iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 10 **Issue:** VIII **Month of publication:** August 2022

DOI: https://doi.org/10.22214/ijraset.2022.46665

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 10 Issue VIII Aug 2022- Available at www.ijraset.com

Developing programming language with compilers using JFlex in NetBean: Expanding and testing simple operators by implementing a calculator

Younten Tshering[1], Suyogya Ratna Tamrakar[2], Sai Preetham Kamishetty[3]
[1]*Jigme Namgyel Engineering College, Royal University of Bhutan*
[2.3]*Asian Institute of Technology*

Abstract: ***JFlex is a lexical analyzer generator and takes input requirements with a set of regular expressions and corresponding actions. It creates a program (a lexer) that reads input, matches the input against the regular expressions, and runs the matching action. This paper shows how Programming Language can be developed. This work was done to develop a simple programming language with compilers using JFlex in NetBean so that it can support assignment statements, if then else, while do and type checking and its execution. The data type included are int, real, char, and Boolean/String. The key concept used in this work was the execution of the grammar or rules in the cup file. The parse tree records a sequence of rules the parser applies to recognize the input. The tool used for the development of lexical analyzers was JFlex. JFlex Lexers was based on deterministic finite automata (DFAs). To show the implementation and working of operators, a simple calculator was designed that supports addition and multiplication operations. Further, the key compiler concepts like lexical analyzers, semantic analysis, and parse trees are discussed. This paper will help understand the syntax and way to develop simple language. For the programming language developed, the evaluations of the expressions and statements are recursively done. Type checking and Error checking are also done where two operands are checked for their compatibility with the operator and are shown if incompatible expressions are found.***

Keywords: Calculator, Jflex, Lexical Analyzers, NetBean, and Programming Language

I. INTRODUCTION

JFlex is a lexical analyzer generator or scanner generator for Java [1]. A lexical analyzer generator takes input requirements with a set of regular expressions and equivalent actions. It creates a program (a lexer) that reads input, matches the input against the regular expressions in the spec file, and runs the matching action of the regular expression [2]. Lexer typically is the initial front-end step in compilers, matching keywords, comments, and operators and generating an input token stream for parsers. Lexers can also be used for many other purposes. JFlex is designed to work together with the LALR parser generator CUP, and the Java modification. It can also be used together with other parser generators like ANTLR (ANother Tool for Language Recognition) or as a standalone tool. JFlex supports JDK 1.8 or above for build and JDK 7 and above for run-time. JFlex Lexers are based on deterministic finite automata (DFAs). They are fast, without expensive backtracking. A standard tool for the development of lexical analyzers is JFlex.

A JFlex program consists of three parts such as 1. User code, 2. Options and 3. Translation rules. User code is copied precisely into the beginning of the Java source file of the generated lexer. This is the place for package declaration and import statements. Options customize the generated lexer and declare constants, variables, and regular definitions. Translation rules have the form *p {action}* where p is a regular expression and action is Java code specifying what the lexer executes when a sequence of characters of the input string matches p. Thus, as per [3] the lexical analyzer created by Lexer works as it gets activated, the lexical analyzer reads the remaining input one character at a time until it has found the longest prefix that is matched by one of the regular expressions on the left-hand side of the translation rules. This is the paper that shows how basic Programming Language is developed. This article will help to know the syntax and run the language. A simple programming language was developed with compilers using JFlex in NetBean so that it can support assignment statements, if then else, sequential that can be run sequentially and type checking and its execution. The data type includes int, real, char, and Boolean/String. To show the implementation and working of operators a simple calculator was designed that supports addition and multiplication operations, and to know the key compiler concepts like lexical analyzers, semantic analysis, and parse trees. The code for this language can be written in an input session and run in its own UI based on Java. Errors are displayed in the output console with color-coded strings, i.e., red for error and black for normal output.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 10 Issue VIII Aug 2022- Available at www.ijraset.com

II. METHOD

A standard tool for the development of lexical analyzers is JFlex. JFlex supports JDK 1.8 or above for build and JDK 7 and above for run-time. JFlex Lexers are based on deterministic finite automata (DFAs). They are fast, without expensive backtracking. Implementing Ubuntu was utilized since the Ubuntu desktop is easy to use, easy to install, and includes everything we need to run our work. It is also open source, secure, accessible, and free to download.

We need to consider the following before installation such as having at least 25 GB of free storage space, having access to either a DVD or a USB flash drive containing the version of Ubuntu you want to install, and making sure we have a recent backup of our data. After that, the installation process is simple to follow as we have minimum hardware and software requirements. Once we have installed Ubuntu in our system then we need to install NetBean as it is an open-source Integrated Development Environment (IDE). For this work, Ubuntu 20.04 was installed on Virtual Box to manage two Operating systems.

A. Installation process

The Ubuntu can be installed in the system or installed on Virtual Box. However, NetBeans can be used for Windows, macOS, and Ubuntu. The main two steps to follow to install NetBean are Install OpenJDK (*sudo apt-get install openjdk-8-jdk*) and Install NetBean. Since now there are new versions of NetBeans and it's not compatible with some old version cup files when we tried. Therefore, it is suggested to use the old version of NetBeans if possible.

Table I briefly shows the software and tools used for developing this calculator and programming language and compiler.:

TABLE I
SOFTWARE AND TOOLS USED

Name	Description
JFlex	A lexical analyzer generator tool based on Java
CUP	An LALR parser generator that works together with JFlex
NetBeans	IDE for Java development
Java Swing	The framework used to build GUI for the application

III. CALCULATOR IMPLEMENTATION FOR TEST

For the implementation of the calculator following symbols were declared:

1) Terminal: *PLUS, MULT, OPAREN – (, CPAREN –), NUMBER*
2) Nonterminal of Parse Tree are *S, E, T, F, K, L, J, M*
3) Grammar Rule implemented in CUP file to have Infix, Prefix, and Postfix operation of Plus (+) and Multiplication (*) is as given below:

Infix Rule:

S = E
E = E + T | T
*T = T * F | F*
F = (E) | NUMBER

Postfix Rule:

S = L
*L = K K + | K K **
K = L
K = NUMBER

Prefix Rule:

