

KASPY

(A python-based interpreting language)

A Project Report

Submitted by

**Aadesh Mallya
Kaushal Damania
Shivam Gulve**

Under the Guidance of

Prof. Ameyaa Biwalkar

*in partial fulfillment for the award of
the degree of*

BACHELOR OF TECHNOLOGY

IN

Computer Engineering

At



**NMIMS's Mukesh Patel School of Technology Management
& Engineering**

MONTH & YEAR

March, 2020

Annexure-II

DECLARATION

I, Aadesh Mallya_B054, Kaushal Damania_B065, Shivam Gulve_B066 of B.Tech (Computer Engineering), VI semester understand that plagiarism is defined as anyone or combination of the following:

1. Un-credited verbatim copying of individual sentences, paragraphs or illustration (such as graphs, diagrams, etc.) from any source, published or unpublished, including the internet.
2. Un-credited improper paraphrasing of pages paragraphs (changing a few words phrases, or rearranging the original sentence order)
3. Credited verbatim copying of a major portion of a paper (or thesis chapter) without clear delineation of who did wrote what. (Source: IEEE, The institute, Dec. 2004)
4. I have made sure that all the ideas, expressions, graphs, diagrams, etc., that are not a result of my work, are properly credited. Long phrases or sentences that had to be used verbatim from published literature have been clearly identified using quotation marks.
5. I affirm that no portion of my work can be considered as plagiarism and I take full responsibility if such a complaint occurs. I understand fully well that the guide of the seminar/ project report may not be in a position to check for the possibility of such incidences of plagiarism in this body of work.

Signature of the Students:

Names: Aadesh Mallya, Kaushal Damania, Shivam Gulve

Roll Nos. B054, B065, B066

Place: Mumbai, Parle

Date:

Annexure-III

CERTIFICATE

This is to certify that the project entitled “**KASPY Interpreting Language**” is the bonafide work carried out by **Aadesh Mallya, Kaushal Damania, Shivam Gulve**, of B.Tech MPSTME (NMIMS), Mumbai, during the VI semester of the academic year 2019-20, in partial fulfillment of the requirements for the Course Programming Laboratory III.

X

Prof. Ameyaa Biwalkar
Internal Mentor

X

Examiner 1

X

Examiner 2

TABLE OF CONTENTS

CHAPTER NO	TITLE	PAGE NO.
1.	INTRODUCTION	5
2.	SOFTWARE USED AND KEY CONCEPT	6-8
3.	METHODS IMPLEMENTED	9-10
4.	SCREENSHOTS	11-14
5.	CONCLUSION & FUTURE SCOPE	15
6.	SOCIETAL APPLICATION	16

CHAPTER 1: INTRODUCTION

1.1 INTRODUCTION

KASPYPY is an interpreter based on python. The idea was to explore the possibility of creating one's own programming language. The language consists of arithmetic operation execution, variable declaration, storing values in two different types of keywords and an additional feature that we all miss in python, the post-increment/decrement feature.

****The name KASPYPY is based on the first letters of the name of the developers of the language namely, **K**aushal/**A**adesh/**S**hivam and **PY** for the base language used.**

1.2 PROBLEM STATEMENT

To create a programming language using Python. The user should be able to declare variables, carry out arithmetic operations (add, subtract, multiply, divide, modulo, power), store the values in a variable, dynamically initialize a variable and refer its value later.

The following constraints have to be followed for input:

1. Variable can be declared by using only two keywords – kINT and kFLOAT.
2. Any other keyword used to declare variables will raise an error.
3. Illegal arithmetic operations such as Division by Error will raise an error.
4. All arithmetic operations will be executed according to the BODMAS rule and the grammar defined for the language.
5. Multiple additions can be made to the language as more features are added to the language.
6. Decimal values if stored in a kINT variable will automatically get truncated /floored and it's the integer value will be stored.
7. Integer values if stored in a kFLOAT variable will automatically be converted to a decimal value by adding 2 decimal places to the number and that number will be stored.

CHAPTER 2: SOFTWARE USED AND KEY CONCEPT

2.1 SOFTWARE USED

KASPY is built purely using Python 3. The entire language is based on using the concept of implementing the tree data structure in python.

IDLE:

- IDLE (Integrated Development and Learning Environment) is an integrated development environment (IDE) for Python.
- Coded purely in Python 3.
- Cross-platform: works mostly the same on Windows, Unix, and Mac OS X
- Python shell window (interactive interpreter) for code input, output, and error messages
- Multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- Debugger with persistent breakpoints, stepping, and viewing of global and local namespaces

2.1 KEY CONCEPT

Making one's own interpreter involves teaching the compiler to understand the language of the code input by the user. To understand the entire input, the entire language.

