CMP-6048A Advanced Programming

Project Report - 15 December 2021

Maths Interpreter Software

Group members:
James Mason and Nia Preston

School of Computing Sciences, University of East Anglia

Version 1.0

Abstract

In this project, our aim was to create a maths-based interpreter, capable of a range of useful mathematical operations outlined by the project brief. This includes expression evaluation and function graphing via abstract syntax tree based interpretation. The abstract syntax tree allows the correct precedence when interpreting based on the depth of the node in the tree. Each node within the tree, implements a visitor pattern allowing further abstraction and generalization within the interpreter. We used a scrum based development methodology to develop our language in sprints, allowing us to evolve our program over time with frequent re-adjustment of requirements and functionality.

At the conclusion of our project, we found that we were able to implement all "Must" criteria alongside most of our "Should" criteria. Our program is built in a way that it can be easily extended, but is a stable and practical program in its current state.

Chapter 1

Introduction

1.1 Project statement

The purpose of this project is to create a piece of software that is an interpreter for mathematical functions. It will run both as an interactive prompt and a utility to interpret and execute pre-written files. It will additionally provide the ability to visualize mathematical functions such as on a graph.

1.2 MoSCoW

The following outlines the requirements we aim to meet with this software.

1.2.1 Musts

- Expressions for basic arithmetic
- Print Statements
- Other Commands
- Custom syntax using a custom lexer and parser

1.2.2 Shoulds

- Expressions for variable assignment
- Exponential operator expression
- Conditional Statements
- Function Declaration
- Mathematical function visualization
- Flexible and scalable architecture
- Simple trigonometric functions
- Zero-crossings finder

1.2.3 Coulds

- Intuitive GUI
- Interactive visualizations
- Differentiation and integration of functions
- Simple equation solver (for example finding a variable)

1.2.4 Won'ts

- Imaginary numbers
- 3D function graphing
- Multi-variable equation solver (beyond simple simultaneous equations)

1.3 Report structure

1.3.1 Background

The background section of this report will consist of an analysis of similar systems. This will allow us a better understanding of the functionalities our system should contain as well as negative points our system should avoid.

1.3.2 Methodology

This section outlines the design patterns and data structures we used in our software.

1.3.3 Implementation

Our implementation describes the structure we used to develop our system and the requirements we met in each sprint. It also details how we adapted our requirements overtime.

1.3.4 Testing

The testing section is a record of how we ensured each requirement was met, and the functionality performed as expected.

1.3.5 Conclusion

This chapter details the successes and failures of our system and what we have achieved during this process.

Chapter 2

Background

To better understand what we do and don't want from our software, we analysed systems on the market that have a similar function to our own. We analysed the good and bad features of each and identified any we would take forward into our own product.

2.1 Similar Systems

2.1.1 MATLAB[1]

MATLAB is a standalone piece of mathematical software designed with a focus on manipulation of mathematical data types such as matrices as well as including generic programming. It is a Turing complete language, so it has a very wide range of possible uses, but it is very much geared towards easily running numeric calculations. It has its own syntax, which means that users will have to learn it, but the custom syntax it uses is very similar to many popular programming languages, and so users with that background may find transitioning easier. From this program, we like the layout of the user interface. It is intuitive and everything you need is in easy reach. This is the sort of UI we would like to implement in our own product. However, this program takes up a lot of storage and does not run well on lower specification machines. We would like our own product to be able to run on most desktop devices, so it is more accessible.

2.1.2 Math Inspector[2]

Math Inspector is a package of add-ons for Python which allow for mathematical functions to be graphically visualized, both by displaying a flow-type diagram of the function and graphing the numeric outputs across a range. Since it is an add-on for Python, people who are familiar with the language already shouldn't have much difficulty using it. It may, however, mean that people have to install more than what they need in order to get it to run. Whilst the python basis may make it easier for some people, many who wish to use this program will have no prior python knowledge. For this reason, we feel the benefits of a familiar language are outweighed by the cost of having to install the extra programs. Therefore, we will not be doing something similar in own system.

2.1.3 Wolfram Mathematica[3]

Wolfram Mathematica is an interactive tool used to define and visualize mathematical functions. Its main focus is using technology to visualize mathematics and to allow the user to directly implement a vast variety of features. It offers over 5000 different in-built maths functions, meaning it contains very few restrictions. It runs on its own in-built language which may take the user some time to learn and become familiar with. Its main strength is definitely its advanced visualization features. The vast variety of tools it offers in nice but will be a waste for most people. For our system, we will focus on the most common maths functions to maximize use whilst minimizing memory cost. In the future, our system could use add-ons to cover more areas of mathematics without the user having to install hundreds of tools they will never need.

2.1.4 Maple[4]

Maple offers a number of different versions of its software allowing the user to tailor their experience before they even download. Its main goal is solving equations, but it still offers an intuitive UI with function visualization features. It allows the user to explore a range of mathematical fields by offering a large variety of in-built functions. Similarly to the previous system, it operates using its own programming language which may take users time to learn and understand. The different versions of the software (tailored based on who is using it) is this systems' main strength. Our system should also include an intuitive user interface. The different systems versions are a definite positive and something we would take forward in future versions of our product. For now, our product will focus on the most commonly used areas of mathematics to maximize usability.

Chapter 3

Methodology

Throughout this project, we used a number of different method to implement the various features of our software. These methods are outlined below.

3.1 Language Syntax

The language uses a C or Java-like syntax. It contains basic arithmetic operators (+ - * / ˆ) and comparison operators (<, <=, >, >= and =). Assignment uses the <- operator. Variables can be declared with the keyword ”var” and functions with the keyword ”fun”. Statements can be in a code block between curly brackets. True, false, null, strings, integers and floats can be input as literals. If and while statements are available for control flow. For the full grammar, see appendix A.4 on page 28.

3.2 Method 1: Recursive Descent

This algorithm is a type of top-down parsing that functions by recursively calling functions that represent the non-terminal symbols in the grammar. At each stage an Expression node is created and passed up the recursive chain. This constructs the abstract syntax tree with the highest precedence operations nearer to the leaves and the lower precedence operations nearer to the root. In so doing it is building the tree such that the interpreter can follow it using a depth-first post-order traversal to visit all the nodes in the appropriate order as defined by the grammar and enforced by the parser.