S = M
*M = + J J | * J J*
J = M
J = NUMBER

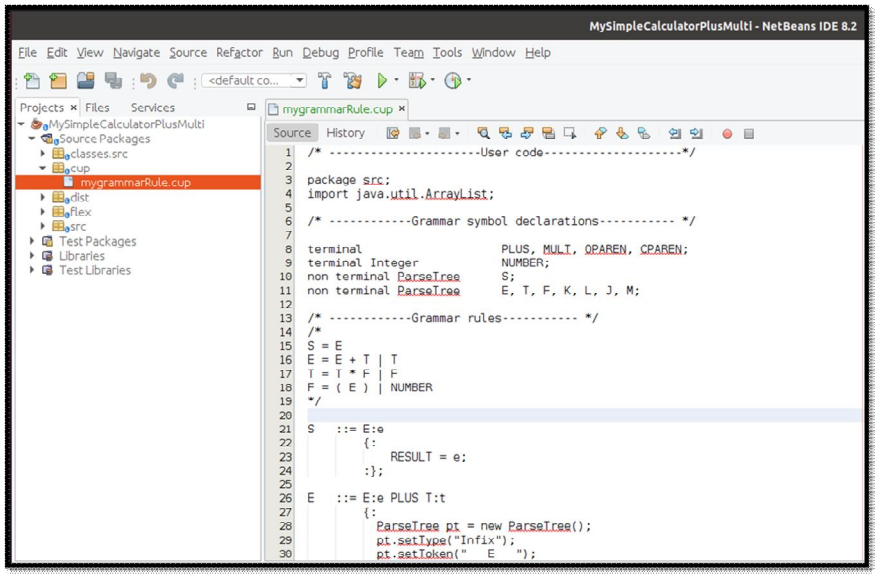


Figure. 1 Grammar in NetBeans

Figure 1 shows the Calculator with grammar rules in the cup file. In the *mygrammarRule.cup*, the rule for the parse tree is defined. The grammar consists of Infix, Postfix, and Prefix notation for addition and multiplication operations.

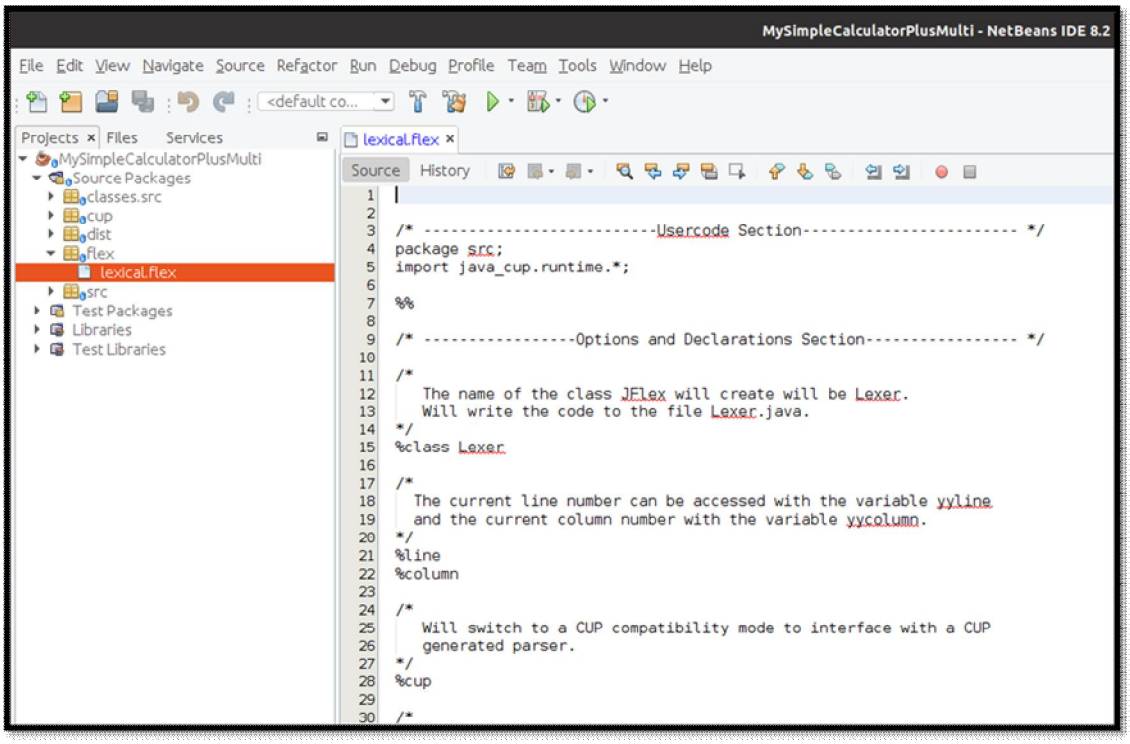


Figure. 2 Lexical Analyzer

A standard tool for the development of lexical analyzers is Jflex and we can see lexical.flex file where it consists of User code, Options, and Translation rules. This section contains regular expressions and actions, i.e., Java code, that will be executed when the scanner matches the associated regular expression.

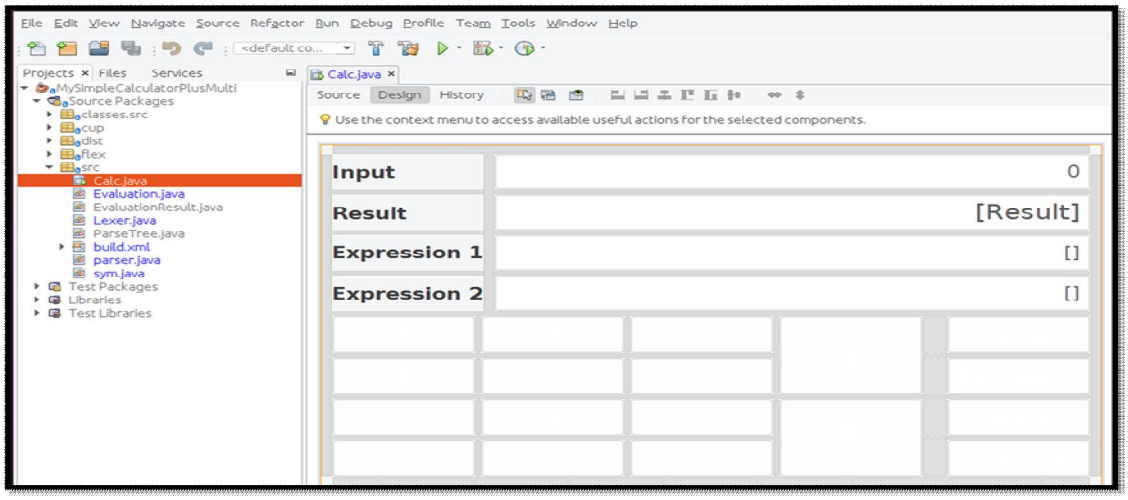


Figure 3. Output Screen design

Figure 3 shows the interface design and where the output-oriented programs are performed. *Evaluation.java* and *EvaluationResult.java* link the calculator interface with operation functionality and display the output on the screen depending on the input. The Calculator takes integer input and computes the operation and displays it. If the input is an infix, then the output should be the value of the operation and show the prefix and postfix expression of it.

ParseTree.java is having getter and setter [5] for the parse tree with token, level, lexeme, and type.