Consider an input "6+9"

Tokens

When you enter an expression "6+9" on the IDLE your interpreter gets a string "6+9". In order for the interpreter to actually understand what to do with that string it first needs to break the input "6+9" into components called **tokens**. A **token** is an object (which is an instance of a class) that has a type and a value. For

example, for the string “6” the type of the token will be INTEGER and the corresponding value will be integer 3. Token generation will be as follows:

Token (6, INT) / Token (+, PLUS) / Token (9, INTEGER)

Parser

The interpreter now needs to understand the underlying structure in the stream of tokens, or put in another way, recognize a phrase in the entire stream of tokens. This process is called parsing. And the part of the interpreter that executed this process is called a parser. The parser basically does the job of making an abstract syntax tree (namely, the AST).

Need for an AST:

The input above was pretty basic “6+9”. We have two integers and both are connected by a ‘+’ symbol which means that they are to be added. However, when multiple arithmetic operations are included in a single input, the interpreter needs to understand the syntax of the input, and arrange execution based on the priority of different arithmetic operators used. In order to analyze such complex language constructs, we need to build an intermediate representation (IR). Our parser will be responsible for building an IR and our interpreter will use it to interpret the input represented as the IR.

As it turns out, a tree is a very suitable data structure to denote the syntax of these languages.

Considering this input for e.g. :1+2*3

Clearly, ‘*’ has a higher priority than ‘+’ as per BODMAS rule of mathematics. The tree will be generated as:

[[INT:1], [PLUS: ‘+’], [[INT:2], [PLUS: ‘*’], [INT:3]]]

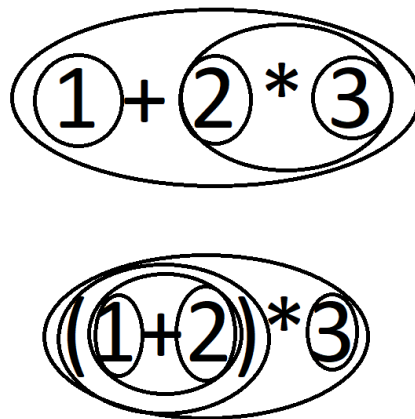
However, if we change the input to: (1+2) * 3

Now, the interpreter has to understand that the parenthesis and the expression inside exceeds the priority of the ‘*’ operator inside.

LANGUAGE:

To enable the interpreter to understand the syntax of the input, we have divided the entire input language into parts based on priority.

```
expr  : KEYWORD: VAR IDENTIFIER EQUALS expr
       : term ((PLUS|MINUS) term) *
term   : factor ((MUL|DIV) factor) *
factor : (PLUS|MINUS) factor
power  : root (POW factor) *
root   : INT|FLOAT
       : LPAREN expr RPAREN
```



*** FIG: Drawn by us in paint for explanation*

The above figure is used to denote the priority of every operator and how the interpreter understands the syntax of the input.

Every part of the language and the terms defined above will be explained in detail in the next chapter.

CHAPTER 3: METHODS IMPLEMENTED

3.1 kaspy.py file

This file is the heart of the entire interpreter. All the methods implemented here are as follows:

TOKENS:

- `__init__`: to initialize the token type and its value
- `matches()`: to compare the token type and its value and check its validity

LEXERS:

- `make_tokens`: to return an array of TOKENS class objects each containing the token type and its value.
- `make_number`: for poly-digit numbers, we can't extract every single digit. This method iterates till a whitespace is encountered to get the complete number, be it a float or an integer value.
- `make_identifier`: like number, to get a keyword of multiple letters, extracting each letter doesn't make sense. This method iterates to extract the entire word as a token type to check for a keyword or a variable.

PARSER:

- `expr()`: this method is to start assigning nodes to the entire node. This method has the lowest priority i.e. this method has to be executed at the end. The highest priority being to navigate to the parenthesis expression (if it exists) and then to the individual number nodes and finally the assignment. This method verifies for existence of a keyword, variable and a mandatory equal sign if the former two exist. The keyword and the variable are assigned a node and the expression evaluation begins.
- `factor()`: the addition/subtraction operator between two number nodes or two expression is called a factor. This has the second lowest priority.
- `power()`: the power(raised to) operator between two number or a number and a factor is called the power. This has the third lowest priority.
- `term()`: the multiplication/division operator between two number nodes (roots) or two factors has the second highest priority and is called term.

- `root()`: this term is used for declaration of number nodes(object of NumberNode class) and has the highest priority. It also includes the expression inside parenthesis. Every expression recursively calls the next method which has a higher priority than it, till it reaches the root.
- `bin_op()`: this is the basis for creation of the abstract syntax tree. A node is created for the right and the left, each of which contains the value of the root of the left, the operator type and the value of the right root.

NUMBER:

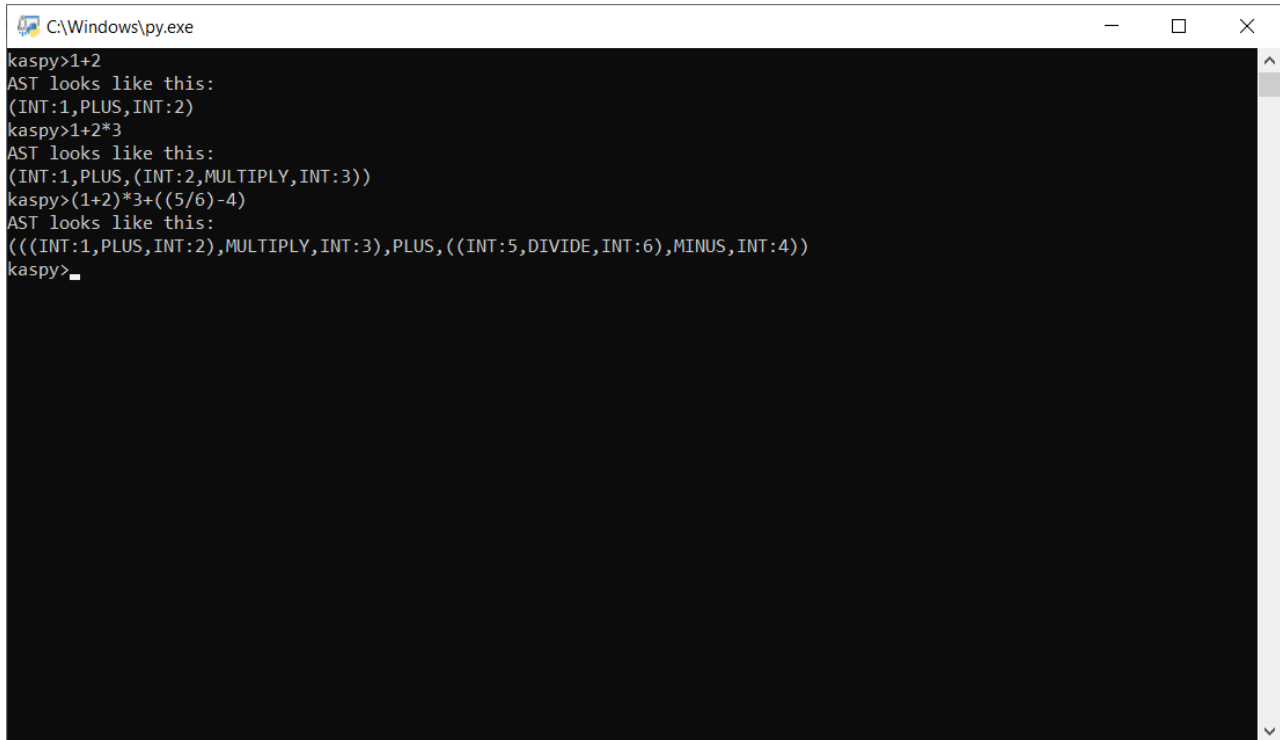
- `add_to()`: returns the arithmetically added value of the values of the two number nodes.
- `sub_from()`: returns the arithmetically subtracted value of the values of the two number nodes.
- `mul_by()`: returns the arithmetically multiplied value of the values of the two number nodes.
- `div_by()`: returns the arithmetically divided value of the values of the two number nodes.
- `raised_to()`: returns the arithmetically powered value of the values of the two number nodes.
- `mod_of()`: returns the arithmetically modulus value of the values of the two number nodes.

INTERPRETER:

- `visit_NumberNode()`: to create an object of the value obtained from the NumberNode.
- `visitkINTAccessNode()`: to get value of the previously declared variable from the global symbol table.
- `visit_kINTAssignNode()`: to assign value to a variable by getting value of the executed expression stored in a Number Node.
- `visit_BinOpNode()`: this is where the arithmetic operation of two number nodes are carried out and stored in the result which later gets printed. This method invokes assign node if a variable is included, access node if a variable's value is being called in the expression.

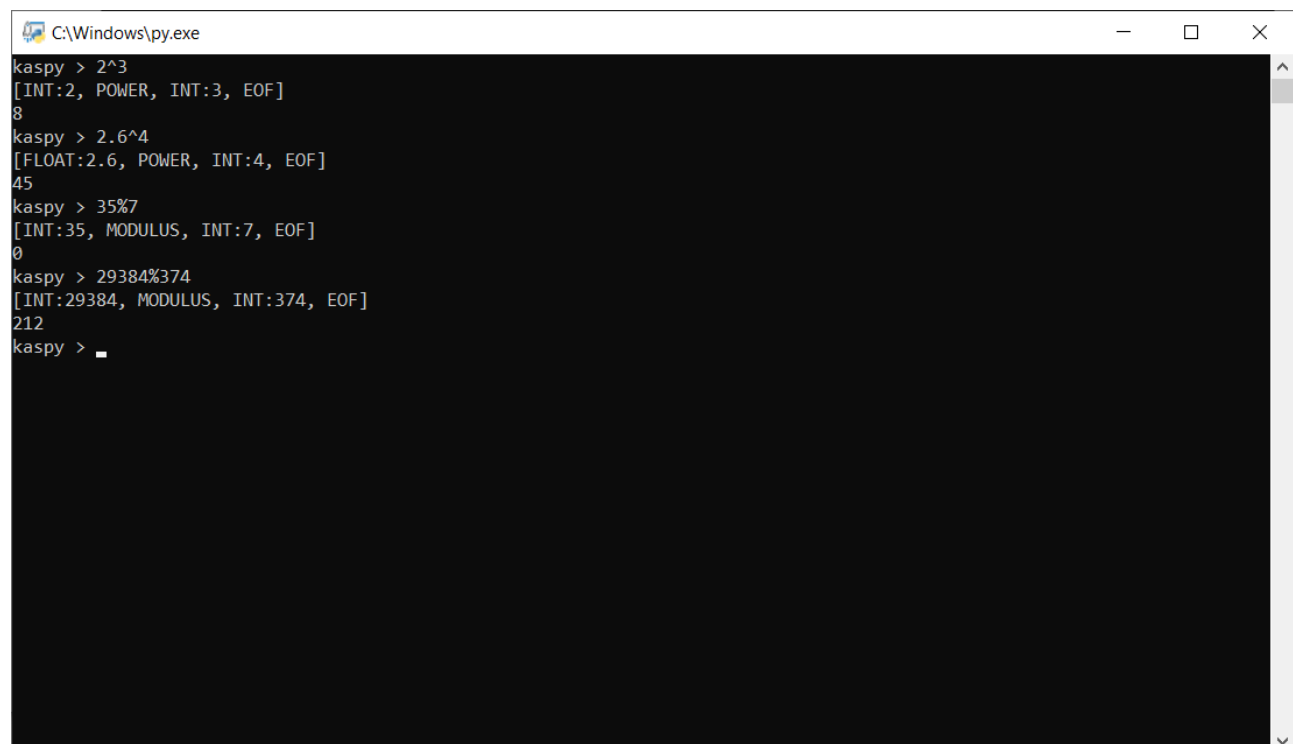
CHAPTER 4: SCREENSHOTS

1. Abstract Syntax Tree for Simple Arithmetic Operation



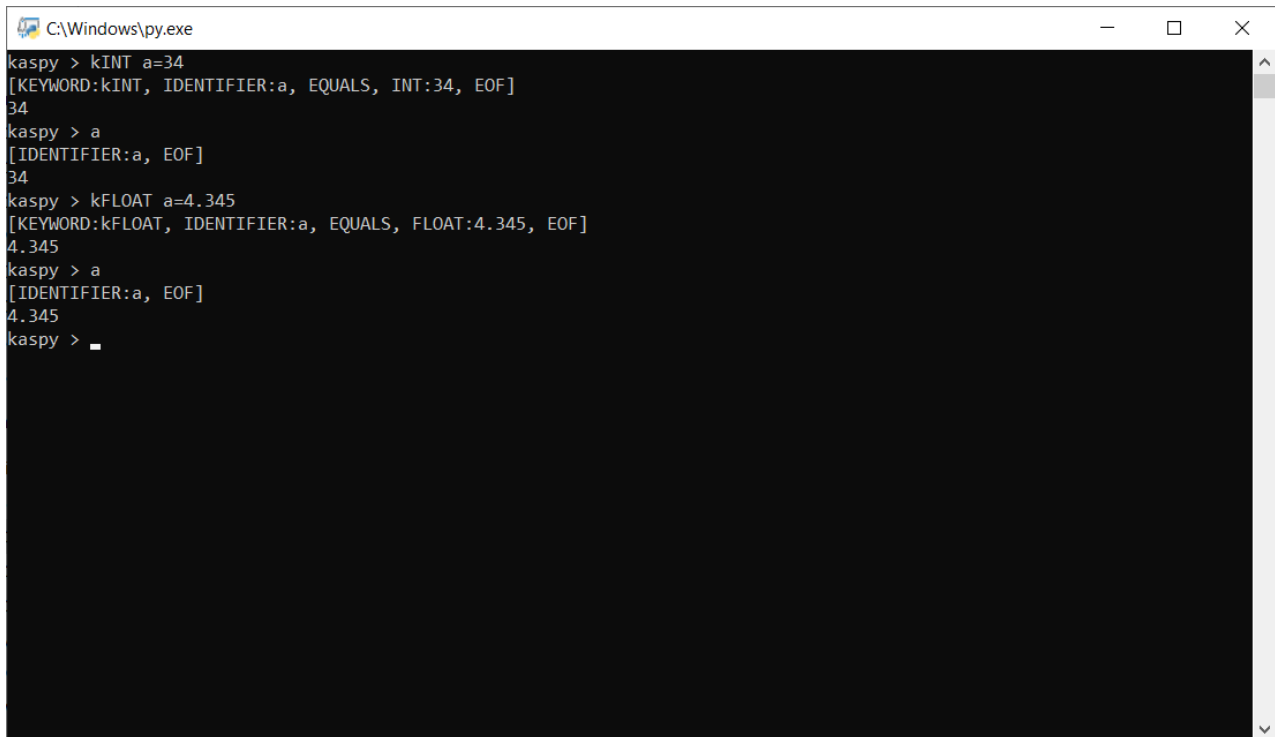
```
C:\Windows\py.exe
kaspy>1+2
AST looks like this:
(INT:1,PLUS,INT:2)
kaspy>1+2*3
AST looks like this:
(INT:1,PLUS,(INT:2,MULTIPLY,INT:3))
kaspy>(1+2)*3+((5/6)-4)
AST looks like this:
(((INT:1,PLUS,INT:2),MULTIPLY,INT:3),PLUS,((INT:5,DIVIDE,INT:6),MINUS,INT:4))
kaspy>
```