Each function has a counterpart ”prime” function to avoid problems with infinite recursion, allowing chains of arbitrary length.

3.3 Method 2: Abstract Syntax Tree

Each type of operator (e.g. binary, unary, etcetera) is defined by a class that extends the Expression class. Each of these acts as a node in the tree and stores leaf nodes containing other expressions. Statements work similarly but extend from the Statement class instead.

3.3.1 Post-Order Traversal

The way that the abstract syntax tree is constructed ensures that performing a post order traversal on it performs the operations in accordance with the precedence rules of the language. See figure 3.1 for an example.

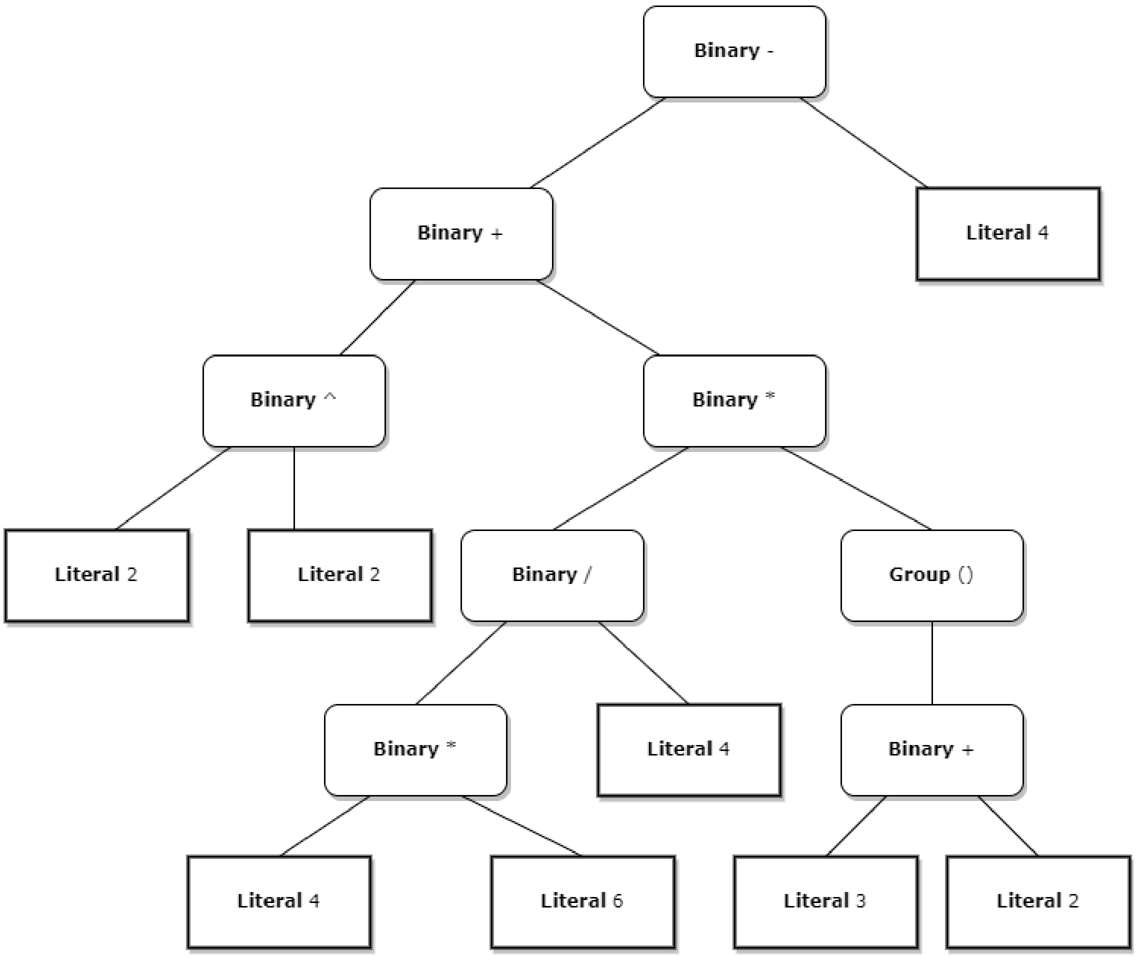


Figure 3.1: An abstract syntax tree for the expression $2\hat{}2 + 4 * 6/4 * (3 + 2) - 4$

3.3.2 Visitor Design Pattern

We used the visitor design pattern to enable distinct functionality between types of nodes without the interpreter needing to manually check the type of any node. The interpreter implements an interface and overrides the visit function for each type of node with the appropriate behaviour. We used the book "Crafting Interpreters" [5] as a reference for implementing this pattern. See appendix A.5 on page 29 for example code.

Chapter 4

Implementation

We are implementing the agile development methodology for this project with a SCRUM inspired approach, using short 1-2 week sprints and adjusting the development trajectory at the end of each.

4.1 Sprint 1

For our first sprint, we defined the following set of requirements:

1. Create the program skeleton we can build upon
2. Build the Lexer and parser for our language
3. Allow the program to identify integers
4. Allow the program to recognize basic operators, including addition, multiplication, subtraction, division and brackets
5. The program should be able to execute the basic operation is can identify

Requirement 1:

The program now contains the different classes it will need to run. These include the Lexer Class which is the class used to identify the different tokens, The parser class that checks the syntax of the languages and the execution class that runs through the stack and performs the operations held there in the correct order. There is also the main class that is responsible for the input display and parsing that input to the Lexer.

Requirement 2:

Once all the classes were created, we were able to build the Lexer and parser for our language. The Lexer can correctly identify integers, plus signs, minus signs, multiplication and division signs and brackets as well as ignoring blank space. The only limitation of our current Lexer is that it is unable to handle integer outside the integer limit imposed by Java.

The parser can check the syntax of the language by making sure the tokens are written the correct order and lets the user know if there is an issue.

Requirement 3:

When the Lexer encounters a digit character, a loop begins and checks each concurrent characters' digit status. If the next character is a digit also, the loop continues. When the Lexer encounters a non-digit character, the loop ends and a number token is added to the token array and the string of digits is converted into a numerical representation which is put into the symbol table. The symbol table is implemented as a map. The index key for the numeric value is the index in the array of the corresponding number token.

Requirement 4:

The recognition of basic operators works in the same manner as the integer recognition except without the need for a looping function. Each operator is assigned a given token which is added to the token array upon recognition.