- *parser.java and sym.java* are generated by CUP.
- *Lexer.java* is generated by Jflex.

To generate the above file by cup and Jflex, we use *build.xml* to run in different targets to compile and run. The detail is shown in the below figure.

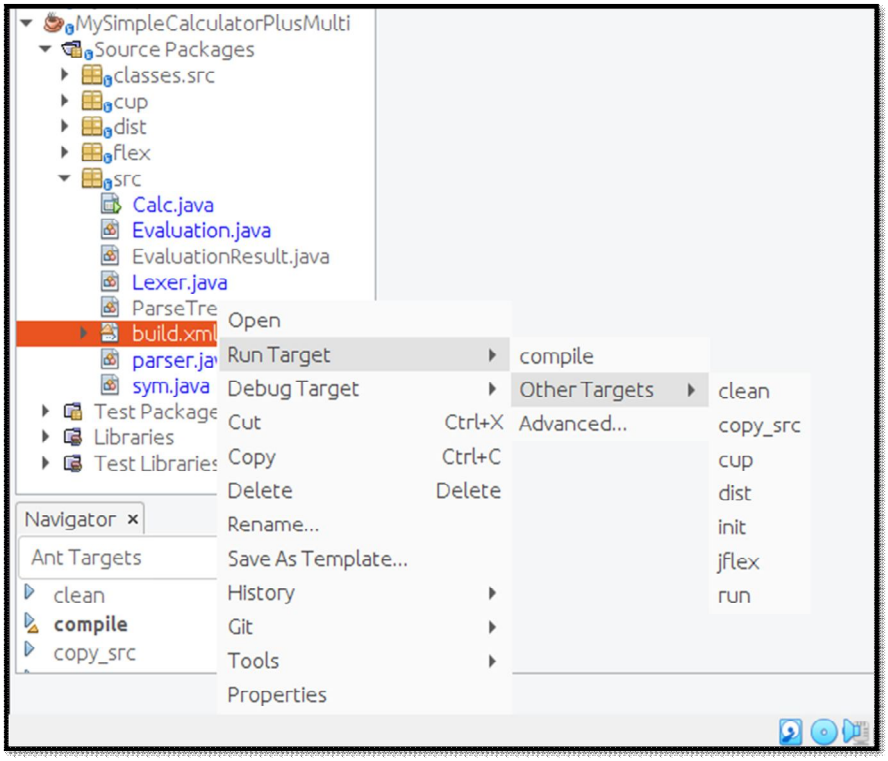


Figure 4. Build to generate CUP and Jflex file

With the *bulid.xml* we can clean and run targets on other targets to get required and make our file executable. The main targets are cup and jflex or we can have dist and init for compiling since we clean it in build.xml. Therefore, we can run on all targets, or we can change the clean configuration in *bulid.xml*, and in addition, we can also set the source of the cup file and flex file.

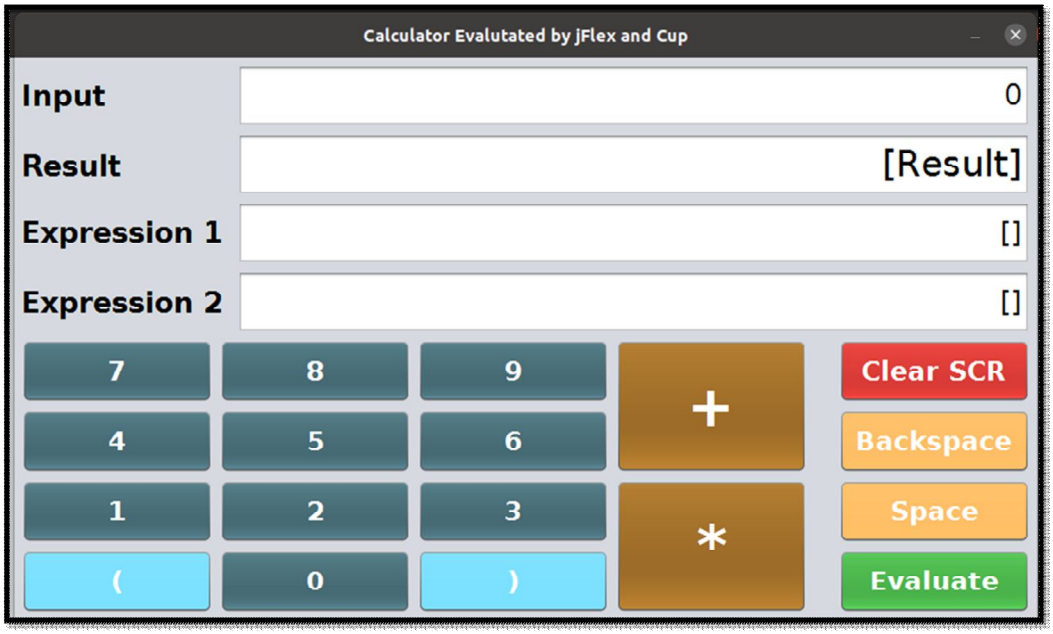


Figure 5. Running Calculator Screen

Figure 5 is the output after running the program where the main interface or page of the calculator is where we can see four text fields, operator buttons, operant buttons, and an evaluation button. The *input field* is for the user to input the expression such as infix notation or postfix notation or prefix notation. Depending on the input the next field, which is the Result field will display the result or value of expression given in the input. Then the next two fields, *Expression 1 and 2 fields* will display the expression (infix or postfix or prefix) which is two expression notations other than the input notation. Additional buttons are there to clear the screen, backspace, and space. The space button is used to input the expression such as postfix and prefix notation to have space between operator and operant. To understand more about how the calculator works, the following figures will help you to make things clear.

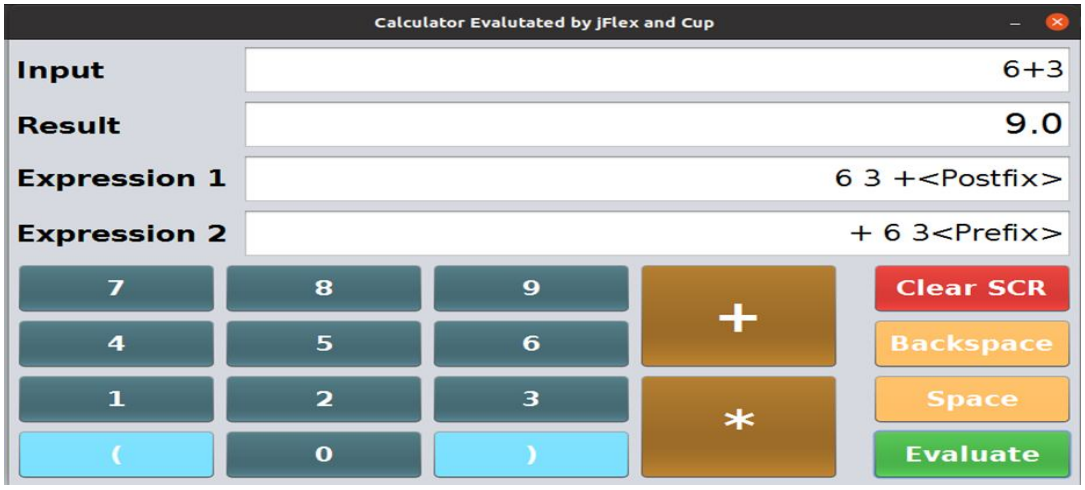


Figure 6. Infix Input

Figure 6 shows simple input like infix (like in math) displays the value of the expression and different expressions (postfix and prefix expression) of input.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 10 Issue VIII Aug 2022- Available at www.ijraset.com