2. Some more arithmetic operations



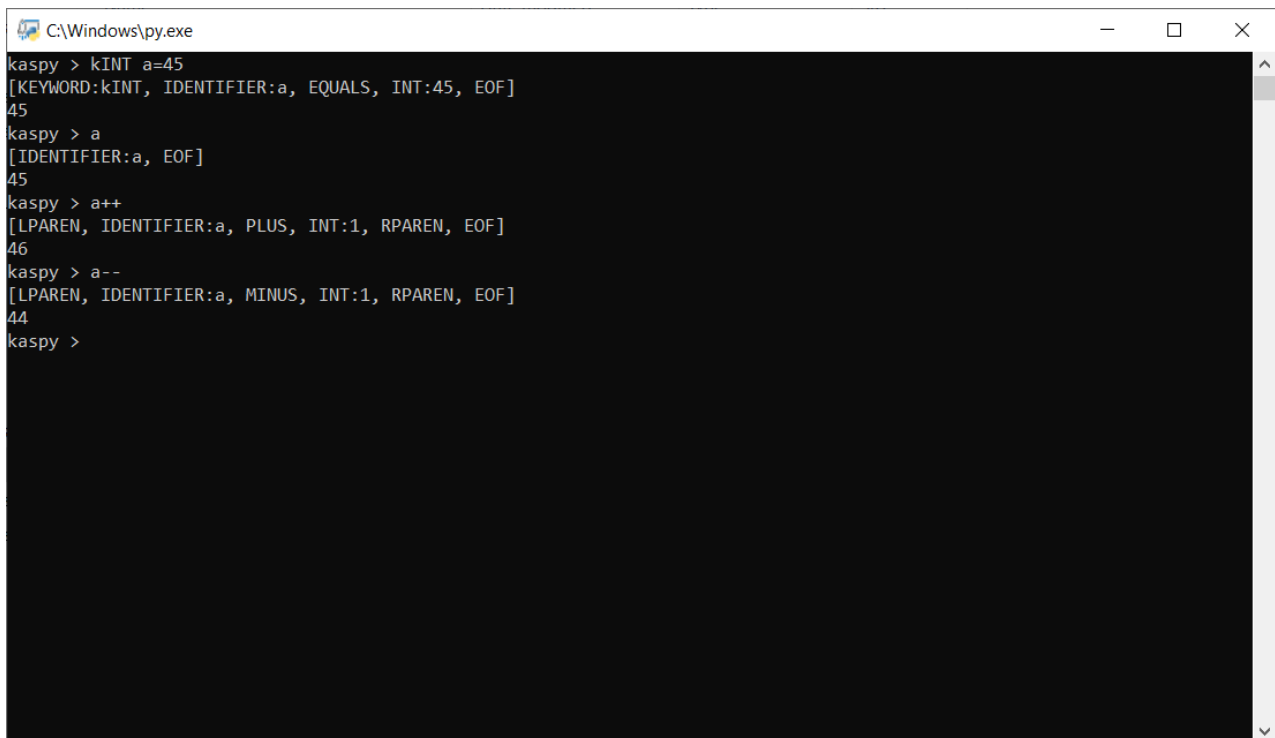
```
C:\Windows\py.exe
kaspy > 2^3
[INT:2, POWER, INT:3, EOF]
8
kaspy > 2.6^4
[FLOAT:2.6, POWER, INT:4, EOF]
45
kaspy > 35%7
[INT:35, MODULUS, INT:7, EOF]
0
kaspy > 29384%374
[INT:29384, MODULUS, INT:374, EOF]
212
kaspy >
```

3. Variable Declaration



```
C:\Windows\py.exe
kaspy > kINT a=34
[KEYWORD:kINT, IDENTIFIER:a, EQUALS, INT:34, EOF]
34
kaspy > a
[IDENTIFIER:a, EOF]
34
kaspy > kFLOAT a=4.345
[KEYWORD:kFLOAT, IDENTIFIER:a, EQUALS, FLOAT:4.345, EOF]
4.345
kaspy > a
[IDENTIFIER:a, EOF]
4.345
kaspy > _
```

4. Post Increment/Decrement



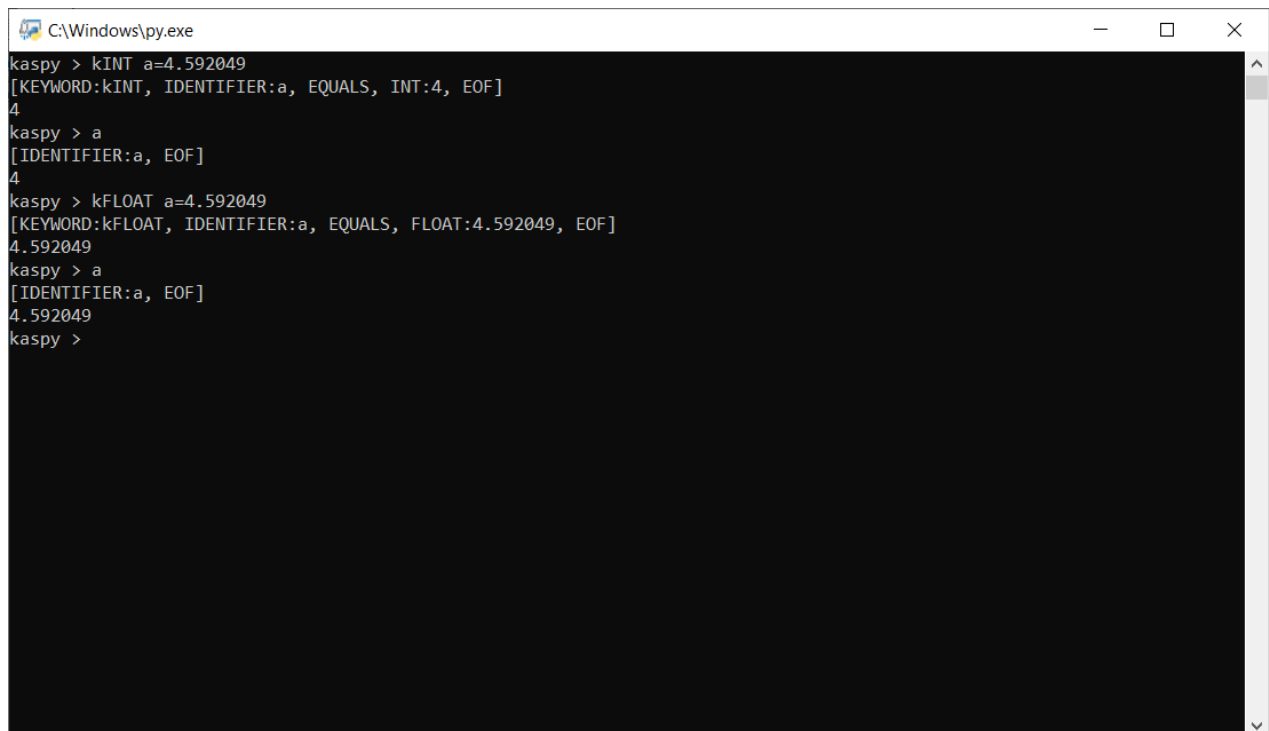
```
C:\Windows\py.exe
kaspy > kINT a=45
[KEYWORD:kINT, IDENTIFIER:a, EQUALS, INT:45, EOF]
45
kaspy > a
[IDENTIFIER:a, EOF]
45
kaspy > a++
[LPAREN, IDENTIFIER:a, PLUS, INT:1, RPAREN, EOF]
46
kaspy > a--
[LPAREN, IDENTIFIER:a, MINUS, INT:1, RPAREN, EOF]
44
kaspy >
```

```
C:\Windows\py.exe
kaspy > 3+4
[INT:3, PLUS, INT:4, EOF]
7
kaspy > 3++4
[INT:3, PLUS, PLUS, INT:4, EOF]
7
kaspy > 3++
[LPAREN, INT:3, PLUS, INT:1, RPAREN, EOF]
4
kaspy > ++3
[PLUS, PLUS, INT:3, EOF]
3
kaspy > 3-4
[INT:3, MINUS, INT:4, EOF]
-1
kaspy > 3--
[LPAREN, INT:3, MINUS, INT:1, RPAREN, EOF]
2
kaspy > 3--4
[INT:3, MINUS, MINUS, INT:4, EOF]
7
kaspy > 3---4
[INT:3, MINUS, MINUS, MINUS, INT:4, EOF]
-1
kaspy >
```

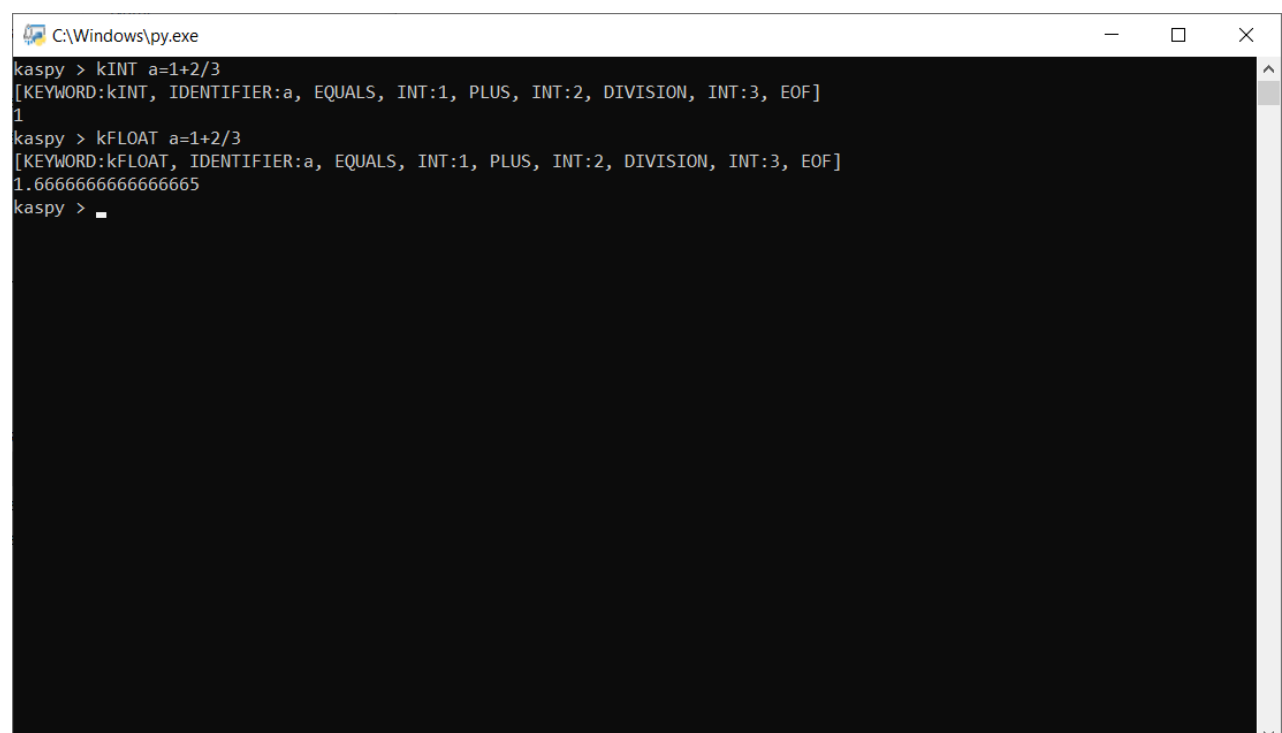
5. Dynamic Initialization of variables

```
C:\Windows\py.exe
kaspy > 34 + (kINT a=45)
[INT:34, PLUS, LPAREN, KEYWORD:kINT, IDENTIFIER:a, EQUALS, INT:45, RPAREN, EOF]
79
kaspy > a
[IDENTIFIER:a, EOF]
45
kaspy > a + (kFLOAT b=4.56)
[IDENTIFIER:a, PLUS, LPAREN, KEYWORD:kFLOAT, IDENTIFIER:b, EQUALS, FLOAT:4.56, RPAREN, EOF]
49.56
kaspy >
```

6. Value truncation based on keyword used



```
C:\Windows\py.exe
kaspy > kINT a=4.592049
[KEYWORD:kINT, IDENTIFIER:a, EQUALS, INT:4, EOF]
4
kaspy > a
[IDENTIFIER:a, EOF]
4
kaspy > kFLOAT a=4.592049
[KEYWORD:kFLOAT, IDENTIFIER:a, EQUALS, FLOAT:4.592049, EOF]
4.592049
kaspy > a
[IDENTIFIER:a, EOF]
4.592049
kaspy >
```



```
C:\Windows\py.exe
kaspy > kINT a=1+2/3
[KEYWORD:kINT, IDENTIFIER:a, EQUALS, INT:1, PLUS, INT:2, DIVISION, INT:3, EOF]
1
kaspy > kFLOAT a=1+2/3
[KEYWORD:kFLOAT, IDENTIFIER:a, EQUALS, INT:1, PLUS, INT:2, DIVISION, INT:3, EOF]
1.6666666666666665
kaspy > █
```

CHAPTER 5: CONCLUSION AND FUTURE SCOPE

KASPY is not merely an interpreter for us. It was like looking into the heart of the system. Understanding how the system works, executing a code that we enter, was one of the greatest learning experiences for us. We learned about the different processes that go into the interpretation of a high-level language before it is converted into machine level language by the compiler.

Starting from lexical analysis, moving on parsing, abstract syntactic tree creation and interpretation of the tree to derive the answer, the project has been very an exciting for my group.

As for future scope, we have decided to add the following features to make the language a more complete language:

1. Loops (FOR and WHILE loops)
2. Conditional Statements (IF-ELSE, SWITCH)
3. Function Declaration
4. Multiline Statements
5. Built-In Functions defined in KASPY language

As mentioned in the next chapter, we can add a graphical visualization of the parse tree generation which will not only help understand the working of the interpreter better but spark an interest in people who were previously unable to grasp the execution sequence of code.

****A website can be created on which a window can be provided that will allow users to write a sample code in KASPY language and see the live parse tree generation and understand how his/her code is getting executed.**

CHAPTER 6: SOCIETAL APPLICATION

It has been observed however, that not many people know/understand the working of a compiler/interpreter. We looked for a way to visualize the entire process, namely lexical analysis and the creation of a parse tree and found that a package namely, graphviz can be used to visualize the entire parse tree.

We found a project, namely genptdot.py which follows a similar approach to ours to build a parse tree except it is used to visualize instead of getting executed to get an answer. The graphviz package creates a parseetree.png file which can be opened to see the parse tree.

Thus, in this way, people can learn more about the interpreter by visually seeing the parse tree getting created. It will not only help understand the working of the interpreter better but spark an interest in people who were previously unable to grasp the execution sequence of code.