Requirement 5:

Each token from the token array is copied onto the operator stack and the integers corresponding to the number tokens are copied onto the number stack from the symbol table. When a complete expression has been transferred onto the stack, it will be evaluated and the result of that expression with replace the operands used on the number stack. Any operators used are removed from the stack. Token will then continue to be copied onto the two stacks until the end of the token array is reached.

When the program encounters a right bracket, we know a left bracket must exist on the stack as the parser has validated the code. This means that when a right bracket is encountered, the program continually runs calculate until the left bracket reaches the top of the stack. This ensures the number at the top of the number stack is the evaluation of the expression within brackets, ensuring the order of operations is preserved.

4.2 Sprint 2

For our second sprint, we defined the following set of requirements:

1. Allow the program to recognize variable identifiers
2. Be able to assign value to a variable
3. The program should be able to evaluate expression that contain pre-defined variables

4.2.1 Requirement 1:

The lexer class cycles through the characters typed by the user. If the character is not a number or a reserved symbol, the program identifies it as a variable. The consecutive characters are then checked and added to the variable name if they are alphanumeric. This process ends when a non-alphanumeric character is reached. The symbol table has also been updated, so it can hold variable names as well as numbers. This requirement has been successfully met.

4.2.2 Requirement 2:

To begin developing the assignment functionality, we have used the equals sign to represent assignment for simplicity. This sign has be added as a reserved character to identify assignment operations. When an equals gin is preceded by a variable and proceeded by an integer, the integer value is mapped to the variable. If the assignment value is an expression, the expression is evaluated and the solution mapped to the variable. This requirement has been successfully met.

4.2.3 Requirement 3:

When the program evaluates an expression that contains a variable, the execution class identifies the value mapped to the variable and replaces it in the expression. The expression is then evaluated as normal and the solution displayed. This requirement has been successfully met.

4.3 Sprint 3

For our second sprint, we defined the following set of requirements:

1. Developing the abstract syntax tree
2. Adding the graphing functionality
3. Allowing the support of floats

We were able to fully implement and test these requirements quickly so we integrated a number of extra features into our program on top of our base requirements. These included:

1. Support for negative numbers and unary operators
2. Support for simple statements including print, variable declaration and graph plotting
3. Ability to calculate exponential expressions

Requirement 1:

Each type of operator has its own class. The depth of the tree determines the order in which the operations will be executed. The higher an operator exists within the tree, the lower their priority.

Requirement 2:

To add the graph, we used the Java2D library. Our program take each point, as calculated by the interpreter, and maps them onto a simple axis in a window. Currently, this graph only supports positive axis and does not scale with the window size, this means only simple functions can be plotted.

Requirement 3:

To support floats, our program detects any dots within a number. If a dot is preset, it continues to scan through until the end of the number is reached. This value is then saved as a float rather than a double.

Extra Requirement 1:

We have another class within the Abstract Syntax Tree (AST) that supports unary operators, including negative numbers.

Extra Requirement 2:

Statements also become part of the AST. The expressions they contain are the children of the statement within the tree.

Extra Requirement 3:

Exponential operations are identified in the same way as other binary operators. There is a new step in the parsing process required for it as it has a higher priority than most other operators.

4.4 Sprint 4

Our fourth sprint involved the following set of requirements:

1. Adding support for negative numbers on the graph
2. Allowing the graph to change size dynamically with the window size
3. Add support for Boolean types and logic
4. Add support for control flow statements
5. Add variable scoping

Requirement 1:

This addition was relatively simple. We change to code4 to draw to lines that crossed in the centre to act as the axis, we also modified our mapping function to account for the change in position of (0,0).

Requirement 2:

During our research for this requirement, we found that JPanel has built in getWidth() and getHeight() functions. This meant we could replace our hard-coded constants for these values in our calculations which meant anything drawn in the window would automatically scale as the window size was changed. We also found out about the Points2D library which is compatible with Java 2D, so we changed our code to handle points with this library rather than manually.

Requirement 3:

This is another literal type and the comparisons for them are binary operators that exist within the abstract syntax tree. We also have support for ”and” and ”or” which works similarly to other binary operators but with the ability to short circuit depending on the first operand.

Requirement 4:

Similarly to other statements, these additions are new nodes in the syntax tree with the same precedence as other statements like print. If statements looks for an expression as the condition and a statement as the thing to do if the condition is true. While statements work similarly except for that they repeat the statement until the expression is no longer true.

Requirement 5:

A variable defined within a block statement will be local to that block. Any variables created within that block are discarded once that block has been completed. A new environment is created when a block statement is entered and any variables are kept there.

4.4.1 Sprint 5

These were the requirements we outlined for our final sprint:

1. Adding function syntax
2. Creating a degrees to radians function
3. Adding more user customization of the graphing feature
4. Create a root finding algorithm to solve equations

Requirement 1:

This works similarly to defining a variable. The identifier of the name of the function is saved as a variable with a special type known as Function. Only variables with this type can be used with the call operator (open and close bracket) with parameters inside (if required). The fact it is treated as a variable means it can be passed as such. This allows a function to return a function or pass it as an argument.

Requirement 2:

We have implemented the ability to convert between degrees and radians, such that it can be called with the above function syntax, making it a native function.

Requirement 3:

We created more native functions to allow the user to define the parameters of the graph which then are used when the plot statement is used.

Requirement 4:

Time restrictions on this project meant we were unable to meet this requirement.

Chapter 5

Testing

This section details the testing we performed to make sure we meet each of the requirements our software aims to achieve. When we were just beginning the program, we used a cyclical testing. This meant we continually tested each feature as it was developed. Once we had finished the first two sprints, we switched to unit testing and began testing the individual components of the code. The result of our initial testing for sprint one and two are recorded (see appendix) as are the results of our component tests (see appendix A.1 on page 17).

Once the program was complete, we conducted a full black box test of our program. This means testing our program as if we did not have access to the code. We tested each requirement by splitting it into sub requirements and performing multiple tests for each. An example of the testing for our first requirement is given here.

The rest of our testing can be found in Appendix A.2 on page 18 with the evidence for each of these tests in appendix A.3 on 23.