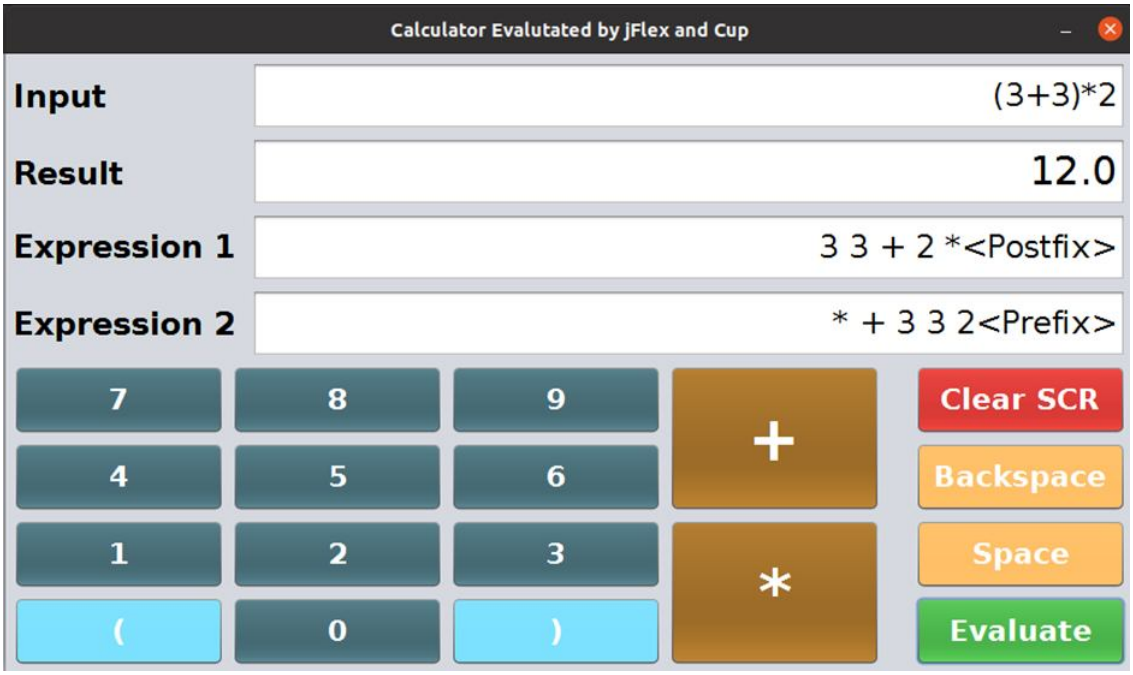


Figure 7. Input with parentheses ‘ (‘ and ‘) ‘

Figure 7 shows the priority of ( ) expression and calculation with it such expression.

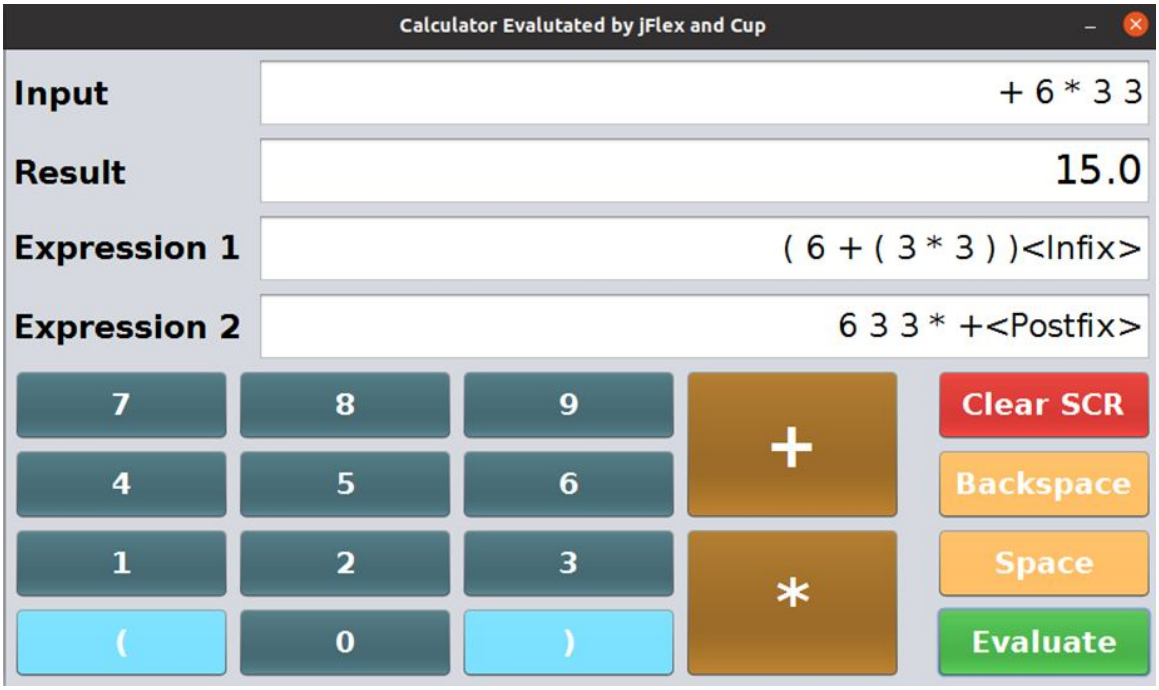


Figure 8. Prefix Input

Figure 8 shows that expression 1 and 2 depends on the input expression. The input is a prefix expression, then the output expression is of infix and postfix notation. If the input is an infix, then the output should be the value of the operation and show the prefix and postfix expression of it. Even if the input is prefix or postfix then the calculator should be able to show the infix expression and vice versa.

