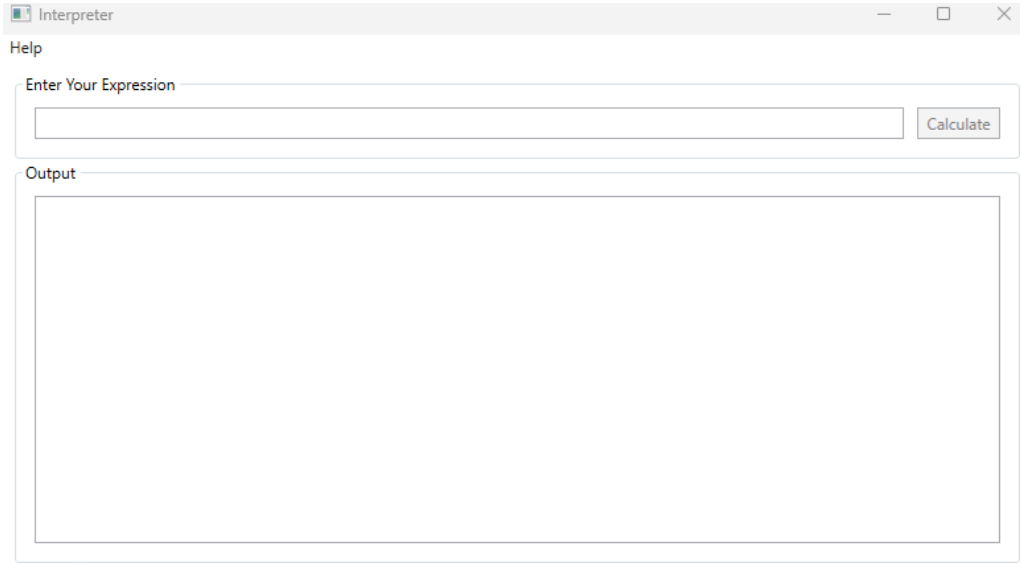

Figure 3.2: Updated GUI

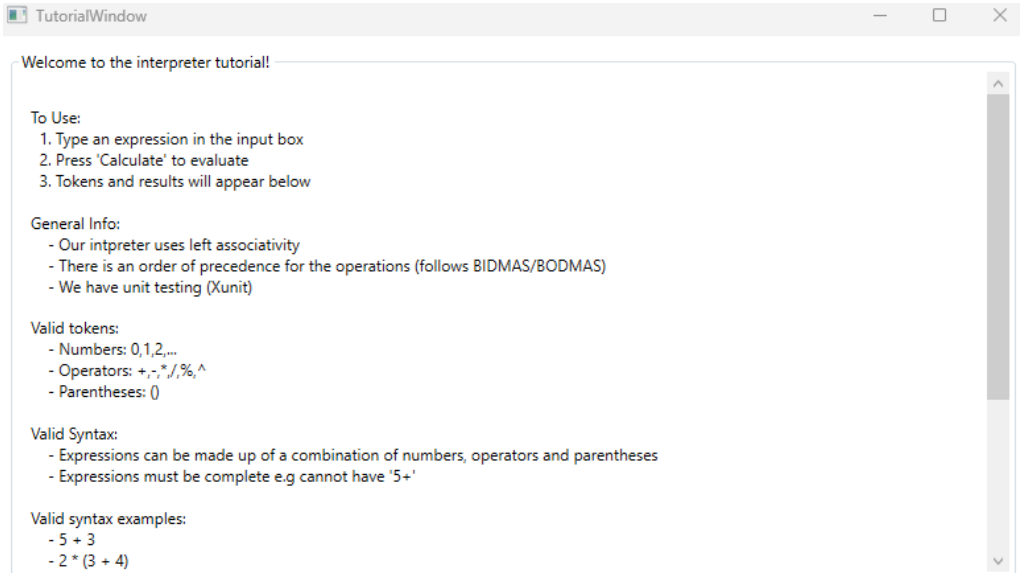

Figure 3.3: Tutorial Page

3.2.3 Testing

A summary of the critical test cases is presented in Table 3.3. These results represent a subset of the complete test suite found in Table **??** (Appendix B).

Expression	ResE	ResA	Status
$10/3$	3	3	Pass
`10 - - 2`	12	12	Pass
$2 * 3^2$	18	18	Pass

Table 3.2: Key testing outcomes (Subset of full data in Table **??**).

3.3 Sprint 3: Added floating point

Sprint 3 addressed the limitation of integer only arithmetic by introducing floating point numbers. This was done by modifying the lexer to recognize decimal points and parse fractional values. There was also an update to the BNF, splitting number into distinct integer (`<IN>`) and Float (`<FL>`) definitions.

3.3.1 BNF

```
<E>    ::= <T> <Eopt>
<Eopt> ::= "+" <T> <Eopt> | "-" <T> <Eopt> | <empty>
<T>    ::= <P> <Topt>
<Topt> ::= "*" <P> <Topt> | "/" <P> <Topt> | "%" <P> <Topt> | <empty>
<P>    ::= <U> <Popt>
<Popt> ::= "^" <U> <Popt> | <empty>
<U>    ::= "-" <U> | <NM>
<NM>   ::= <IN> | <FL> | "(" <E> ")"
<IN>   ::= <digit>+
<FL>   ::= <digit>+ "." <digit>+
```

3.3.2 Testing

A summary of the critical test cases is presented in Table 3.3. These results represent a subset of the complete test suite found in Table **??** (Appendix B).

Expression	ResE	ResA	Status
`10/3.0`	`3.333`	`3.333`	Pass
`3 + 1.1`	`12`	`12`	Pass
$2^{1.1}$	`2.144`	`2.144`	Pass

Table 3.3: Key testing outcomes (Subset of full data in Table **??**).