Requirement 1:		Expressions For Basic Arithmetics		
Sub-Requirement 1:		Addition Expressions		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(1+2)	3	3	P
2	print(3.2+5.6)	8.8	8.8	P
3	print(-10+185)	175	175	P
Sub-Requirement 2:		Subtraction Expressions		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(10-8)	2	2	P
2	print(50-263)	-213	-213	P
3	print(10.6-0.5)	10.1	10.1	P
Sub-Requirement 3:		Multiplication Expressions		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(4*7)	28	28	P
2	print(20*5.8)	116.0	116.0	P
3	print(-3*10)	-30	-30	P
Sub-Requirement 4:		Division Expressions		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(10/4)	2	2	P
2	print(500/2.3)	217.39..	217.39..	P
3	print(-25/5)	-5	-5	P

Chapter 6

Discussion, conclusion and future work

Over the course of the project we've achieved many of the requirements that we set out to achieve in our MoSCoW. We've built up an architecture for the interpreter which is easily extendable and scalable, lending itself well to the features we have implemented during development. Our grammar includes commands such as print and plot which work in parallel with the basic arithmetic features of our language. These features, alongside our custom lexical analyser and parser, fulfil all of our "Must" criteria. Furthermore, our language contains syntax for both global and local variable assignment as well as control flow statements which allows conditional execution of code. This required the addition of logical operators in conjunction with boolean expressions. Developing our language further, we added the syntax for functions which are implemented as identifiers like any other variable. Our program is also able to visualize mathematical expressions on a custom graph whose parameters can be defined by the user. Finally, we added a range of trigonometric functions alongside a pre-defined variable for Pi as a utility to the language. Altogether, these features encompass most of our "Should" criteria, with the exception of the "Zero-crossings finder" criterion. Overall, we have a multifaceted maths interpreter language that could easily be extended in the future by building upon the established architecture. Such extensions could include, functions for integration and differentiation as well as an equation solver. From the user's perspective, the software would be improved by the addition of a more interactive and intuitive graphical user interface. All of these would be able to easily integrated into the program after they have been designed conceptually. Whilst we were not able to add all the features we had initially planned, we were able to produce a stable and practical piece of software.

Bibliography

[1] MathWorks, Natick, Massachusetts, United States. *MATLAB*, 2017.

[2] *Math Inspector*, 2021.

[3] Wolfram, The Wolfram Centre, Lower Road, Long Hanborough, Oxfordshire OX29 8FD, United Kingdom. *Wolfram Mathematica*, 2021.

[4] Maplesoft, 615 Kumpf Drive, Waterloo, ON, Canada, N2V 1K8. *Maple*, 2021.

[5] Robert Nystrom. *Crafting interpreters.* Genever Benning, 2021.

Contributions

State here the % contribution to the project of each individual member of the group and describe in brief what each member has done (if this corresponds to particular sections in the report then please specify these).

Appendix

A.1: Testing for Sprint 1 and 2

Sprint 1

Test No.	Test Case	Test Data	Expected Outcome	Result
1	The program is able to identify a number	5	NUMBER	PASSED
		83	NUMBER	PASSED
		324876312487124	NUMBER	FAILED
2	The program can identify and evaluate an addition expression	5+5	10	PASSED
		54+6	60	PASSED
		2+7	9	PASSED
3	The program can identify and evaluate a subtraction expression	5-5	0	PASSED
		18-9	9	PASSED
		4-0	4	PASSED
4	The program can identify and evaluate a multiplication expression	5*5	25	PASSED
		4*7	28	PASSED
		32*8	256	PASSED
5	The program can identify and evaluate a division expression	5/5	1	PASSED
		42*7	294	PASSED
		100/10	10	PASSED
6	The program can evaluate expressions that include blank space	5 + 5	10	PASSED
		10 - 8	2	PASSED
		32 - 23	9	PASSED
7	The program can identify and evaluate expression with multiple operators	4+7-2	9	PASSED
		23+6+4	33	PASSED
		1*8/4	2	PASSED
8	The program can identify and handle incorrect syntax	5+	Expression expected	PASSED
		5+(5	Expression expected	PASSED
		*8	Expression expected	PASSED
9	The program can identify and evaluate expression involving brackets	5+(4-2+3)	10	PASSED
		(4*2)+7	15	PASSED
		10/5+(6*8)	50	PASSED

Sprint 2

Test No.	Test Case	Test Data	Result
1	The program can assign a value to a variable	x=5	PASSED
		foo=56	PASSED
		y=9	FAILED
2	The program can assign an expression to a variable	a = 19+7	PASSED
		y = 7-5	PASSED
		bar = 3*6	PASSED
3	The program can evaluate an expression that includes a variable	x-10	PASSED
		y = 7+x	PASSED
		foo*x	PASSED
4	The program can identify incorrect assignment expression	5x	FAILED
		x465	FAILED
		foo x+6	FAILED

A.2: Results for final testing

Requirement 2:		Print Statements		
Sub-Requirement 1:		Printing Strings		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(”hello”);	hello	hello	P
2	print(”I like cheese”);	I like cheese	I like cheese	P
3	print(”a1b2c3”);	a1b2c3	a1b2c3	P
Sub-Requirement 2:		Print Numbers		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(84326);	84326	84326	P
2	print(-48);	-48	-48	P
3	print(3.4);	3.4	3.4	P
Sub-Requirement 3:		Print Expressions		
Tested in Requirement 1 Page 13				

Requirement 3:	Other Commands
Sub-Requirement 1:	Print
Tested in Requirement 2 Page 19	
Sub-Requirement 2:	Plot
Tested in Requirement 8 Page 21	

Requirement 4:	Custom syntax using a custom lexer and parser
Shown by all other parts of the language working	

Requirement 5:		Variable Assignment		
Sub-Requirement 1:		Assign Numbers		
Test No.	Data	Expected Outcome	Outcome	P/F

1	var x<-5;
	print(x);
		5	5	P

2	var dec<-10.56;
	print(dec);
		10.56	10.56	P

3	var neg<-10.56;
	print(neg);
		-3	-3	P

Sub-Requirement 2:		Assign Strings		
Test No.	Data	Expected Outcome	Outcome	P/F
1	var hey<-”hello”; print(hey);	hello	hello	P
2	var ch<-”cheese”; print(ch);	cheese	cheese	P
3	var odd<-”1a2b3c”; print(odd);	1a2b3c	1a2b3c	P
Sub-Requirement 3:		Assign Expressions		
Test No.	Data	Expected Outcome	Outcome	P/F