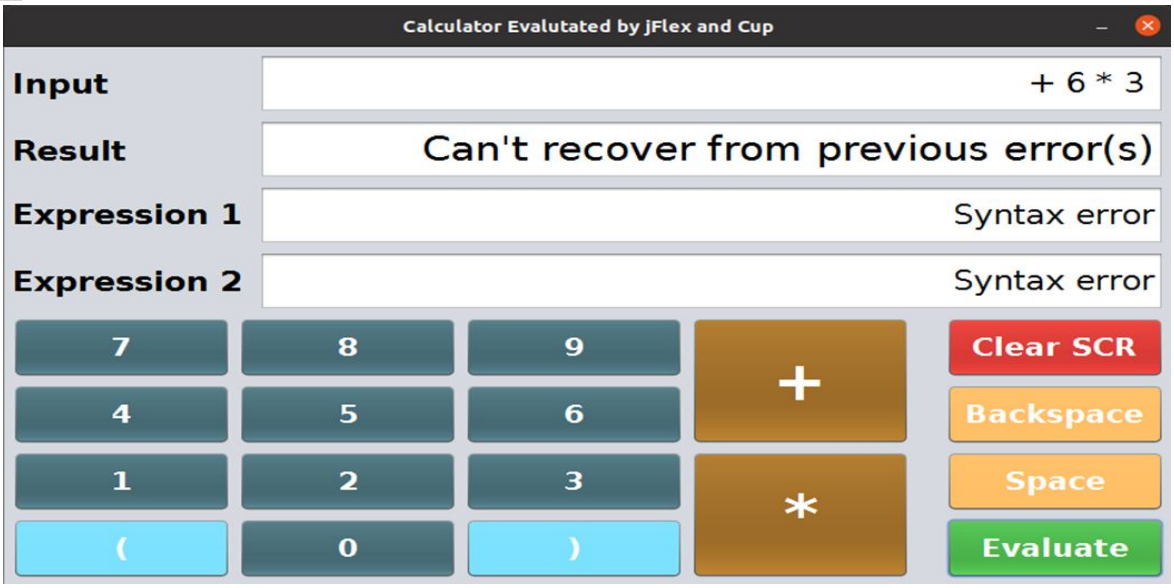


Figure 8. Syntax error

If the input is the wrong expression, then the error message is displayed as shown in figure 8. For example, if the input expression is not infix or postfix or prefix notation then the result and expressions 1 & 2 will have errors while we evaluate the input.

IV. IMPLEMENTATION AND RESULTS OF PROGRAMMING LANGUAGE

The main intention of this development is to create a small programming language that supports primitive types, arithmetic expressions, conditionals, and loops, and to realize the key compiler concepts like lexical analyzers, semantic analysis, and parse trees. The work was expanded to develop a calculator with an Addition and Multiplication operator. The Calculator was able to take integer input and compute the operation and display it.

A. The Developed Language Used The Terminal Symbols As Follows

TABLE II
Terminal Symbols

PLUS	MINUS	TIMES	DIVIDE	MOD	SEMI	COMMA	EQUALS
LPAREN	RPAREN	ET	NET	LT	LTE	GT	GTE
AND	OR	INT_LITERAL	FLOAT_LITERAL	BOL	STRING_LITERAL	IF	ENDIF
ELSE	WHILE	BEGIN	END	PRINT	PRINTLN	INT	FLOAT
BOOL	STRING	VAR					

B. These Are The Non-Terminal Symbols Used By The Language

TABLE III
Non-Terminal Symbols

program	declarations	declaration	statements	statement	type	assignment
ifelse	while	print	println	variables	expr	term
factor	relop					

C. Language Grammar Was Stated as Follows

```
program      := declarations statements | statements;
declarations := declarations declaration | declaration;
declaration  := type variables SEMI;
variables    := variables COMMA VAR | VAR;
statements   := statements statement | statement;
statement    := assignment SEMI | ifelse | while | print SEMI
                | println SEMI | BEGIN statements END;
ifelse       := IF LPAREN expr RPAREN statement ENDIF
                | IF LPAREN expr RPAREN statement ELSE statement ENDIF;
while        := WHILE LPAREN expr RPAREN statement;
print        := PRINT LPAREN expr RPAREN;
println      := PRINTLN LPAREN RPAREN;
type         := INT | FLOAT | STRING | BOOL;
assignment   := type VAR EQUALS expr | VAR EQUALS expr;
expr         := expr PLUS factor | expr MINUS factor | factor | replop;
factor       := factor TIMES term | factor DIVIDE term
                | factor MOD term | term;
relop        := term AND term | term OR term | term LT term | term LTE term | term GT term
                | term GTE term | term ET term | term NET term;
term         := LPAREN expr RPAREN | INT_LITERAL | FLOAT_LITERAL
                | STRING_LITERAL | VAR | BOL;
```

The Language like in [8], also consists of various tokens. Some tokens are keyword, variable, string literal, int literal, float literal, bool literal, and symbol. For example, the following statement consists of five tokens: print ('Hello World!');

D. All The Tokens That Our Language Supports Are As Follows

TABLE IV
Tokens

+	-	*	/	%	;	,	=
(	)	==	!=	<	<=	>	>=
&&	||	int	float	boolean	string	if	endif
else	while	{	}	print	println		

Semicolons act as a statement terminator. Each individual statement must be ended with a semicolon. However, if statement and while statement do not require a semicolon to mark the end.

An identifier is a name used to identify a variable. In this programming language, the identifier must start with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9). Punctuation characters are not allowed as identifiers. The identifiers are case-sensitive. So, Name and NAME are two different identifiers in our language. An identifier consists of alphanumeric characters. However, it may not start with digits but may start with any alphabetical character or underscore (_).

Whitespaces are required when the compiler requires to distinguish between keyword and variable name. However, in assignment statements, whitespaces can be ignored. These whitespaces improve the readability of the program.

Data types refer to a system used for declaring variables of different types. The types of data that our language supports are namely Integer (int), Floating Point (float), Boolean (boolean), and String (string).

TABLE V
Data Type

Data Type	Acceptable Value	Keyword
Integer	Decimal digits (integers)	int
Floating	Floating point numbers	float
Boolean	true or false	boolean
String (char)	alphabets, digits, and special characters enclosed in single quotes	string

E. Variable Definition

A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

- *type variable;*
- *type variable1, variable2, variable3;*

Here type means a valid data type specified in the token list. In the second declaration, three variables are defined type "type". By default, variables are initialized as 0, 0.0, a false and empty string for integer, float, boolean, and string types respectively. Variables can also be initialized and assigned an initial value in their declaration. The initializer consists of an equal's sign followed by a constant expression as follows:

- *type variable_name = value;*

F. Arithmetic Operators

The following table shows all the arithmetic operators supported by our language.