3.4 Sprint 4: Added linear plotting

Sprint 4 focused on transforming the project into a visual graphic tool. The main objective was the integration with the OxyPlot library [OxyPlot, 2023] into the frontend. The C# was significantly updated to translate constant functions e.g. y=5, this established a good framework for more complex visualisations in later springs.

3.4.1 Updated GUI

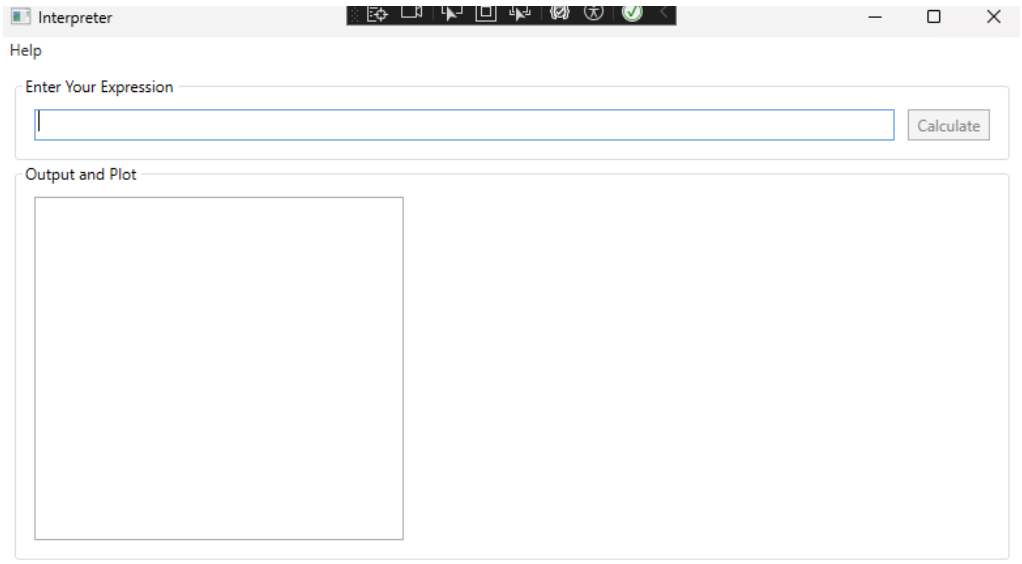

Figure 3.4: Plot GUI

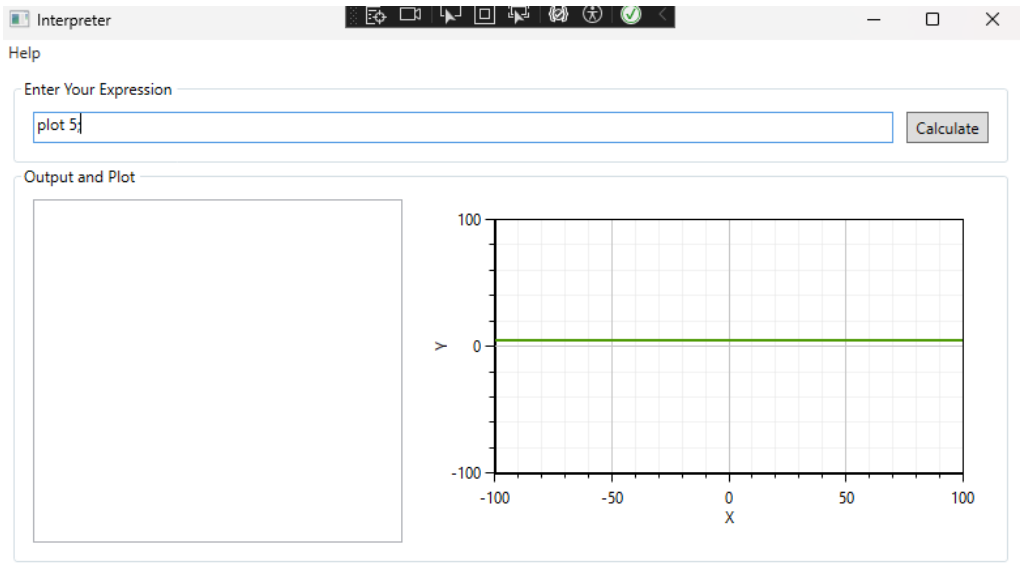

Figure 3.5: Plot GUI with test plot

3.5 Sprint 5: Added polynomial plotting

The focus of sprint 5 was to extend the interpreter's capabilities to support variables and polynomial functions. To achieve this a symbol table was integrated.

3.5.1 BNF

```
STATEMENTS
<STA>    ::= ( <ASN> | <PLT> | <PTR> ) ";"
           | <STA> <STA>
            | <empty>
<ASN>    ::= "let" <SYM> "=" <NM>
<PLT>    ::= "plot" <NM> <PLTopt>
<PLTopt> ::= "," <NM> <PLopt> | <empty>
<PRT>    ::= "print" <NM>
<SYM>    ::= <alpha+>

EXPRESSIONS
<E>    ::= <T> <Eopt>
<Eopt> ::= "+" <T> <Eopt> | "-" <T> <Eopt> | <empty>
<T>    ::= <P> <Topt>
<Topt> ::= "*" <P> <Topt> | "/" <P> <Topt> | "%" <P> <Topt> | <empty>
<P>    ::= <U> <Popt>
<Popt> ::= "^" <U> <Popt> | <empty>
<U>    ::= "-" <U> | <NM>
<NM>   ::= <IN> | <FL> | <SYM> | "(" <E> ")"
<IN>   ::= <digit+>
<FL>   ::= <digit+> "." <digit+>
```

3.5.2 Updated GUI

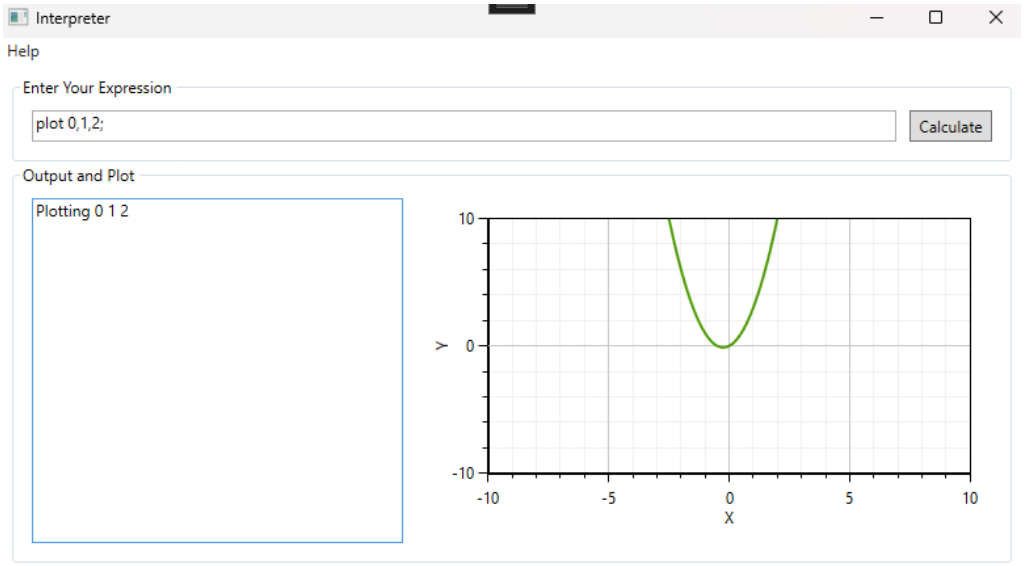

Figure 3.6: Plot GUI with a parabola

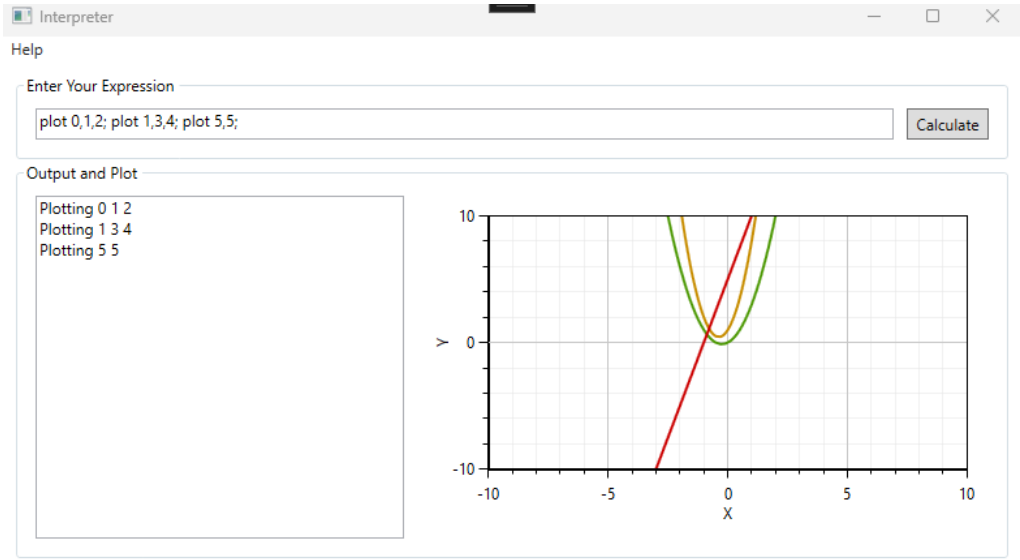

Figure 3.7: Plot GUI with multiple lines and parabola

Chapter 4

Final deliverable

The chapter will cover the final deliverable version of the project, including BNF and code architecture.

4.1 Final BNF

```
STATEMENTS
<PROG>  ::= <STA> <PROG> | <empty>
<STA>   ::= <WHL> | <IF> | <ASN> | <PLT> | <PRT>
<WHL>   ::= "while" <BE> "{" <PROG> "}"
<IF>    ::= "if" <BE> "{" <PROG> "}"
<ASN>   ::= "let" <SYM> "=" <BE> ";" | "func" <SYM> "=" <BE> ";"
<PLT>   ::= "plot" <BE> ";"
<PRT>   ::= "print" <BE> ";" | <BE> ";"    // Print top level expressions.
<SYM>   ::= <alpha+>

EXPRESSIONS
<BE>    ::= <BU> <BEopt>
<BEopt> ::= "and" <BU> <BEopt> | "or" <BU> <BEopt> | <empty>
<BU>    ::= "!" <BU> | <BT>
<BT>    ::= <E> <BTopt>
<BTopt> ::= "==" <E> <BTopt>
          | "!=" <E> <BTopt>
          | ">" <E> <BTopt>
          | "<" <E> <BTopt>
          | <empty>
<E>     ::= <T> <Eopt>
<Eopt>  ::= "+" <T> <Eopt> | "-" <T> <Eopt> | <empty>
<T>     ::= <P> <Topt>
<Topt>  ::= "*" <P> <Topt> | "/" <P> <Topt> | "%" <P> <Topt> | <empty>
<P>     ::= <U> <Popt>
<Popt>  ::= "^" <U> <Popt> | <empty>
<U>     ::= "-" <U> | <NM>
<NM>    ::= <VL> | <SYM> | "(" <BE> ")"    // Where SYM is defined in symbol table.
<VL>    ::= <IN> | <FL>    // Separate literal value simplifies implementation.
<IN>    ::= <digit+>
<FL>    ::= <digit+> "." <digit+>

KEY
STATEMENTS
PROG  -> Program
STA   -> Statement
```

```
ASN   -> Variable/Function Assignment
PLT   -> Plot
PTR   -> Print
SYM   -> Symbol
EXPRESSIONS
BE    -> Boolean Expression
BEopt -> Boolean Expression/Optional
BU    -> Boolean Unary
BT    -> Boolean Term
E     -> Expression
Eopt  -> Expression/Optional
T     -> Term
Topt  -> Term/Optional
P     -> Power
Popt  -> Power/Optional
U     -> Unary
NM    -> Number
VL    -> Value
IN    -> Integer
FL    -> Floating Point
```

This complete BNF models our final interpreter's capabilities. The top of the structure includes statements which are designed to run sequentially with the recursive $PROG$ type. Statements are broken down into several types where each one is denoted with a string keyword and at least one argument.

Loops WHL allow the user to execute $PROG$ blocks repeatedly as long as a condition BE is met. Ifs IF follow effectively the same behaviour without the repetition. A condition is met if the expression resolves as any non zero value. The language lacks a dedicated boolean type for simplicity.

Assignments ASN allow variables to be set with $let =$ and functions (effectively expressions that are only fully evaluated at plot time) to be set with $func =$.

Plot PLT is a special statement that allows the user to provide an expression to be used to generate a plot sequence. During execution, the given expression is resolved for each step in the plotting sequence. For each step, the symbol table is modified such that the special variable x is set to the plot point X, where Y is the result of the expression. The implementation code can be found in Fig.C.1. When a plot is made, a message is written to the output text stream.

The print statement PRT writes the given expression result to the output text stream.

After statements come expressions. The expression hierarchy is structured in such an order as to facilitate the desirable order of execution in a recursive descent parser. Boolean expressions all resolve to return either an integer or float value 1 or 0, depending on the logical result. Boolean expressions BE *and* and *or* are resolved after boolean unary BU ! and boolean terms BT ==, > in order to get the conventionally expected behaviour from a condition such as 5 != 3 and 2 == 1 - 3 found in other languages.

After boolean expressions come the rest of the arithmetic expressions. In order to produce mathematically correct operator precedence, unary minuses U are resolved last, followed by powers P, terms T which includes $*$, /, and % whilst expressions E with + and − are resolved last.

Finally numbers NM contain our literal values VL, symbol SYM strings and bracketed BE. Integers IN and floats FL are wrapped by their own type for the sake of ease of implementation. SYM strings are used as variable and function identifies to lookup and write to the symbol table. BE are resolved before other kinds of expressions here in order to set appropriate bracketed precedence.

4.2 Final GUI

Fig.4.1 shows a screenshot of the final application running a simple program, plotting several lines. The program being run demonstrates the full capabilities of the final interpreter, demonstrating variables, functions, loops, conditionals, arithmetic and plotting.

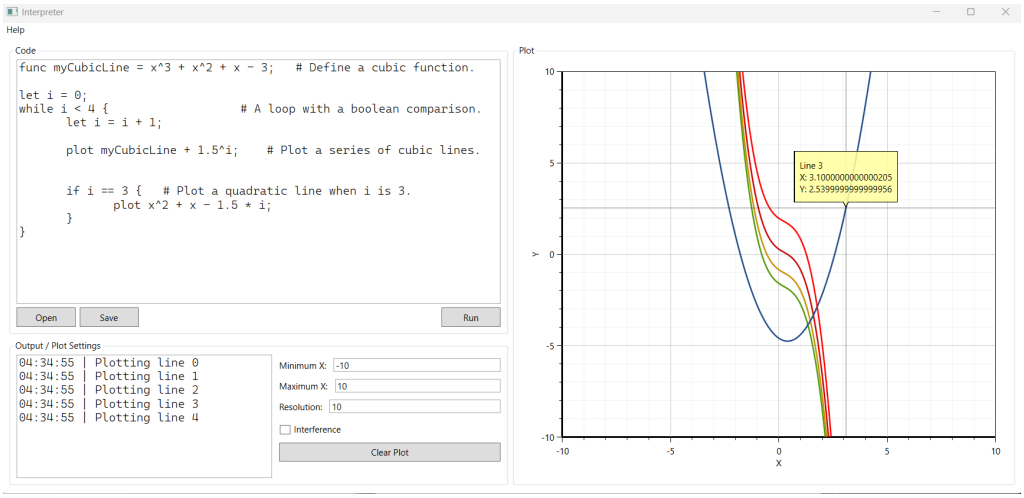

Figure 4.1: The final application.

Fig.4.2 shows a screenshot of the final application running the same program in interference mode. This was a mode of rendering added to facilitate the generate of interference patterns based on the combination of every wave or line being plotted at once. Although not directly described in the brief, this feature was added at the end as the development cost was considered minimal. This style of rendering can be toggled via a simple labelled checkbox. This capability would have significant use cases in the fields of wave processing and communications.

4.3 Code architecture

The F# interpreter is implemented as a recursive descent parser which is covered in further detail in the following section. BNF specific nodes are covered in the previous section. The parser is implemented as a function that returns an AST made up of nodes defined as type unions. This AST is then passed into the executor which then traverses the AST to resolve the program.

Fig.4.3 shows a UML class diagram of the GUI. In order to keep the front end as minimal as possible, it consists only of 2 window classes and 2 wrapper classes for the F# interpreter and plot modelling, named *InterpreterController* and *PlotController* respectively. The application was architected with an MVC style model was attempted in order to follow software engineering best practices, forming abstract interfaces that encapsulate each area of responsibility and facilitate smooth automated testing.

InterpreterController worked as the interface between the F# interpreter and the C# frontend, cleaning abstracting away any language interop difficulties. Plot data rendering and modelling was provided by Oxyplot's *PlotModel* and *LineSeries* types.

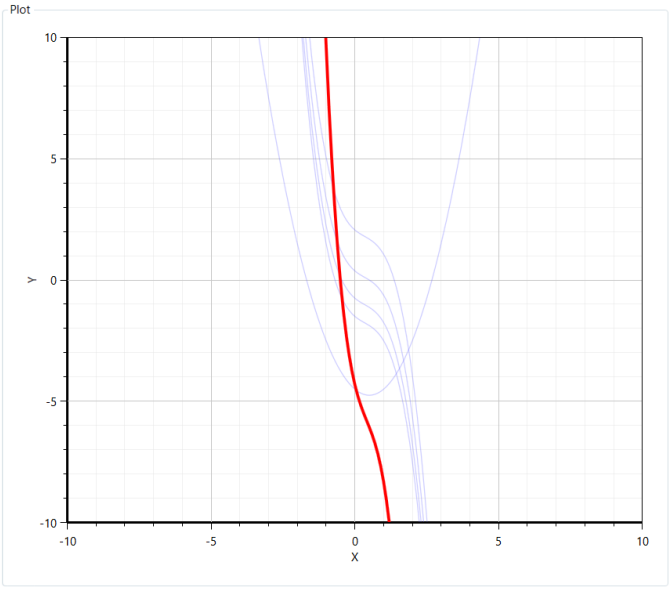

Figure 4.2: The final application. The program from fig.4.1 rendering in interference mode.

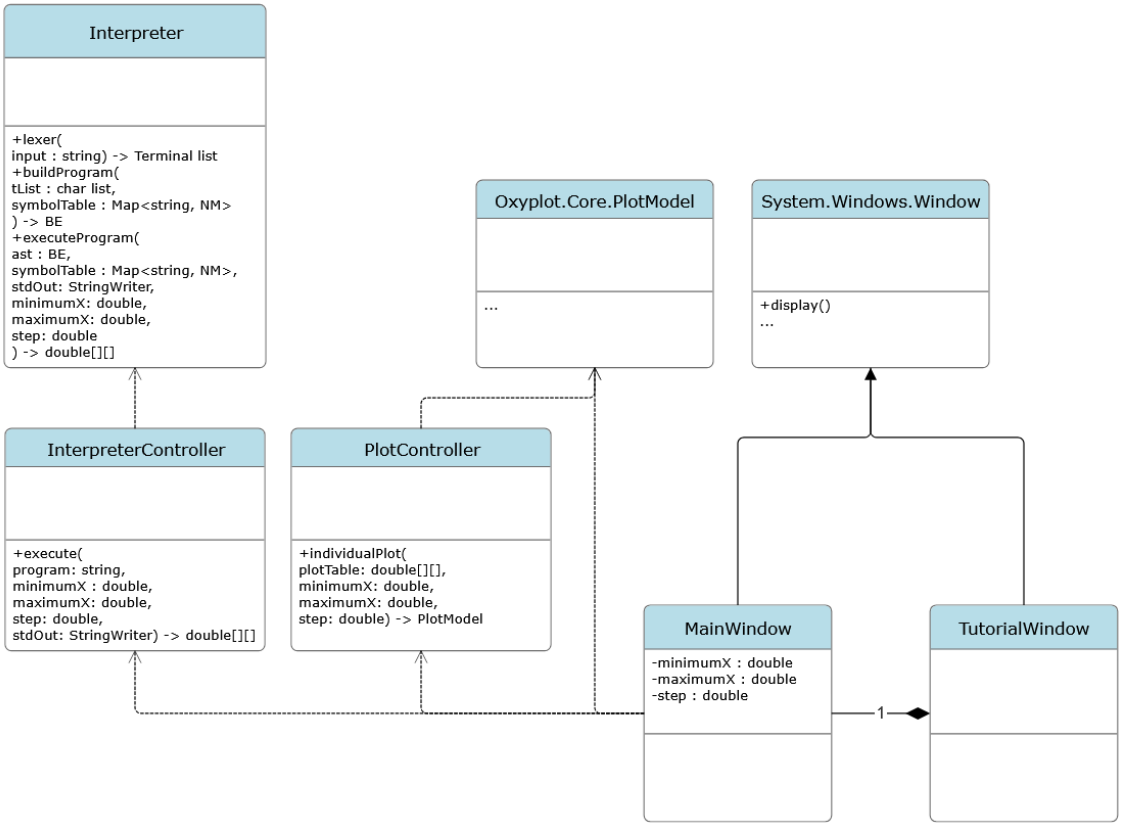

Figure 4.3: The GUI UML class diagram. Note that *System.Windows.Window* is provided by WPF and *Oxyplot.Core.PlotModel* is provided by Oxyplot.

4.4 Algorithms

The lexer in Fig.4.4 processes the source input in a single left-to-right pass, repeatedly examining the next character (or sequence of characters) to determine the appropriate token to generate. Keywords and operators are detected first using fixed string matches to ensure correct precedence over identifiers. Numbers are recognised by scanning consecutive digits into a single value, while identifiers are formed by consuming contiguous alphabetic

characters. For each construct, a corresponding token is appended to the output list. This is a simple and efficient lexer for use with a recursive descent parser.

Algorithm 1 Lexer algorithm

```
1:  tokens ← [ ]
2:  input ← ToCharList(source)
3:  while input is not empty do
4:      c ← Head(input)
5:      if input starts with a keyword or operator then
6:          tokens.Append(Token)
7:          Consume(n)
8:      else if c is a digit then
9:          (value, rest) ← ScanNumber(input)
10:         tokens.Append(value)
11:         input ← rest
12:     else if c is alphabetic then
13:         (name, rest) ← ScanIdentifier(input)
14:         tokens.Append(name)
15:         input ← rest
16:     else if c = # then
17:         input ← SkipComment(input)
18:     else if c is whitespace then
19:         Consume(1)
20:     else
21:         RaiseSyntaxError(c)
22:     end if
23: end while
24: return tokens
```

Figure 4.4: Simplified lexer algorithm for tokenising source input

The pseudocode found in Fig.4.5 is designed to demonstrate the logical flow of a recursive descent parser. Here, we can see the function *ParseExpression* looks to parse a single expression node, calling *ParseNextExpression* on it's left hand side and *ParseOperand* on it's right hand side first before creating it's own node. This means that when an expression is first parsed, the left is always resolved first, trickling down to it's terminal literal. The right hand side is then parsed. If there is a matching operator, the right hand operand is then evaluated and the cycle continues. If there is no matching operator, the algorithm will continue to bubble up until one is found or the program ends.

This method of traversal depicts both token parsing for building an AST as well as AST traversal for execution and is how both passes of the interpreter are implemented.

Algorithm 2 Recursive descent parser demonstrating recursion and precedence

```
1: function ParseExpression(tokens)
2:     (left, tokens) ← ParseNextExpression(tokens)
3:     (right, tokens) ← ParseOperand(tokens)
4:     return (NewNode(left, right), tokens)
5:
6: function ParseOperand(tokens)
7: if next token is specific operator then
8:         (left, tokens) ← ParseNextExpression(tokens)
9:         (right, tokens) ← ParseOperand(tokens)
10:        return (SpecificNode(, expr), tokens)
11: else
12:        return (EmptyOperand(), tokens)
13: end if
14:
15: function ParseNextExpression(tokens) ...
```

Figure 4.5: Recursive descent parser demonstrating recursion and precedence.

Fig.4.6 Demonstrates how plots data is generated. In the actual implementation, expressions are stored in the interpreter symbol table during parsing and are resolved for different values of x at execution time. Values for x are inserted as values within the symbol table in each iteration. In order to facilitate multiple lines, the real implementation features a 2 dimensional array of plot points, where each sequence represents a single line to be plotted.

Algorithm 3 Generate plot points from expression

```
1: f ← Expression
2: minX ← −10.0
3: maxX ← 10.0
4: step ← 0.1
5: x ← maxX
6: while x > minX do
7:     x ← x − step
8:     y ← Compute(f(x))
9:     AppendResult(x, y)
10: end while
```

Figure 4.6: Algorithm to generate plot points. $minX$, $maxX$, and $step$ are set as example values.

Fig.4.7 Demonstrates how a new interference pattern can be generate from a collection of line plot sequences in a simple manner.

Algorithm 4 Generate interference pattern based on existing plots

```
lines ← ExistingPlotSequences
pattern ← NewPlotSequence
for each index i of lines[0] do
    x ← lines[0][i].x
    y ← 0
    for each line in lines do
        y ← y + line[i].y
    end for
    pattern.AppendResult(x, y)
end for
```

Figure 4.7: Algorithm to generate interference patterns.

4.4.1 Testing

Extensive testing as been performed for all aspects of the system including the interpreter, GUI and plotting functionality.

The interpreter was validated through comprehensive unit testing. Both valid and inputs with specific focus on syntactic correctness and adherence to the grammar BNF. Test cases were designed to assess operator precedence and associativity as well as syntactic errors. An automated build and test pipeline was implemented for project pull requests. This pipeline help to detect regressions early and keep development fluid. Tables B.1, B.2, B.3, B.4 and B.5 contain the automated unit test cases.

Structured manual testing of the GUI was performed in order to determine functionality. Core UI functions were run and outputs checked against expected results. WPF based GUI testing solutions such as [WPF Pilot, 2026], which allows you to inspect specific UI elements, were considered, as additional automated testing would make it easier and more practical to programmatically test the GUI more exhaustively. We decided against integrating these types of tools as the time cost could not be justified as the GUI is too simple. A similar manner was also used to determine plotting correctness where we plotted a known function and manually calculated the correctness of Y values at a series of known points of X. Tables B.6 and B.7 contain the manual test cases.

This overall testing methodology was informed by time cost to benefit analysis and previous industry experience. The automated test pipeline proved invaluable as it caught a number of regression issues throughout the development process.

Chapter 5

Discussion, conclusion and future work

Briefly discuss your achievements and put them in perspective with the MoSCoW analysis you specified in Table 1.1. Also discuss future developments and how you see the deliverable improving if more time could be spent. Note that this section should not be used as a medium to vent frustrations on whatever did not work out (group issues, not enough time, illness, etc.) as this should be dealt with separately - keep it professional!

Bibliography

[Desmos Studio PBC, 2023] Desmos Studio PBC (2023). Desmos website. https://desmos.com [Accessed: 30/11/2023].

[MathWorks ®, 2023] MathWorks ® (2023). Matlab website. https://uk.mathworks.com/products/matlab.html [Accessed: 30/11/2023].

[Microsoft Learn, WPF, 2023] Microsoft Learn, WPF (2023). *Windows Presentation Foundation documentation*. Microsoft.

[Nystrom, 2021] Nystrom, R. (2021). *Crafting Interpreters*. Genever Benning, Great Brittain.

[OxyPlot, 2023] OxyPlot (2023). Oxyplot: A cross-platform plotting library for .net. [https://oxyplot.github.io/] [Accessed: 2023-11-30].

[Wikipedia contributors, 2026] Wikipedia contributors (2026). Matlab - wikipedia. https://en.wikipedia.org/wiki/MATLAB.

[WPF Pilot, 2026] WPF Pilot (2026). Wpf pilot: A wpf testing library. [https://wpfpilot.dev/] [Accessed: 2026-01-11].

Appendix A

Contributions

A.1 Individual Contributions

Name	Contribution
Anthony de Cruz	50%
Mason Buckle	50%

Both members equally contributed to the project. We both worked on designing the application/BNF in equal time as well as performing testing of the application. Anthony developed most of the F# interpreter and Mason developed the plotting and GUI in C#. This complete report was also written in equal parts by both members with Mason writing chapters 1, 2 and 3 and Anthony writing chapters 4 and 5.

Appendix B

Testing

B.1 Arithmetic expression testing

Table B.1: Original set of arithmetic expression tests. Note that floating pointing values are accurate to three decimal places for the fractional part. ResE is expected result and ResA is actual result.

Expression	ResE	ResA	Pass/Fail	Action/comment
$5*3+(2*3-2)/2+6$	23	23	PASS	...
$9-3-2$	4	4	PASS	left assoc.
$10/3$	3	3	PASS	int division
$10/3.0$	3.333	3.333	PASS	float division
$10\%3$	1	1	PASS	
$10--2$	12	12	PASS	unary minus
$-2+10$	8	8	PASS	
$3*5\hat{}(-1+3)-2\hat{}2*-3$	87	87	PASS	power test
$-3\hat{}2$	-9(*) or 9	9	PASS	precedence
$-7\%3$	2(*) or -1	-1	PASS	precedence (*)Python
$2*3^2$	18	18	PASS	precedence pow ¿ mult
$3*5\hat{}(-1+3)-2\hat{}-2*-3$	75.750 or 75	75	PASS	
$3*5\hat{}(-1+3)-2.0\hat{}-2*-3$	75.750	75.750	PASS	
$(((3*2--2)))$	8	8	PASS	
$(((3*2--2))$	Error	Syntax Error	PASS	syntax error
$-((3*5-2*3))$	-9	-9	PASS	minus expression
$x=3;(2*x)-x\hat{}2*5$	-39	39	PASS	var assign
$x=3;(2*x)-x\hat{}2*5/2$	-16	-16	PASS	
$x=3;(2*x)-x\hat{}2*(5/2)$	-12	-12	PASS	
$x=3;(2*x)-x\hat{}2*5/2.0$	-16.5	-16.5	PASS	
$x=3;(2*x)-x\hat{}2*5\%2$	5	5	PASS	
$x=3;(2*x)-x\hat{}2*(5\%2)$	-3	-3	PASS	

Table B.2: Boolean expression & statement tests.

Expression	ResE	ResA	Pass/Fail	Action/comment
5 > 3	1	1	PASS	
3 > 5	0	0	PASS	
3 < 5	1	1	PASS	
5 < 3	0	0	PASS	
2 == 2	1	1	PASS	
2 == 1	0	0	PASS	
2! = 2	0	0	PASS	
2! = 1	1	1	PASS	
!1	0	0	PASS	
!0	1	1	PASS	
1 and 1	1	1	PASS	
0 and 1	0	0	PASS	
1 and 0	0	0	PASS	
0 and 0	0	0	PASS	
5 > 2 and 10 == 4	0	0	PASS	
5 > 2 and 10! = 4	1	1	PASS	
1 or 1	1	1	PASS	
0 or 1	1	1	PASS	
1 or 0	1	1	PASS	
0 or 0	0	0	PASS	
5 > 2 or 10 == 4	1	1	PASS	
5 > 2 or 10! = 4	1	1	PASS	
>	Error	Syntax Error	PASS	
< 3	Error	Syntax Error	PASS	
!	Error	Syntax Error	PASS	
3 == 5! =	Error	Syntax Error	PASS	
i = 0; while i ¡ 5 { i = i + 1; } y = i;	5	5	PASS	
i = 7; if i == 7 { y = 8; }	8	8	PASS	
i = 0; while i ¡ 5 y = i;	Error	Syntax Error	PASS	
i = 0; if i ¡ 5 y = i;	Error	Syntax Error	PASS	
i = 0; while { y = i; }	Error	Syntax Error	PASS	
i = 0; if { y = i; }	Error	Syntax Error	PASS	

Table B.3: Expression tests.

Expression	ResE	ResA	Pass/Fail	Action/comment
$5 + 3$	8	8	PASS	
$200 + 13 + 45$	258	258	PASS	
$3 + 1.1$	4.1	4.1	PASS	
$5 + 3$	8	8	PASS	
$+$	Error	Syntax Error	PASS	
$+3$	Error	Syntax Error	PASS	
$3+$	Error	Syntax Error	PASS	
$3 + 5+$	Error	Syntax Error	PASS	
$3 * 3$	9	9	PASS	
$8 * 4 * 3$	96	96	PASS	
$3 * 1.1$	3.3	3.3	PASS	
$3.256 * 1.59$	5.177	5.177	PASS	
$*$	Error	Syntax Error	PASS	
$*3$	Error	Syntax Error	PASS	
$3*$	Error	Syntax Error	PASS	
$3 * 5*$	Error	Syntax Error	PASS	
$6/3$	2	2	PASS	
$5/3.0$	1.667	1.667	PASS	
$12/3/2$	2	2	PASS	
$3.2/2$	1.6	1.6	PASS	
$3.4/2.3$	1.478	1.478	PASS	
$/$	Error	Syntax Error	PASS	
$/3$	Error	Syntax Error	PASS	
$3/$	Error	Syntax Error	PASS	
$3/5/$	Error	Syntax Error	PASS	
$3/0$	Error	Divide By Zero	PASS	
$3/0.0$	Error	Divide By Zero	PASS	

Table B.4: Expression tests continued.

Expression	ResE	ResA	Pass/Fail	Action/comment
6%3	0	0	PASS	
5%3	2	2	PASS	
19%5%3	1	1	PASS	
3.2%2	1.2	1.2	PASS	
3.4%2.3	1.1	1.1	PASS	
%	Error	Syntax Error	PASS	
%3	Error	Syntax Error	PASS	
3%	Error	Syntax Error	PASS	
3%5%	Error	Syntax Error	PASS	
3%0	Error	Divide By Zero	PASS	
3%0.0	Error	Divide By Zero	PASS	
5^2	25	25	PASS	
5^2^2	625	625	PASS	
2^1.1	2.144	2.144	PASS	
^	Error	Syntax Error	PASS	
^3	Error	Syntax Error	PASS	
3^	Error	Syntax Error	PASS	
3^5^	Error	Syntax Error	PASS	
-2	-2	-2	PASS	
$5--2$	7	7	PASS	
$5+-2$	3	3	PASS	
$5+--3$	8	8	PASS	
$2^{\wedge}-2$	0	0	PASS	
$2^{\wedge}--2$	4	4	PASS	
$-$	Error	Syntax Error	PASS	
$5++-3$	Error	Syntax Error	PASS	
$2^{\wedge}-$	Error	Syntax Error	PASS	
$2--+--2$	Error	Syntax Error	PASS	

Table B.5: Lexer tests for operators, floating point numbers, and symbols.

Expression	ResE	ResA	Pass/Fail	Action/comment
3 + 5	[Int 3; Add; Int 5]	[Int 3; Add; Int 5]	PASS	
1 * 2	[Int 1; Mul; Int 2]	[Int 1; Mul; Int 2]	PASS	
3 ^ 8	[Int 3; Pwr; Int 8]	[Int 3; Pwr; Int 8]	PASS	
3 £ 4	Error	Syntax Error	PASS	
5 5 :	Error	Syntax Error	PASS	
3.8	[Flt 3.8]	[Flt 3.8]	PASS	
3.008	[Flt 3.008]	[Flt 3.008]	PASS	
3.811	[Flt 3.811]	[Flt 3.811]	PASS	
0.811	[Flt 0.811]	[Flt 0.811]	PASS	
100.001	[Flt 100.001]	[Flt 100.001]	PASS	
7.	Error	Syntax Error	PASS	
.7	Error	Syntax Error	PASS	
5 .2	Error	Syntax Error	PASS	
2. 5	Error	Syntax Error	PASS	
x	[Sym "x"]	[Sym "x"]	PASS	
y	[Sym "y"]	[Sym "y"]	PASS	
varname	[Sym "varname"]	[Sym "varname"]	PASS	
myvar	[Sym "myvar"]	[Sym "myvar"]	PASS	
3 + variable	[Int 3; Add; SymT "variable"]	[Int 3; Add; Sym "variable"]	PASS	

B.2 GUI testing

Table B.6: A series of manual/visual tests to determine functionality.

Action	ResE	ResA	Pass/Fail
Write code into main text field and hit the run button.	The program is executed, with expected plots and output.	The program is executed as expected.	PASS
Press the clear button.	Any existing plots should be cleared.	Existing plots are cleared.	PASS
Press the open button.	A dialogue is shown allowing the user to select a text file to load into the buffer.	A file is loaded into the text buffer via a dialogue.	PASS
Press the save button.	The existing text buffer should be saved to a text file via a dialogue.	The current text buffer is written to a text file.	PASS
Press the help button.	The tutorial window should be displayed.	The tutorial window is displayed.	PASS

B.3 Plot testing

Table B.7: A Series of manual/visual tests to determine functionality.

Action	ResE	ResA	Pass/Fail
Execute "plot x + 2;" and set minimum X = -10, maximum X = 10, resolution = 10. Observe the plotted values in the GUI.	Where Y = 0, X should be 2. Where Y = 1 X should be 3.	Where Y = 0, X = 2. Where Y = 0, X = 3.	PASS
Execute "plot x^2 - 2;" and set minimum X = -10, maximum X = 10, resolution = 10. Observe the plotted values in the GUI.	Where Y = 0, X = -2. Where Y = 5, X = 23.	Where Y = 0, X = -2. Where Y = 0, X = 23.	PASS

Appendix C

Other Materials

The program found in Fig.3.2.

```
func myCubicLine = x^3 + x^2 + x - 3;   # Define a cubic function.

let i = 0;
while i < 4 {                    # A loop with a boolean comparison.
    let i = i + 1;

    plot myCubicLine + 1.5^i;    # Plot a series of cubic lines.

    if i == 3 {   # Plot a quadratic line when i is 3.
        plot x^2 + x - 1.5 * i;
    }
}
```

```
// Execute plots.
// <PLT>   ::= "plot" <E> ";"
and execPLT (e: PLT) (symbolTable: Map<string, NM>) (plotTable: VL list list) : VL list list =
    let yValues =
        [ for x in plotXmin .. plotXstep .. plotXmax -> // Calculate Y points by injecting an X value for each step.
            let tempTable = Map.add "x" (Val (Flt x)) symbolTable
            evalExpr e tempTable ]
    let timeStr = DateTime.Now.ToString("hh:mm:ss")
    stdOut.WriteLine($"{timeStr} | Plotting line {plotTable.Length}")
    plotTable @ [yValues]
```

Figure C.1: The real F# implementation for plot point calculation.