1	var ten<-6+4;
	print(ten);
		10	10	P

2	var zero<-ten-10;
	print(zero);
		0	0	P

3	var div<-1000/25;
	print(div);
		40	40	P

4	var mult<-8*6;
	print(mult);
		48	48	P

Requirement 6:		Conditional Statements		
Sub-Requirement 1:		If/Else Statements		
Test No.	Data	Expected Outcome	Outcome	P/F

1	var x<-6;
	if(x=6){print(6);}
	else{print(”not 6”);}
		6	6	P

2	x<-10;
	if(x=6){print(6);}
	else{print(”not 6”);
		not 6	6	P

3	if(x<15){print(”under 15”);}	under 15	6	P
Sub-Requirement 2:		While Statements		
Test No.	Data	Expected Outcome	Outcome	P/F

1	var x<-17;
	while(x<20){
	print(”hi”);
	x<-x+1;}
		hi
		hi
		hi
			hi
			hi
			hi
				P

2	x<-0;
	while(x<5){
	print(x);
	x<-x+1;}
		0
		1
		2
		3
		4
			0
			1
			2
			3
			4
				P

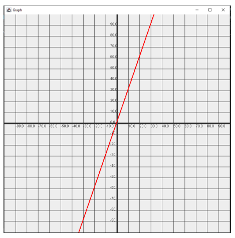
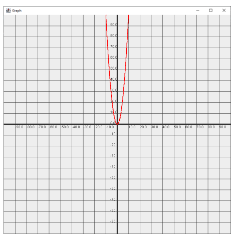
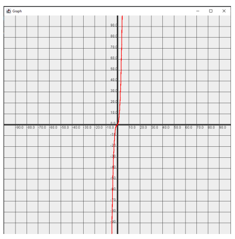
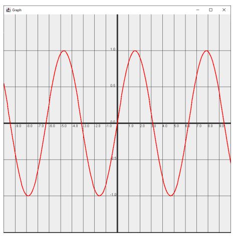
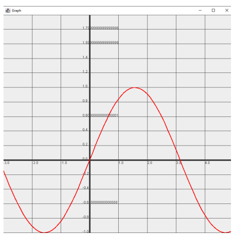
3	x<-10;
	while(x>5){
	print(x);
	x<-x-1;}
		10
		9
		8
		7
		6
			10
			9
			8
			7
			6
				P

Requirement 7:		Function Declaration		
Test No.	Data	Expected Outcome	Outcome	P/F

1	fun count(x,max){
	print(x);
	x<-x+1;
	if(x<max){count(x,max);}
	else{return(x);}}
	count(0,4);
		0
		1
		2
		3
			0
			1
			2
			3
				P

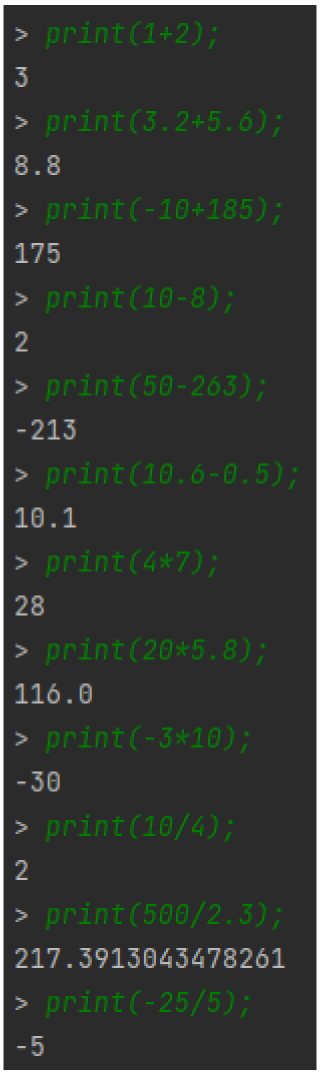
2	fun fib(a,b,c,max){
	print(a);
	c<-a+b;
	a<-b;
	b<-c;
	if(c<max){fib(a,b,c,max);}
	else{return(c);}}
	fib(1,1,0,10);
		1
		1
		2
		3
		5
			1
			1
			2
			3
			5
				P

3	fun age(born, current){
	return(current-age);}
	print(age(1988,2021));
		33	33	P

Requirement 8:		Function Visualisation		
Test No.	Data	Expected Outcome	Outcome	P/F
1	maxPoint(100,100); minPoint(-100,-100); setIncrement(10,10); plot(3*x+2);	A graph displaying the function 3x+2 should appear		P
2	plot(xˆ2);	A graph displaying the function xˆ2 should appear		P
3	plot(xˆ3+xˆ2+x);	A graph displaying the correct polynomial should appear		P
4	minPoint(-10,-1.5); maxPoint(10,1.5); setIncrement(1,0.5); plot(sin(x));	A graph displaying the function sin(x) should appear		P
3	maxPoint(5,2); minPoint(-3,-1); setIncrement(1,0.2); plot(sin(x));	A graph with the new parameters should appear displaying the sin(x) function		P

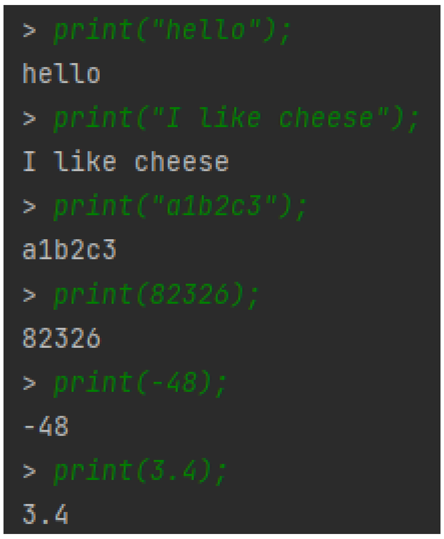
Requirement 9:		Simple Trig Functions		
Sub Requirement 1:		Sin		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(sin(30));	-0.98803162409	-0.98803162409..	P
2	print(sin(50.6));	0.32831349385	0.32831349385..	P
3	print(sin(-15));	-0.65028784015	-0.65028784015..	P
Sub Requirement 2:		Cos		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(cos(60));	-0.95241298041	-0.95241298041..	P
2	print(cos(13.2));	0.80588395764	0.80588395764..	P
3	print(cos(-20));	0.40808206181	0.40808206181..	P
Sub Requirement 3:		Tan		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(tan(90));	-1.99520041221	-1.99520041221..	P
2	print(tan(4.2));	1.77777977451	1.77777977451..	P
3	print(tan(-10));	-0.64836082745	-0.64836082745..	P
Sub Requirement 4:		Arcsin		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(arcsin(1));	1.57079632679	1.57079632679..	P
2	print(arcsin(0.5));	0.52359877559	0.52359877559..	P
3	print(acrsin(-1));	-1.57079632679	-1.57079632679..	P
Sub Requirement 5:		Arccos		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(arccos(1));	0	0	P
2	print(arccos(0.5));	1.0471975512	1.0471975512..	P
3	print(arccos(-1));	3.1415926535	3.1415926535..	P
Sub Requirement 6:		Arctan		
Test No.	Data	Expected Outcome	Outcome	P/F
1	print(arctan(1));	0.73539816339	0.73539816339..	P
2	print(arctan(0.5));	0.463647609	0.463647609	P
3	print(arctan(-1));	-0.73539816339	-0.73539816339..	P

A.3: Evidence for Tests



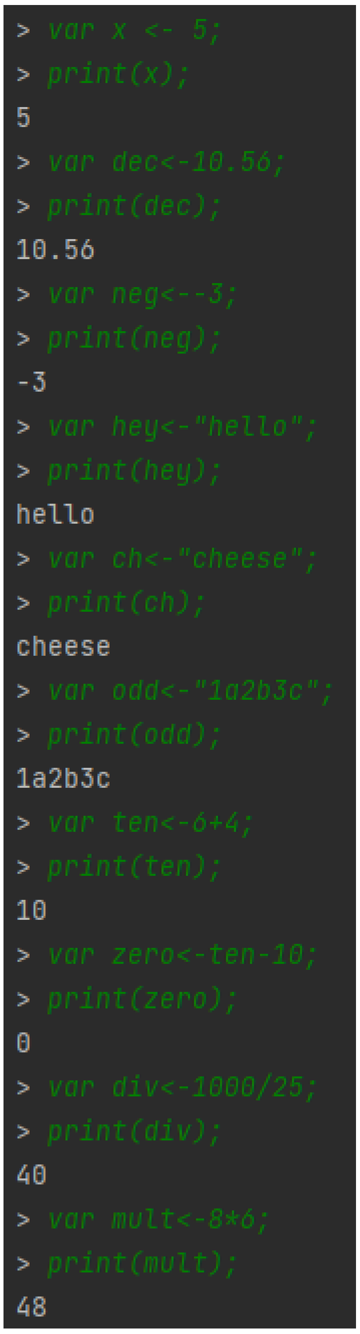
```
> print(1+2);
3
> print(3.2+5.6);
8.8
> print(-10+185);
175
> print(10-8);
2
> print(50-263);
-213
> print(10.6-0.5);
10.1
> print(4*7);
28
> print(20*5.8);
116.0
> print(-3*10);
-30
> print(10/4);
2
> print(500/2.3);
217.3913043478261
> print(-25/5);
-5
```

Figure 6.1: Evidence for Requirement 1 testing



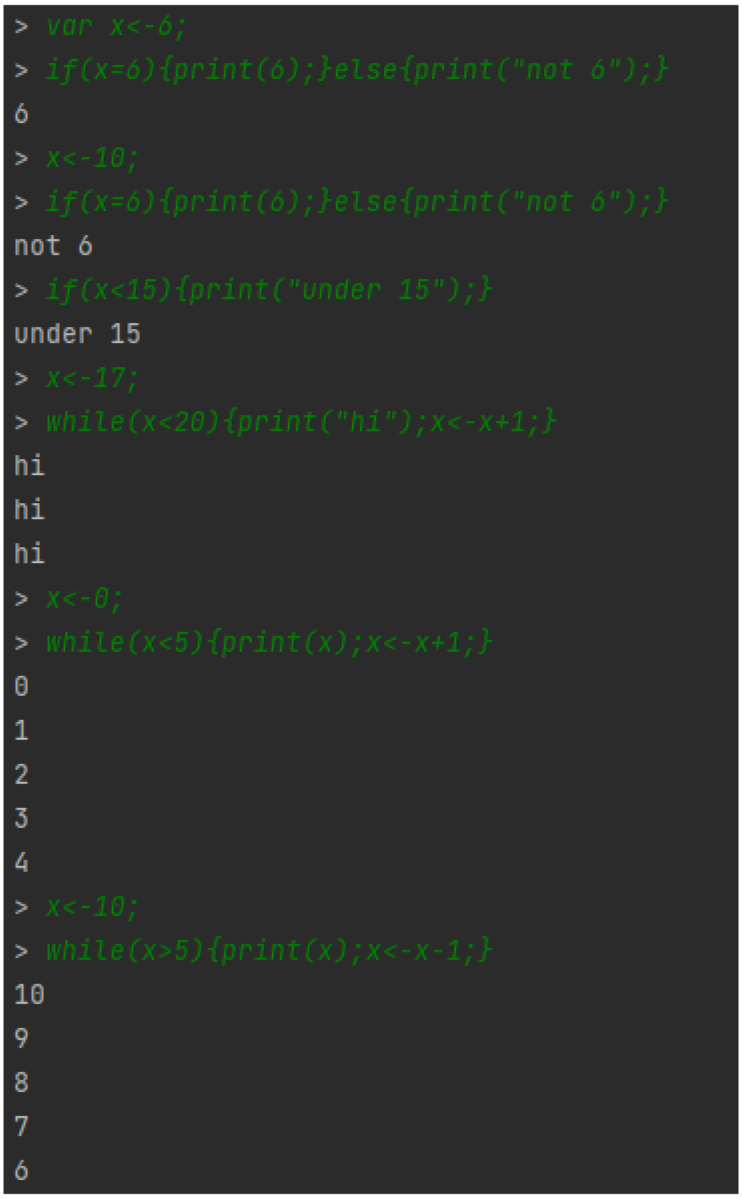
```
> print("hello");
hello
> print("I like cheese");
I like cheese
> print("a1b2c3");
a1b2c3
> print(82326);
82326
> print(-48);
-48
> print(3.4);
3.4
```

Figure 6.2: Evidence for Requirement 2 testing



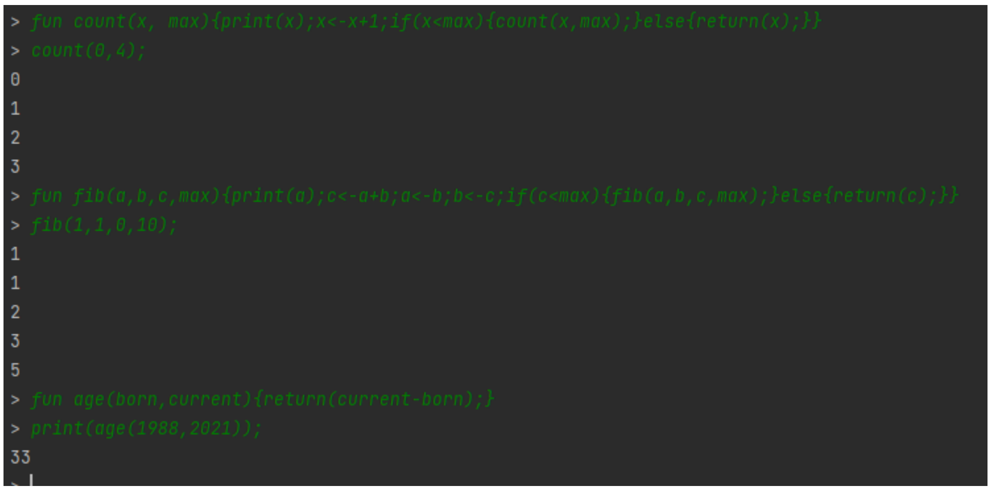
```
> var x <- 5;
> print(x);
5
> var dec<-10.56;
> print(dec);
10.56
> var neg<--3;
> print(neg);
-3
> var hey<-"hello";
> print(hey);
hello
> var ch<-"cheese";
> print(ch);
cheese
> var odd<-"1a2b3c";
> print(odd);
1a2b3c
> var ten<-6+4;
> print(ten);
10
> var zero<-ten-10;
> print(zero);
0
> var div<-1000/25;
> print(div);
40
> var mult<-8*6;
> print(mult);
48
```

Figure 6.3: Evidence for Requirement 5 testing



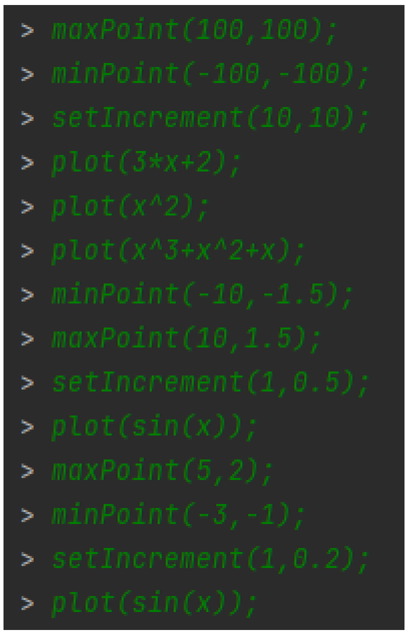
```
> var x<-6;
> if(x=6){print(6);}else{print("not 6");}
6
> x<-10;
> if(x=6){print(6);}else{print("not 6");}
not 6
> if(x<15){print("under 15");}
under 15
> x<-17;
> while(x<20){print("hi");x<-x+1;}
hi
hi
hi
> x<-0;
> while(x<5){print(x);x<-x+1;}
0
1
2
3
4
> x<-10;
> while(x>5){print(x);x<-x-1;}
10
9
8
7
6
```

Figure 6.4: Evidence for Requirement 6 testing



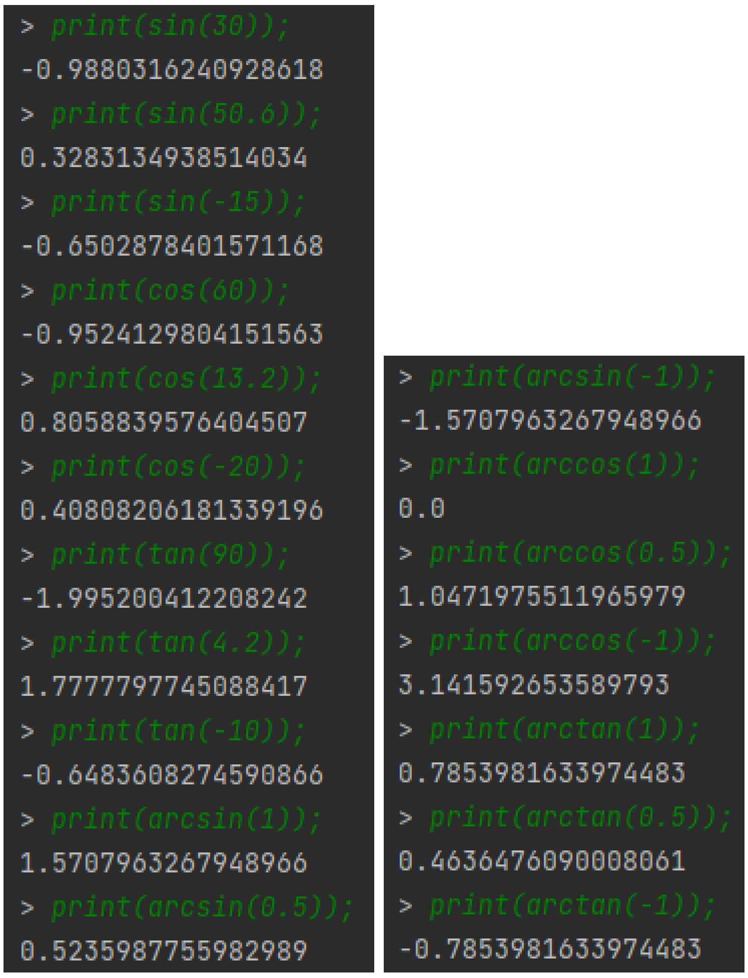
```
> fun count(x, max){print(x);x<-x+1;if(x<max){count(x,max);}else{return(x);}}
> count(0,4);
0
1
2
3
> fun fib(a,b,c,max){print(a);c<-a+b;a<-b;b<-c;if(c<max){fib(a,b,c,max);}else{return(c);}}
> fib(1,1,0,10);
1
1
2
3
5
> fun age(born,current){return(current-born);}
> print(age(1988,2021));
33
```

Figure 6.5: Evidence for Requirement 7 testing



```
> maxPoint(100,100);
> minPoint(-100,-100);
> setIncrement(10,10);
> plot(3*x+2);
> plot(x^2);
> plot(x^3+x^2+x);
> minPoint(-10,-1.5);
> maxPoint(10,1.5);
> setIncrement(1,0.5);
> plot(sin(x));
> maxPoint(5,2);
> minPoint(-3,-1);
> setIncrement(1,0.2);
> plot(sin(x));
```

Figure 6.6: Evidence for Requirement 8 testing



```
> print(sin(30));
-0.9880316240928618
> print(sin(50.6));
0.3283134938514034
> print(sin(-15));
-0.6502878401571168
> print(cos(60));
-0.9524129804151563
> print(cos(13.2));
0.8058839576404507
> print(cos(-20));
0.40808206181339196
> print(tan(90));
-1.995200412208242
> print(tan(4.2));
1.7777797745088417
> print(tan(-10));
-0.6483608274590866
> print(arcsin(1));
1.5707963267948966
> print(arcsin(0.5));
0.5235987755982989
```

```
> print(arcsin(-1));
-1.5707963267948966
> print(arccos(1));
0.0
> print(arccos(0.5));
1.0471975511965979
> print(arccos(-1));
3.141592653589793
> print(arctan(1));
0.7853981633974483
> print(arctan(0.5));
0.4636476090008061
> print(arctan(-1));
-0.7853981633974483
```

Figure 6.7: Evidence for Requirement 9 testing

A.4: Full Grammar

⟨declaration⟩	::=	⟨functionDeclaration⟩ | ⟨variableDeclaration⟩ | ⟨statement⟩
⟨functionDeclaration⟩	::=	`fun` ⟨identifier⟩ `(` ⟨parameters⟩ `)` ⟨block⟩
⟨parameters⟩	::=	λ | ⟨identifier⟩ | ⟨parameters⟩ `,` ⟨identifier⟩
⟨variableDeclaration⟩	::=	`var` ⟨identifier⟩ `;` | `var` ⟨identifier⟩ `<-` ⟨expression⟩ `;`
⟨statement⟩	::=	⟨expressionStatement⟩ | ⟨ifStatement⟩ | ⟨printStatement⟩
		| ⟨plotStatement⟩ | ⟨returnStatement⟩ | ⟨whileStatement⟩
		| ⟨block⟩
⟨expressionStatement⟩	::=	⟨expression⟩ `;`
⟨ifStatement⟩	::=	`if (` ⟨expression⟩ `)` ⟨statement⟩
		| `if (` ⟨expression⟩ `)` ⟨statement⟩ `else` ⟨expression⟩
⟨printStatement⟩	::=	`print` ⟨expression⟩ `;`
⟨plotStatement⟩	::=	`plot` ⟨expression⟩ `;`
⟨returnStatement⟩	::=	`return ;` | `return` ⟨expression⟩ `;`
⟨whileStatement⟩	::=	`while (` ⟨expression⟩ `)` ⟨statement⟩
⟨block⟩	::=	{ ⟨declarationList⟩ }
⟨declarationList⟩	::=	λ | ⟨declaration⟩ | ⟨declarationList⟩ ⟨declaration⟩
⟨expression⟩	::=	⟨assignment⟩
⟨assignment⟩	::=	⟨identifier⟩ `<-` ⟨assignment⟩ | ⟨or⟩
⟨or⟩	::=	⟨and⟩ | ⟨or⟩ `or` ⟨and⟩
⟨and⟩	::=	⟨equality⟩ | ⟨and⟩ `and` ⟨equality⟩
⟨equality⟩	::=	⟨comparison⟩ | ⟨equality⟩ `=` ⟨comparison⟩
⟨comparison⟩	::=	⟨sum⟩ | ⟨comparison⟩ `<` ⟨sum⟩
		| ⟨comparison⟩ `<=` ⟨sum⟩
		| ⟨comparison⟩ `>` ⟨sum⟩
		| ⟨comparison⟩ `>=` ⟨sum⟩
⟨sum⟩	::=	⟨term⟩ | ⟨sum⟩ `+` ⟨term⟩ | ⟨sum⟩ `-` ⟨term⟩
⟨term⟩	::=	⟨exponential⟩ | ⟨term⟩ `*` ⟨exponential⟩ | ⟨term⟩ `/` ⟨Exponential⟩
⟨exponential⟩	::=	⟨unary⟩ | ⟨exponential⟩ `^` ⟨unary⟩
⟨unary⟩	::=	⟨call⟩ | `!` ⟨unary⟩ | `-` ⟨unary⟩
⟨call⟩	::=	⟨factor⟩ | ⟨call⟩ `(` ⟨arguments⟩ `)`
⟨arguments⟩	::=	λ | ⟨expression⟩ | ⟨arguments⟩ `,` ⟨expression⟩
⟨factor⟩	::=	`true` | `false` | `null` | ⟨number⟩ | ⟨string⟩ | ⟨identifier⟩
⟨identifer⟩	::=	⟨alphaChar⟩ | ⟨identifier⟩ ⟨alphaChar⟩ | ⟨identifier⟩ ⟨digit⟩
⟨string⟩	::=	`"` *any number of any character except ” * `"`
⟨alphaChar⟩	::=	`a`...`z` | `A`...`Z` | _
⟨number⟩	::=	⟨integer⟩ | ⟨float⟩
⟨float⟩	::=	⟨integer⟩ `.` ⟨integer⟩
⟨integer⟩	::=	⟨digit⟩ | ⟨integer⟩ ⟨digit⟩
⟨digit⟩	::=	`0`...`9`

A.5: Visitor Pattern Code Example

Listing 6.1: An example of the implementation of a node—the unary expression in this case

```
abstract class Expression {

  interface Visitor<T> {

    T visit(Unary expression);
  }

  static class Unary extends Expression {
    final Token operator;
    final Expression right;

    Unary(Token operator, Expression right) {
      this.operator = operator;
      this.right = right;
    }

    @Override
    <T> T accept(Visitor<T> visitor) {
      return visitor.visit(this);
    }
}

  abstract <T> T accept(Visitor<T> visitor);
}
```