TABLE VI
Arithmetic Operators

Operator	Description
+	Adds two operands
-	Subtracts second operand from first
*	Multiplies two operands
/	Divides the first operand by the second operand
%	Modulus Operator shows the remainder of after division operation

A successful operation is carried out only if two operands are of the same type (Type Checking done). For example, operating on two different types of operands int and float results in an error as follows:

- *int number1 = 56; float number2 = 5.2;*
- *print(number1 + number2); // error message will be generated*

G. Relational Operators

The following table shows all the relational operators supported by our language.

TABLE VII
Relational Operators

Operator	Description
==	true if two operands are equal, else false
!=	true if two operands are not equal, else false
>	true if the first operand is greater than the second operand
>=	true if the first operand is greater than or equal to the second operand
<	true if the first operand is less than the second operand
<=	true if the first operand is less than or equal to the second operand

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 10 Issue VIII Aug 2022- Available at www.ijraset.com

H. Logical Operators

The following table shows two logical operators supported by our language.

TABLE VIII
Logical Operators

Operator	Description
&&	true if two operands are true, else false
||	true if one operand among two operands is true, else false

I. Assignment Operator

Our language supports assignment operators as follows:

TABLE IX
Assignment Operators

Operator	Description
=	Assigns values from right side operand to left side operand

J. Statements

Declaration of the single statement need not be enclosed in the curly braces but if multiple statements need to be executed, they need to be enclosed by "{"and "}".

K. Conditionals (If Else Endif)

Conditionals are decision-making structures that require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally other statements to be executed if the condition is determined to be false. This language supports "if statement" and "if-else statement". *"if" is to be ended by "endif"*. Multiple "if" statements can be nested to make a deeper level of decision-making.

L. Loop (While)

When the block of code needs to be executed several numbers of times, loops are used. Our language supports a while loop. While loop allows to run the block of statements until some condition is met. The programmer needs to be careful about the condition, if the condition is never met, the loop can run indefinitely. The syntax for while loop is shown below:

```
while ( boolean_expression ) {
statements;
statements;
}
```

M. User Interface

The user interface of this language is simple. There is a 'User Input' where the programmer writes codes. It is the only editable area. Output Console is not editable and is just meant to show the output. The 'Run Input' button runs the code. Output Console shows the output or errors if any. The 'Reset Input Output' button resets the whole application and puts it into the initial state. The 'Clear Console' button clears the Output Console box. Figure 9 will give a better understanding of the design.

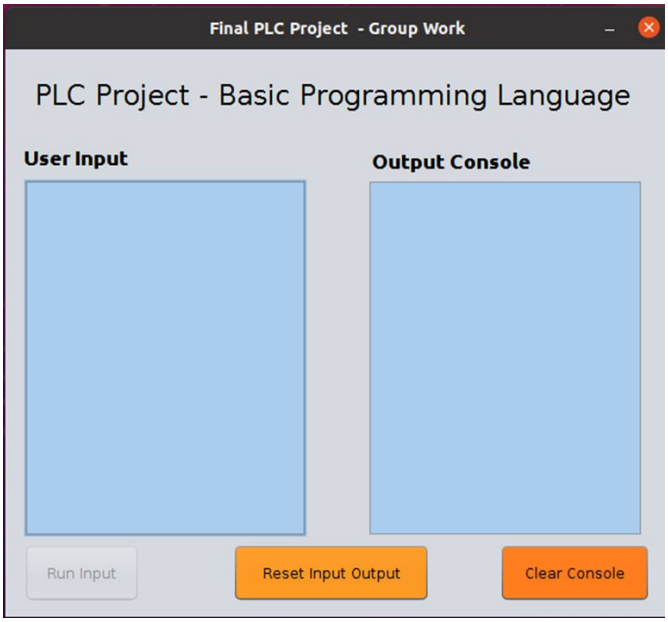


Figure 9. User Interface of Our Language

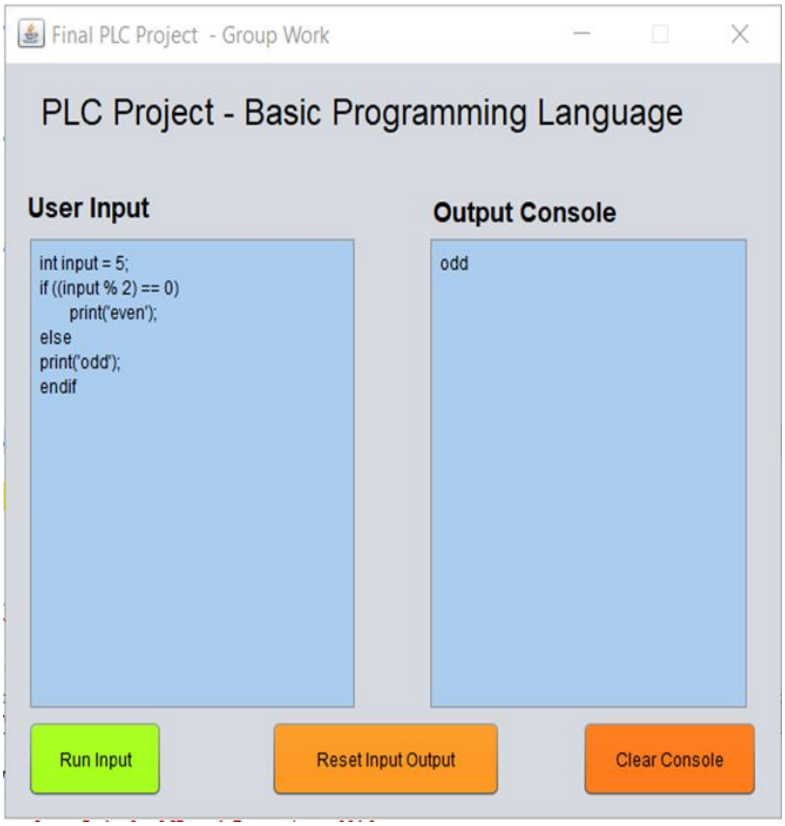


Figure 10. Simple if statement

In figure 10, it is *simple if statement* for checking the input number is odd or even using the relational operator and printing the result depending on the statement.

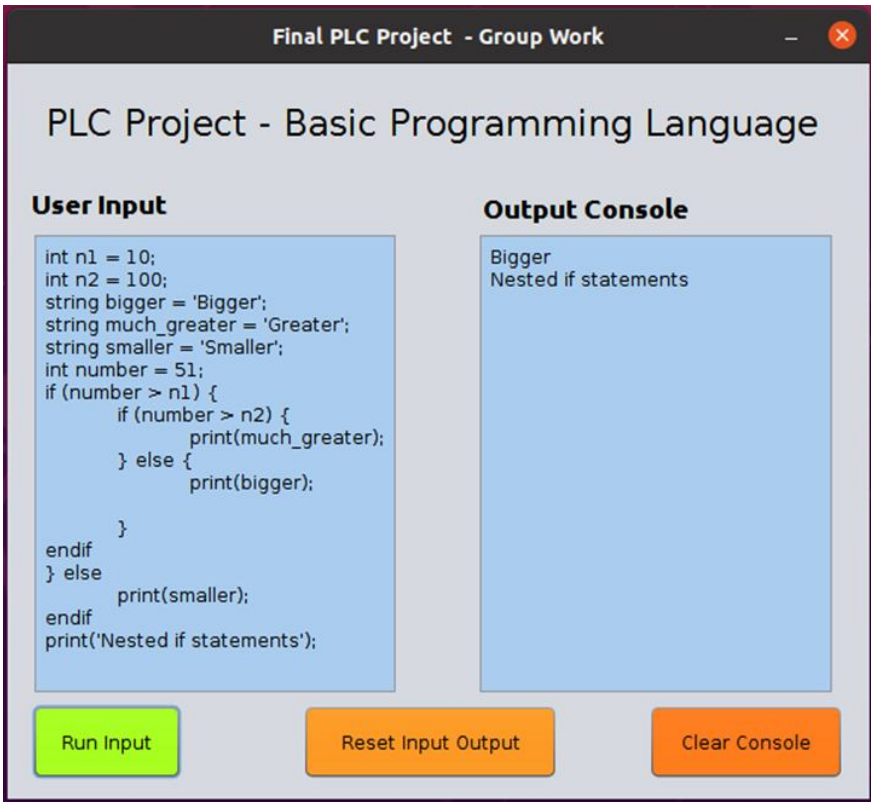


Figure 11. Executing Nested if statements program

Figure 11 shows the Nested if statements program, where we have defined the input number of n1, n2, and numbers with '10', '100'and '551' respectively in the 'User input'. Then we also define a string with bigger, much_greater, and smaller as 'Bigger', 'Greater', and 'Smaller' respectively. After that, we have our nested if statements, and their output is based on the condition being fulfilled.

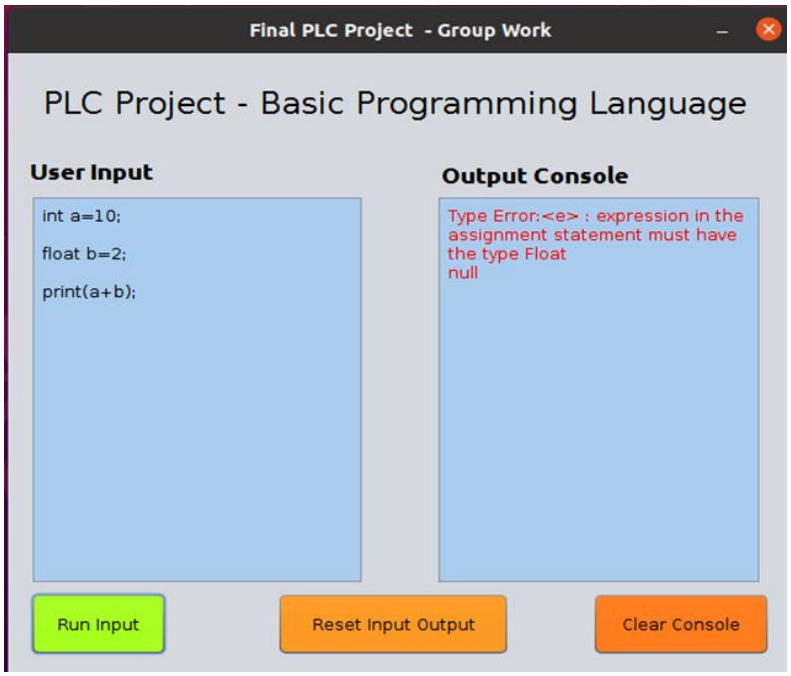


Figure 12. Showing type error in console. Errors are displayed in Red

In figure 12, we have tested the type checking and execution of the program. If error, it should show us the error message of type or execution error. In figure 12, we have initialized the wrong value to the datatype and executed the wrong statement of adding integer and float data types.

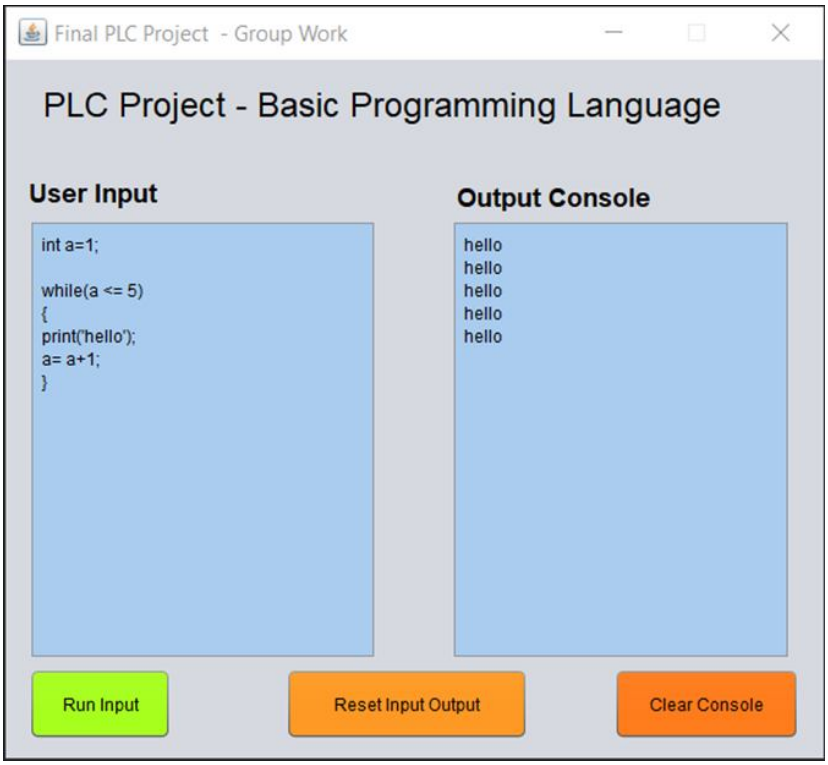


Figure 13. while statements program

Figure 13 shows our while loop running until the condition is failed and result according to it the output will be generated.

- To check more details on the calculator work with source code can be found in the link given below [7]: https://github.com/Younten-Tshering/Projects_and_related_works/tree/main/Calculator%20with%20Plus%20and%20Multiplication

- To check more details on the programming language developed with the user manual and source code of the work can be found in the link given below: https://github.com/Younten-Tshering/Projects_and_related_works/tree/main/Programming%20Language

V. CONCLUSIONS

The key concept used in this work was the execution of the grammar or rules in the cup. The parse tree records a sequence of rules the parser applies to recognize the input. In our grammar for calculator, the start symbol is “S”. It is the root of the parse tree. In the calculator, each interior node represents a non-terminal as represented in the grammar rule like statements, declarations (infix, postfix, and prefix), and operations. Each leaf node represents a token or a terminal symbol. The program is translated into Prefix Notation and Postfix Notation and displayed whenever the program is run.

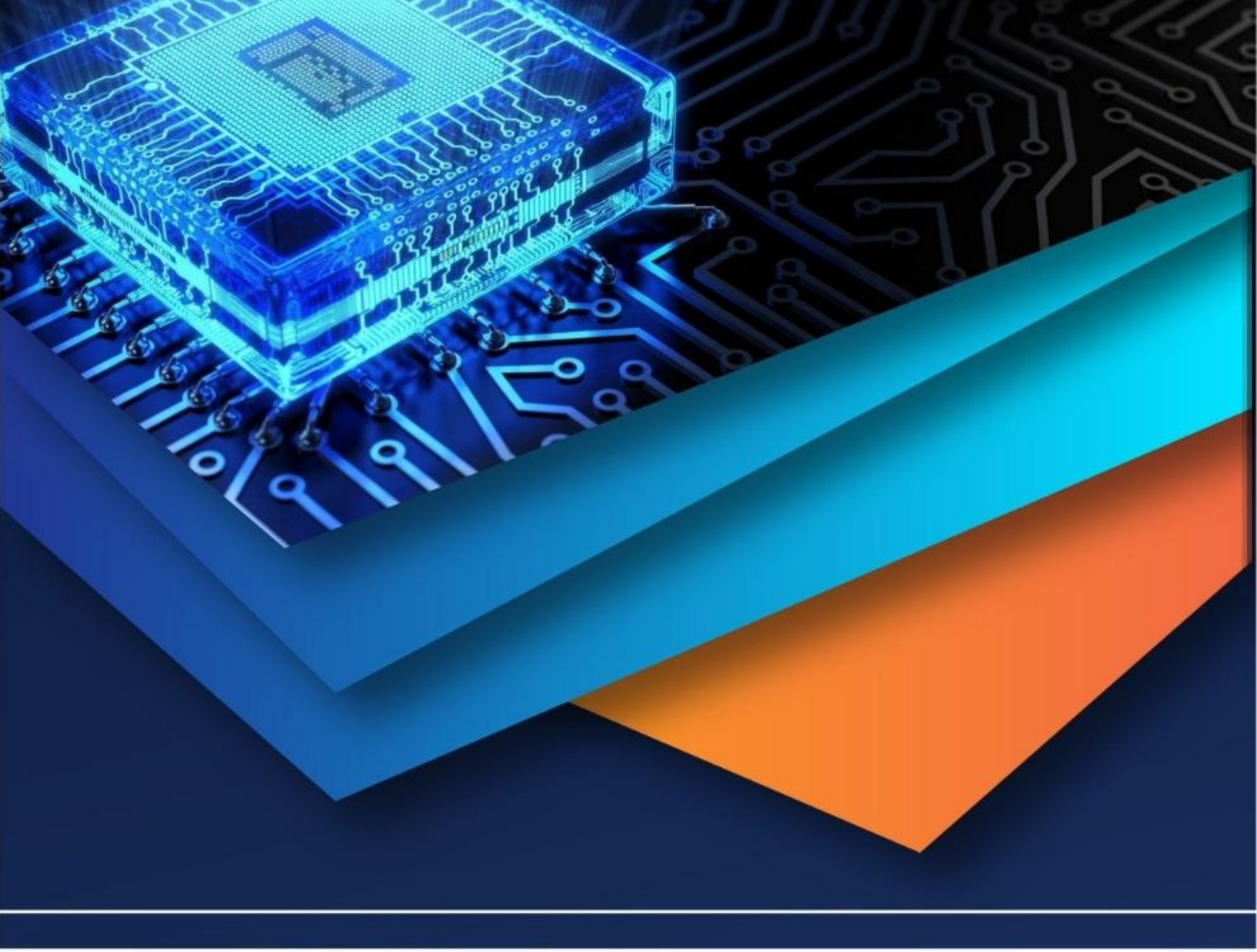
The execution of the statements in the language is based on Abstract Syntax Tree (AST). In the grammar for the programming language, the start symbol is “program”. It is the root of the parse tree. Each interior node represents a non-terminal as represented in the grammar rule like statements, declarations, variables, relop, expr, etc. Each leaf node represents a token or a terminal symbol. The variables declared are stored in the Environment Table. Environment Table is a HashTable that keeps track of variable names and the type that variable holds. The variable names are set as keys and whenever a new variable is declared, this key is checked upon and errors are shown accordingly. All the evaluations of the expressions and statements are recursively done. *Error checking and Type checking* are also done where two operands are checked for their compatibility with the operator and are shown if incompatible expressions are found.

VI. ACKNOWLEDGMENT

We would like to take this opportunity to thank Professor Phan Minh Dung, as the supervisor for his courage, support, help, and guidance throughout this work. We also would like to thank our dear parents, who have faith and support that we can do it. Finally, thank you to everyone who is involved directly or indirectly in providing a great contribution to this work.

REFERENCES

[1] Jflex (2020, May 3). What is it? Retrieved from https://jflex.de/
[2] M. E. Lesk and E. Schmidt. (n.d). Retrieved from Lex - A Lexical Analyzer Generator (wycwiki.readthedocs.io)
[3] A. Aho, R. Sethi, and J. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Publishing, Reading, MA, 1986.
[4] Apache NetBeans. (2020). The Apache Software Foundation. Retrieved from https://netbeans.apache.org/download/index.html
[5] Y. Tshering, S. Kamishetty and H. Sarwarzadah, (2021). Event Management for Social Service Website Using Ruby on Rails: BTO Event View Application Developed Implementing Collaborative Technique. World Journal of Research and Review. Volume-13. 01-09. 10.31871/WJRR.13.6.5.
[6] Wielenga, G. (2014, February 26). Top 5 features of NetBeans 8. Geertjan's Blog. Retrieved from https://blogs.oracle.com/geertjan/top-5-features-of-netbeans-8
[7] Y. Tshering, S. Tamrakar, and S. Gontia, (2021). Bid Buy-Sell system using client-server architecture, solution of concurrent users for the web application through optimistic locking and multithreading. Volume 9,. 2977-2986.
[8] T. Wangmo, (2021). Study on Bilingual Proficiency of Bhutanese Children: Technology Intervention for Language Preservation.
[9] Lexical Analysis using Jflex. (n.d). Retrieved from https://www.cs.auckland.ac.nz/courses/compsci330s1c/lectures/330ChaptersPDF/Chapt1.pdf



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY