VIPER, A New
Programming Language

An Interpreter

Authored by:
Rafid Islam (170042009)
Ratun Rahman (170042011)
Akib Ahmed (170042013)

Abstract—

In this paper, an interpreter design and implementation for a small subset of Python Language using software engineering concepts are presented. This paper reinforces an argument for the application of software engineering concepts in the area of interpreter design but it also focuses on the relevance of the paper to undergraduate computer science curricula. The design and development of the interpreter is also important to software engineering. Some of its components form the basis for different engineering tools.

Also it is important to mention that, this language is based on **Bangla** language. English speaker might be unknown with some terms.

I. DESCRIPTION AND MOTIVATION

"When you don't create things, you become defined by your tastes rather than ability. Your tastes only narrow & exclude people. So create."
- Why the Lucky Stiff

In this paper, an interpreter design and implementation for a small subset of Python Language using software engineering concepts are presented. This paper summarizes the development process used, detail its application to the programming language to be implemented and present a number of metrics that describe the evolution of the project. Incremental development is used as the software engineering approach because it interleaves the activities of specification, development, and validation. The system was developed as a series of versions (increments) where each version adds functionality to the previous version. The paper will also focus on the relevance of compilers and interpreters to undergraduate computer science curricula. Interpreters and compilers represent two traditional but fundamentally different approaches to implementing programming languages. A correct understanding of the basic mechanisms of each is an indispensable part of the knowledge that every computer science student must acquire. The paper is organized as follows: section II presents the background and related work, and section III describes the design and development process, section IV the code example. The conclusions and future work are discussed in section V.

II.   BACKGROUND AND RELATED WORK

A .   BACKGROUND

The main purpose of a compiler or an interpreter is to translate a source program written in a high-level source language to machine language. The language used to write the compiler or interpreter is called implementation language. The difference between a compiler and an interpreter is that a compiler generates object code written in the machine language and the interpreter executes the instructions. A utility program called a linker combines the contents of one or more object files along with any needed runtime library routines into a single object program that the computer can load and execute. An interpreter does not generate an object program. When you feed a source program into an interpreter, it takes over to check and execute the program. Since the interpreter is in control when it is executing the source program, when it encounters an error it can stop and display a message containing the line number of the offending statement and the name of the variable. It can even prompt the user for some corrective action before resuming execution of the program.

B .   RELATED WORK

The process is divided into **6 functional increments**. Before moving to the next increment, the current increment has to be tested and validated. The increments are:
1. The framework
2. The scanner
3. The symbol table
4. Parsing and interpreting expressions and assignment statements
5. Parsing and interpreting control statements
6. Parsing and interpreting declarations.

III.   DESIGN AND IMPLMENTATION

A .   CONCEPTUAL DESIGN

The conceptual design of a program is a high-level view of its software architecture. The conceptual design includes the primary components of the program, how they are organized, and how they interact with each other. An interpreter is classified as a programming language translator. A translator, as seen at the highest level, consists of a front end and a back end. Fig. 1 shows the conceptual design of the Simple interpreter. The front end of a translator reads the source program and performs the initial translation stage. Its primary components are the parser, the scanner, the token, and the source.

The parser controls the translation process in the front end. It continuously asks the scanner for the next token, and it analyzes the sequences of tokens to determine what high-level language elements it is translating. The parser verifies that what it sees is syntactically correct as written in the source program; in other words, the parser detects and flags any syntax errors. The scanner reads the characters of the source program sequentially and constructs tokens, which are the low-level elements of the source language. The scanner scans the source program to break it apart into tokens.
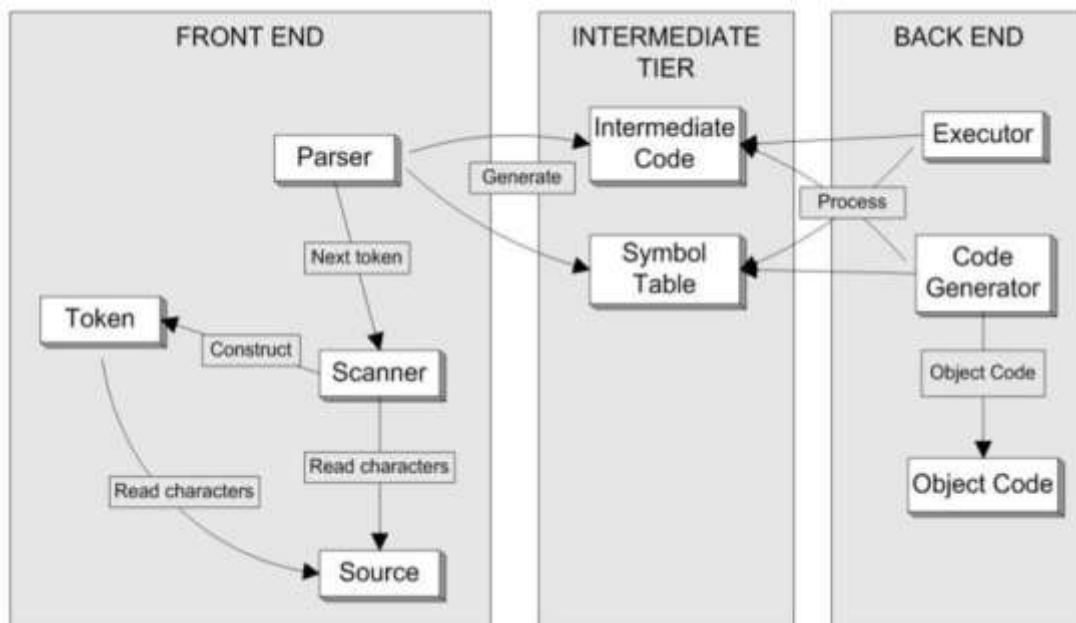
Fig. 1. Conceptual Design of the Simple Interpreter

Syntax and Semantics

The syntax of a programming language is its set of grammar rules that determine whether a statement or an expression is correctly written in that language. The language's semantics give meaning to a statement or an expression. In Simple python, the statement:

$$a = b + c$$

is a valid assignment statement. The semantics of the language tells that the statement says to add the value of variables "b" and "c" and assign the sum's value to the variable "a". A parser performs actions based on both the source language's syntax and semantics. Scanning the source program and extracting tokens are syntactic actions. Looking for "=" token is a syntactic action, entering the identifiers "a", "b", and "c" into the symbol table as variables, or looking them up in the symbol table, are semantic actions because the parser had to understand the meaning of the expression and the assignment to know that it needs to use the symbol table. Syntactic actions occur in the front end, while semantic actions can occur on either the front end or the back end.

B . BASIC INTERPRETER FRAMEWORK

The goals for this increment are:

1. A source language-independent framework (python) that can support interpreters
2. Initial source language-specific components integrated into the front end of the framework
3. Initial interpreter components integrated into the back end of the framework
4. Simple end-to-end runs that exercise the components by generating source program listings from the common front end and messages from the interpreter back end

As mentioned previously, the project will be divided into functional increments, using software engineering concepts.
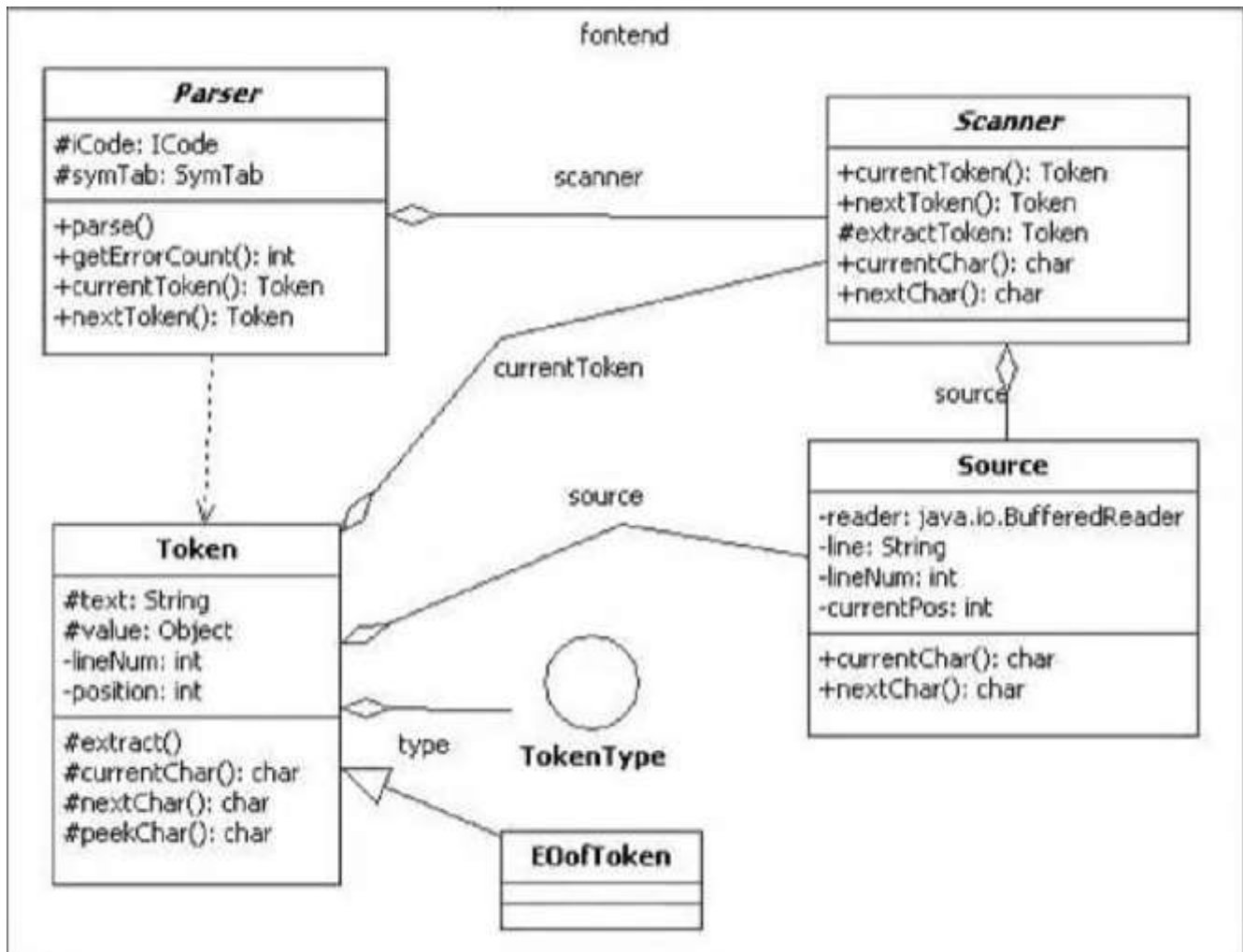
C. SCANNING

The second increment of the project consists on implementing a scanner. The scanner is the component in the front end of a compiler or an interpreter that performs the syntactic actions of reading the source program and breaking it apart into tokens. The parser calls the scanner each time it wants the next token from the source program. The goals for this increment are:

1. Complete the design and development of the scanner.
2. The scanner should be able to:
   a. Extract words, numbers, and special symbols from the source program
   b. Determine whether a word is an identifier or a reserved word
   c. Calculate the value of a number token and determine whether its type is integer or real

3. Perform syntax error handling
    a. Syntax Error Handling: Every parser must be able to handle syntax errors in the source program. Error handling is a three step process:
        i. Detection: Detect the presence of a syntax error.
        ii. Flagging: Flag the error by pointing it out or highlighting it, and display a descriptive error message.
        iii. Recovery: Move past the error and resume parsing.
    If the scanner finds syntax errors (such as an invalid character that cannot start a legitimate token), it will construct an Error

D. THE SYMBOL

The parser of a compiler or an interpreter builds and maintains a symbol table throughout the translation process as part of semantic analysis. The symbol table stores information about the source program's tokens, mostly the identifiers. Goals for this increment:

1. A language-independent symbol table
2. A simple utility program that parses a source program and generates a cross-reference listing of its identifiers

E. EXPRESSIONS AND ASSIGNMENT STATEMENTS

The goals for this increment are:

1. Parsers in the front end for certain constructs: assignment statements, compound statements, and expressions.
2. Flexible, language-independent intermediate code (Python) generated by the parsers to represent these constructs.
3. Language-independent executors in the interpreter back end that will interpret the intermediate code and execute expressions and assignment statements.

F. CONTROL STATEMENTS

The next increment focuses on parsing and interpreting control statements. The goals for this increment are:

1. Parsers in the front end for control statements if, while, and for.

2. Flexible, language-independent intermediate code generated by the parsers to represent these constructs.

3. Reliable error recovery to ensure that the parsers can continue to work despite syntax errors in the source program. The syntax diagrams shown in Fig. 11 were used to guide the development of the parsers.
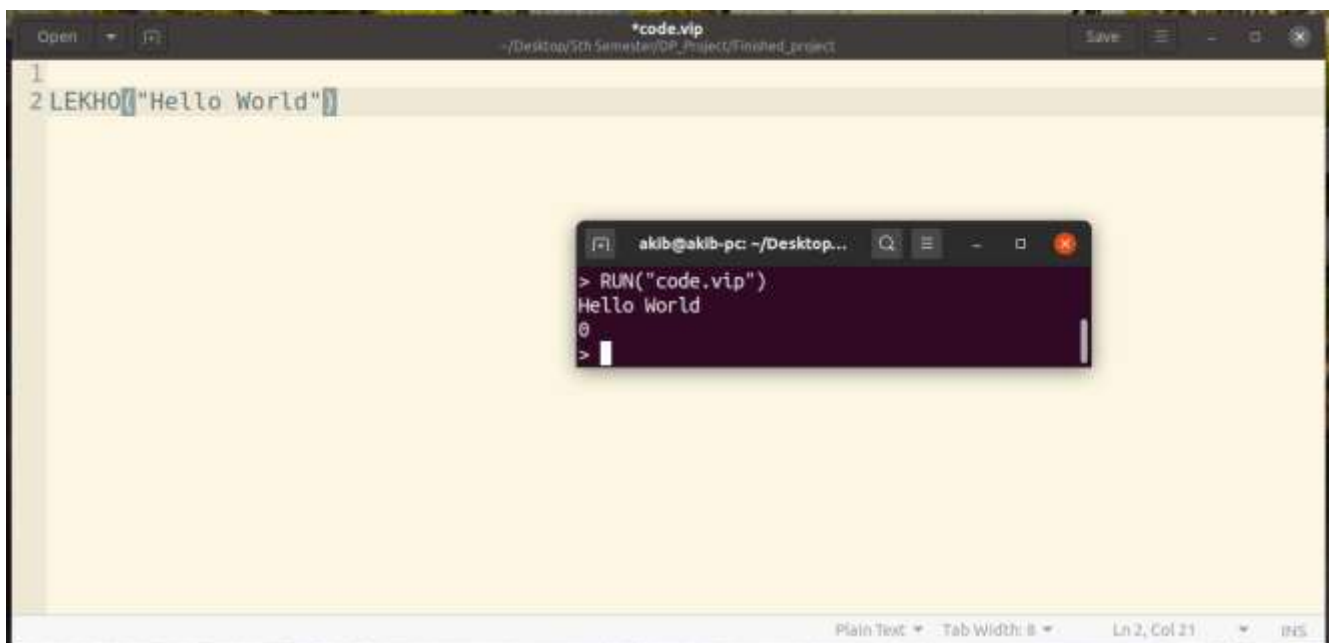
G . PARSING DECLARATIONS

Parsing declarations expands the work in The Symbol Table increment because all the information from the declarations has to be entered in the symbol table. The goals for this increment are:

1. Parsers in the front end for type definitions and type specifications.
2. Additions to the symbol table to contain type information (END)
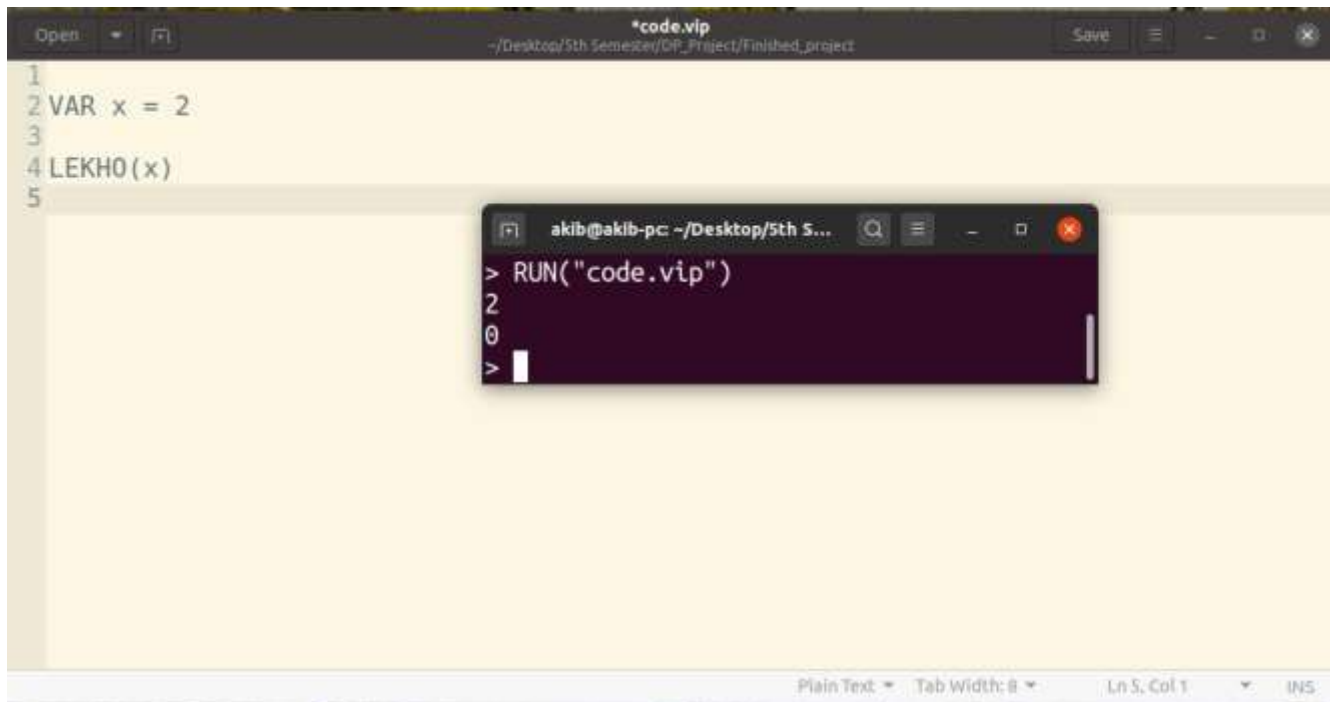
IV.   Example

1. Print "Hello World"

2. Declare a variable


```
1
2 VAR x = 2
3
4 LEKHO(x)
5
```

```
> RUN("code.vip")
2
0
>
```

3. Addition, subtraction, multiplication, division, modulus of 2 variables


```
1
2 VAR x = 15
3 VAR y = 10
4
5 LEKHO("ADDITION")
6 LEKHO(x+y)
7
8 LEKHO("SUBTRACTION")
9 LEKHO(x-y)
10
11 LEKHO("MULTIPLICATION")
12 LEKHO(x*y)
13
14 LEKHO("DIVISION")
15 LEKHO(x/y)
16
17 LEKHO("MODULUS")
18 LEKHO(x%y)
19
```

```
> RUN("code.vip")
ADDITION
25
SUBTRACTION
5
MULTIPLICATION
150
DIVISION
1.5
MODULUS
5
0
>
```

4. Division by 0 error


```
1
2 VAR x = 15
3 VAR y = 0
4
5 VAR z = x/y
6
7 LEKHO(z)
```

```
> RUN("code.vip")
Traceback (most recent call last):
  File <stdin>, line 1, in <program>
  File <stdin>, line 1, in run
Runtime Error: Failed to finish executing script "code.vip"
Traceback (most recent call last):
  File code.vip, line 5, in <program>
Runtime Error: Division by zero


VAR z = x/y
        ^

RUN("code.vip")
^^^^^^^^^^^^^^
>
```

5. String


```
1
2 VAR x = "AKIB"
3
4 LEKHO(x)
```

```
>
> RUN("code.vip")
AKIB
0
>
```

6. Taking input from user (Integer, string both)

7. Taking input from user (Integer only)
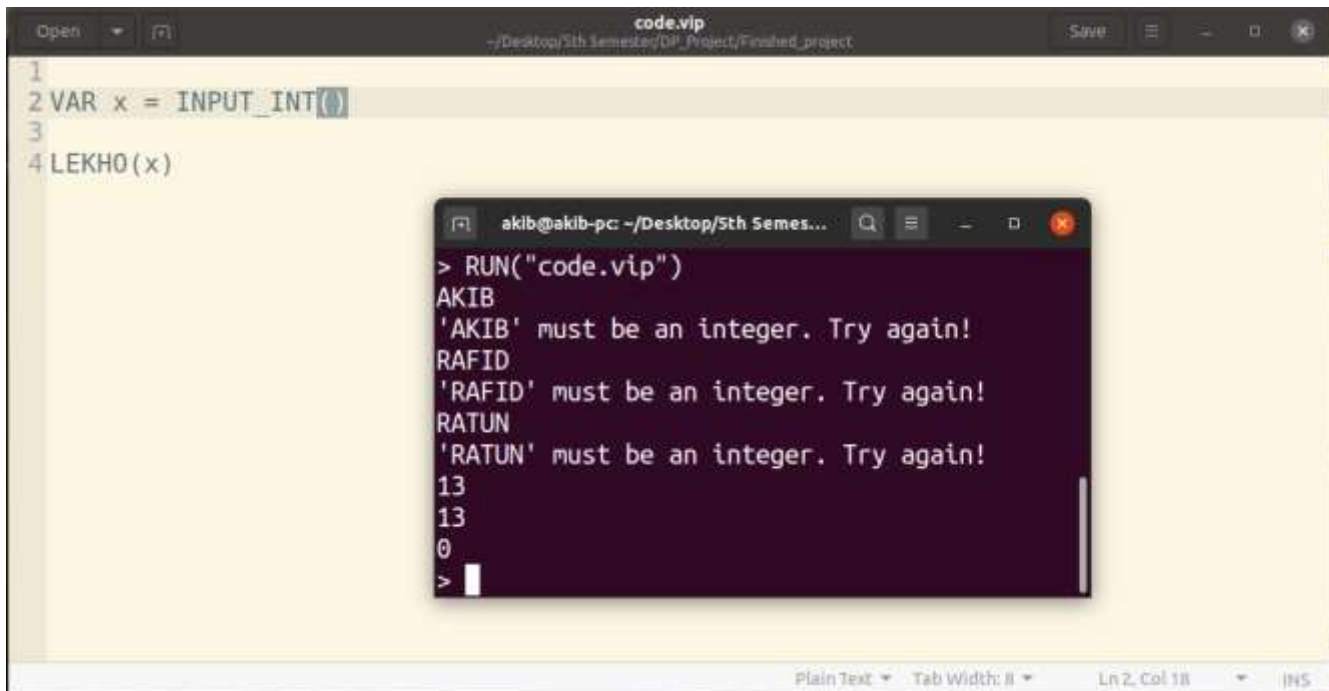
8. Try to put string value in INPUT_INT()

```
1
2 VAR x = INPUT_INT()
3
4 LEKHO(x)
```

```
> RUN("code.vip")
AKIB
'AKIB' must be an integer. Try again!
RAFID
'RAFID' must be an integer. Try again!
RATUN
'RATUN' must be an integer. Try again!
13
13
0
>
```

9. Conditions

```
1
2 VAR x = INPUT_INT()
3
4 JODI x == 10 TAHOLEY LEKHO("x==10") NAJODI x>15 TAHOLEY LEKHO("x>15") NAHOLEY
  LEKHO("TRY")
5
```

```
> RUN("code.vip")
10
x==10
0
> RUN("code.vip")
20
x>15
0
> RUN("code.vip")
3
TRY
0
>
```

10. Loop (While loop)
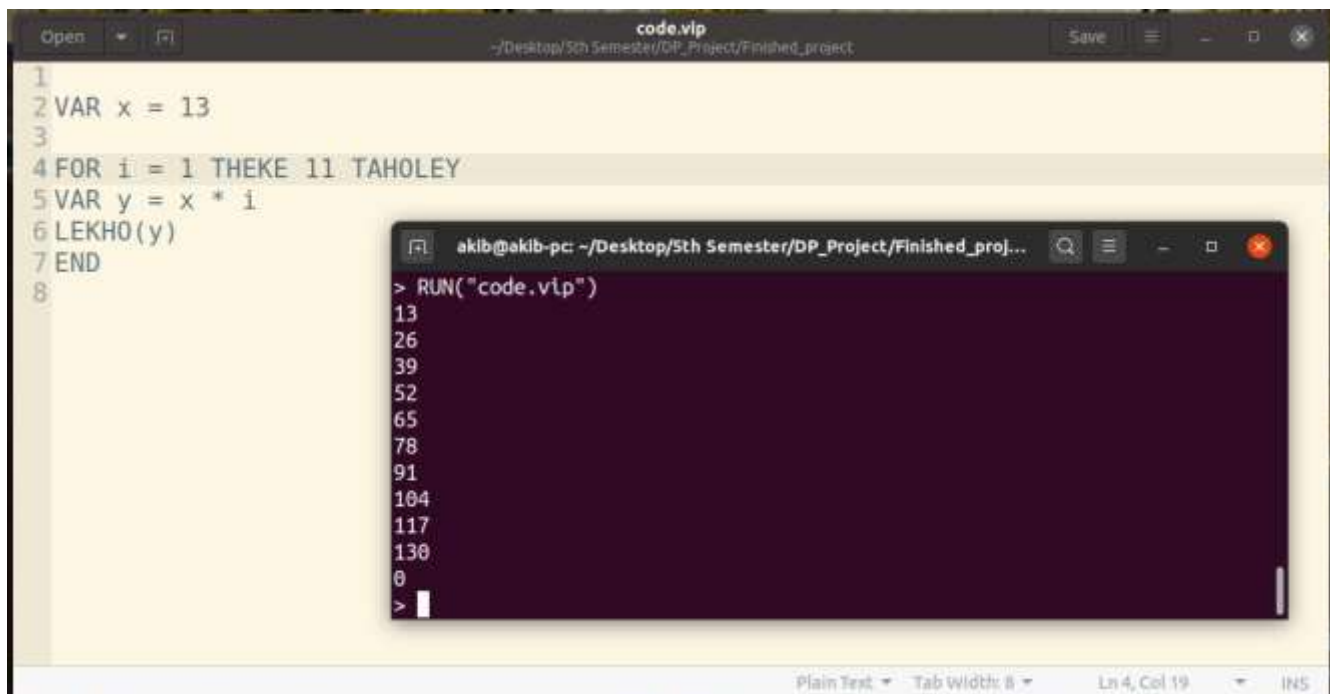
```
1
2 VAR i = 0
3
4 WHILE i<10 TAHOLEY
5 VAR i = i+1
6 LEKHO(i)
7 END
8
```

```
> RUN("code.vip")
1
2
3
4
5
6
7
8
9
10
0
>
```

11. Loop (for loop)

```
1
2 VAR x = 13
3
4 FOR i = 1 THEKE 11 TAHOLEY
5 VAR y = x * i
6 LEKHO(y)
7 END
8
```

```
> RUN("code.vip")
13
26
39
52
65
78
91
104
117
130
0
>
```

12. For loop with steps (and reverse)

```
1
2 VAR x = 13
3
4 FOR i = 5 THEKE 0 STEP -1 TAHOLEY
5 VAR y = x + i
6 LEKHO(y)
7 END
8
```

```
> RUN("code.vip")
18
17
16
15
14
0
>
```

13. Continue

```
1
2 VAR i = 0
3 VAR a = 0
4
5 WHILE i<5 TAHOLEY
6 VAR i = i+1
7 JODI i==3 TAHOLEY CONTINUE
8 VAR a = a+1
9 END
10
11 LEKHO(a)
12
```
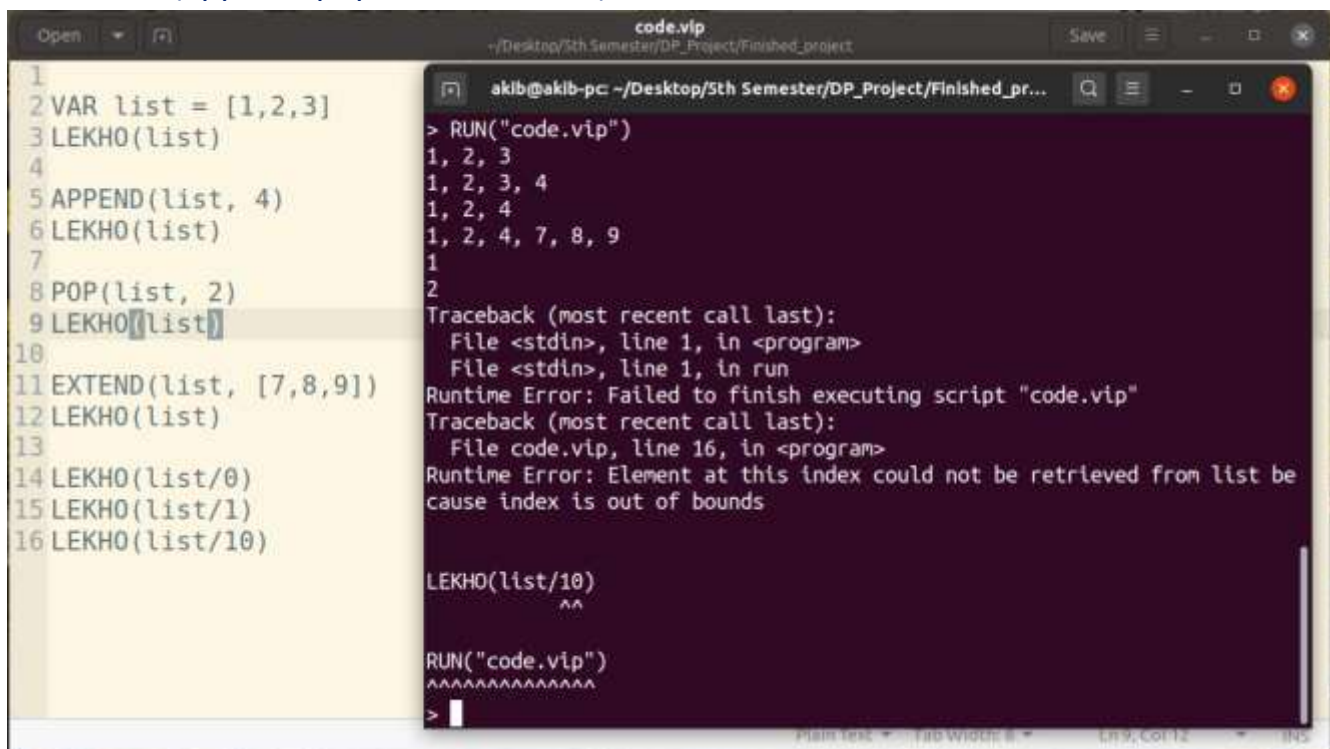
```
>
> RUN("code.vip")
4
0
>
```

14. Break


```
1
2 VAR a = 10
3
4 FOR i = 0 THEKE 15 TAHOLEY
5 VAR a = a+i
6 JODI a==13 TAHOLEY BREAK
7 END
8
9 LEKHO(a)
10
```

```
>
> RUN("code.vip")
13
0
>
```

15. List (append, pop, extend, index)


```
1
2 VAR list = [1,2,3]
3 LEKHO(list)
4
5 APPEND(list, 4)
6 LEKHO(list)
7
8 POP(list, 2)
9 LEKHO(list)
10
11 EXTEND(list, [7,8,9])
12 LEKHO(list)
13
14 LEKHO(list/0)
15 LEKHO(list/1)
16 LEKHO(list/10)
```

```
> RUN("code.vip")
1, 2, 3
1, 2, 3, 4
1, 2, 4
1, 2, 4, 7, 8, 9
1
2
Traceback (most recent call last):
  File <stdin>, line 1, in <program>
  File <stdin>, line 1, in run
Runtime Error: Failed to finish executing script "code.vip"
Traceback (most recent call last):
  File code.vip, line 16, in <program>
Runtime Error: Element at this index could not be retrieved from list be
cause index is out of bounds

LEKHO(list/10)
        ^^

RUN("code.vip")
^^^^^^^^^^^^^^
>
```

16. Functions

```
code.vip
~/Desktop/5th Semester/DP_Project/Finished_project

1 FUN add (a,b) -> a+b
2
3 VAR x = INPUT_INT()
4 VAR y = INPUT_INT()
5 LEKHO(add(x,y))
```

```
akib@akib-pc: ~/Deskto...

>
> RUN("code.vip")
5
6
11
0
>
```

Plain Text ▾   Tab Width: 8 ▾   Ln 4, Col 20   ▾   INS

17.Bitwise Operator

&=and, |=or, ~=XOR

```
code.vip
~/Desktop/5th Semester/DP_Project/Finished_project

1 VAR a = 10
2 VAR b = 4
3
4 LEKHO(a & b)
5 LEKHO(a | b)
6 LEKHO(a ~ b)
```

```
akib@akib-pc: ~/Deskto...

> RUN("code.vip")
0
14
14
0
>
```

Plain Text ▾   Tab Width: 8 ▾   Ln 6, Col 10   ▾   INS

18. Shift Operator

{= left shift, }=right shift
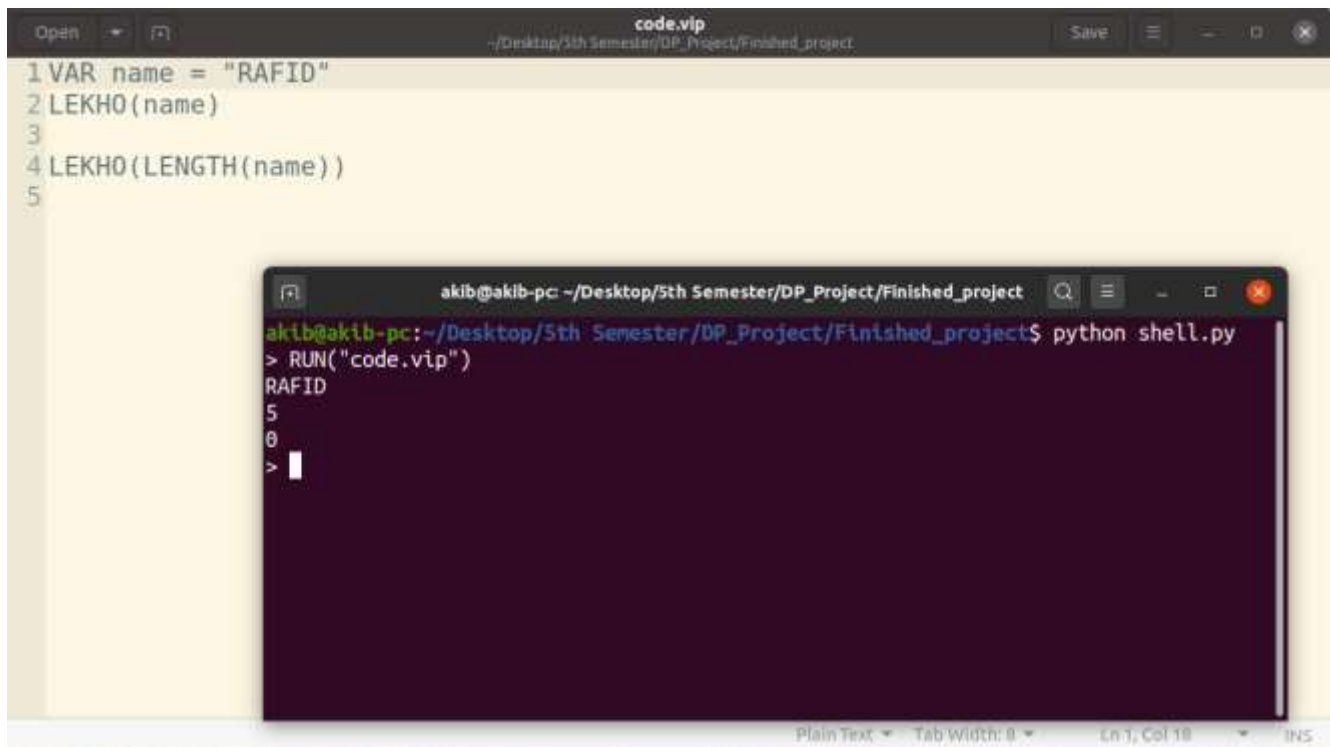

```
1 VAR a = 10
2 VAR x = 1
3
4 LEKHO(a { x)
5 LEKHO(a } x)
6
```

```
> RUN("code.vip")
20
5
0
>
```

19. String Length


```
1 VAR name = "RAFID"
2 LEKHO(name)
3
4 LEKHO(LENGTH(name))
5
```

```
akib@akib-pc:~/Desktop/5th Semester/DP_Project/Finished_project$ python shell.py
> RUN("code.vip")
RAFID
5
0
>
```

20. String length without string


```
1 VAR name = 11
2 LEKHO(name)
3
4 LEKHO(LENGTH(name))
5
```
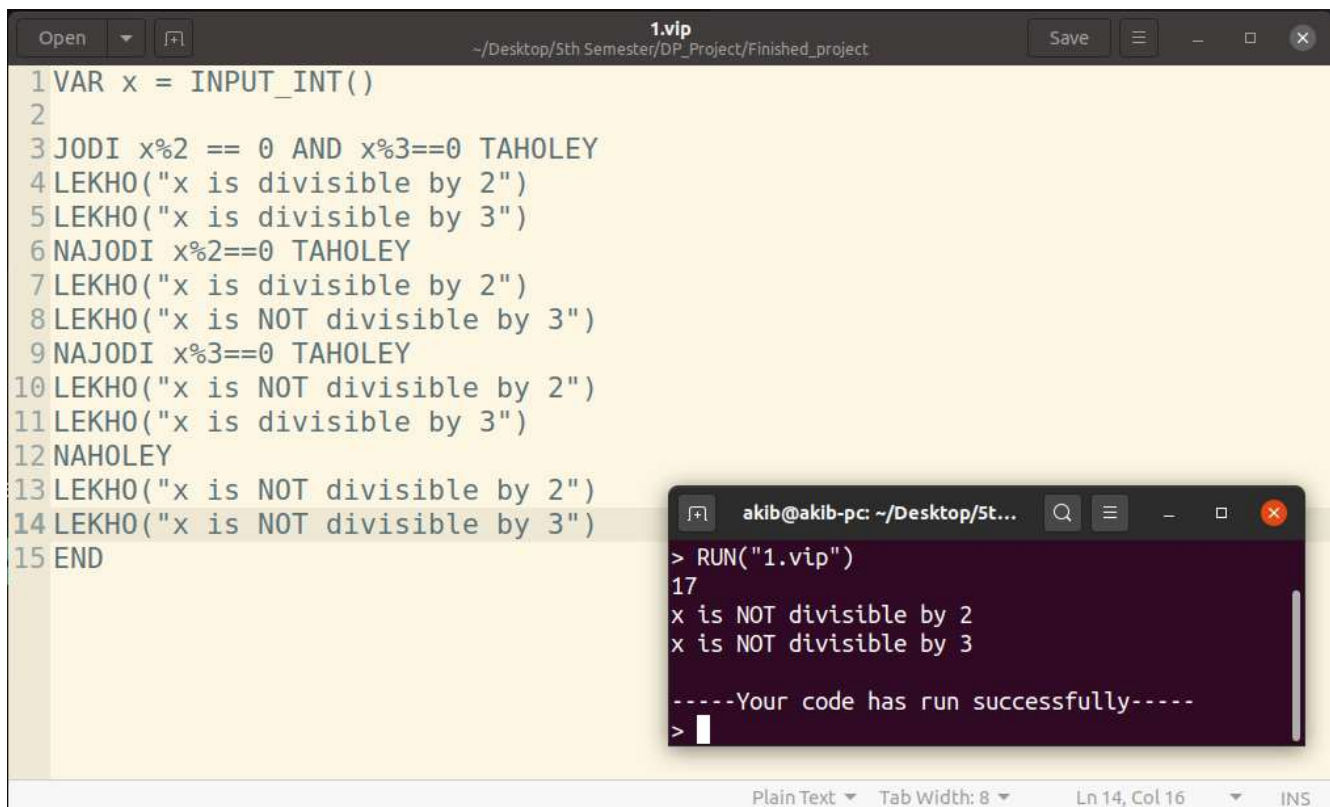
```
akib@akib-pc: ~/Desktop/5th Semester/DP_Project/Finished_project
> RUN("code.vip")
11
Traceback (most recent call last):
  File <stdin>, line 1, in <program>
  File <stdin>, line 1, in run
Runtime Error: Failed to finish executing script "code.vip"
Traceback (most recent call last):
  File code.vip, line 4, in <program>
  File code.vip, line 4, in length
Runtime Error: Argument must be string


LEKHO(LENGTH(name))
        ^^^^^^^^^^^

RUN("code.vip")
^^^^^^^^^^^^^^^

>
```

21.Multiple statement line of if statement (using END)


```
1 VAR x = INPUT_INT()
2
3 JODI x%2 == 0 AND x%3==0 TAHOLEY
4 LEKHO("x is divisible by 2")
5 LEKHO("x is divisible by 3")
6 NAJODI x%2==0 TAHOLEY
7 LEKHO("x is divisible by 2")
8 LEKHO("x is NOT divisible by 3")
9 NAJODI x%3==0 TAHOLEY
10 LEKHO("x is NOT divisible by 2")
11 LEKHO("x is divisible by 3")
12 NAHOLEY
13 LEKHO("x is NOT divisible by 2")
14 LEKHO("x is NOT divisible by 3")
15 END
```

```
akib@akib-pc: ~/Desktop/5t...
> RUN("1.vip")
17
x is NOT divisible by 2
x is NOT divisible by 3

-----Your code has run successfully-----
>
```

19

22. Multiple statement line of function (using END)

```
1 VAR x = INPUT_INT()
2 VAR y = INPUT_INT()
3
4 FUN add(x,y)
5 VAR a = x
6 VAR b = y
7 VAR c = a+b
8 RETURN c
9 END
10
11 LEKHO(add(x,y))
12
```
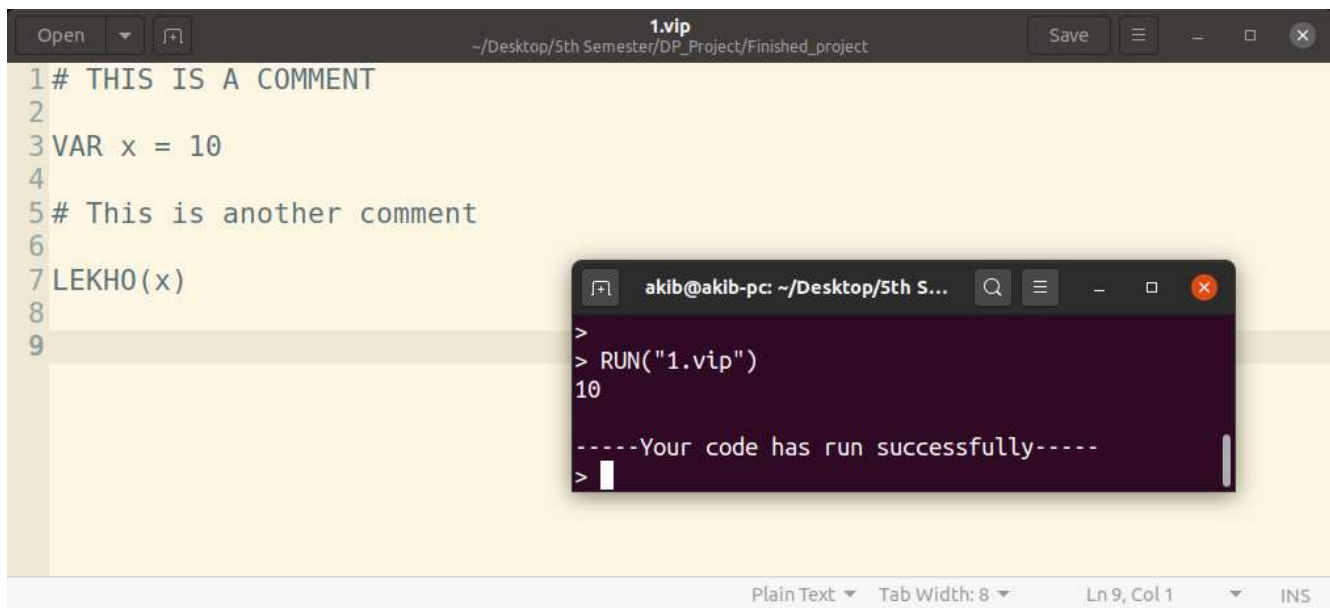
```
> RUN("1.vip")
2
3
5

-----Your code has run successfully-----
>
```
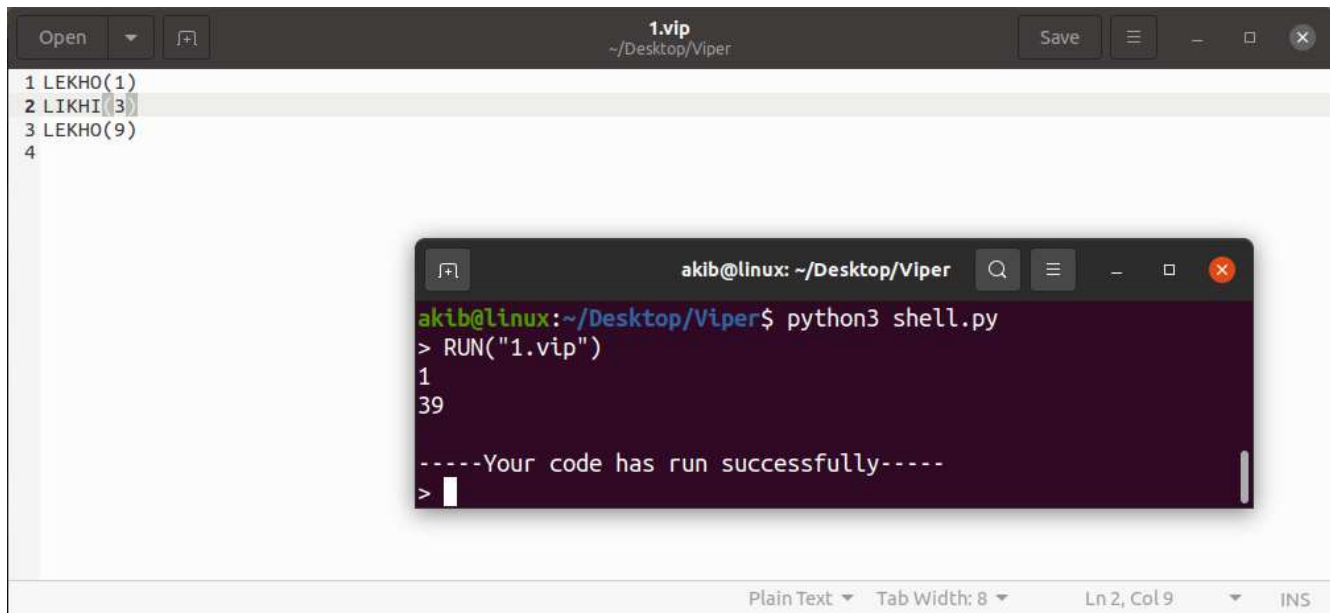
23. Comment

```
1 # THIS IS A COMMENT
2
3 VAR x = 10
4
5 # This is another comment
6
7 LEKHO(x)
8
9
```

```
>
> RUN("1.vip")
10

-----Your code has run successfully-----
>
```

24. Print with and without newline


```
1 LEKHO(1)
2 LIKHI(3)
3 LEKHO(9)
4
```

```
akib@linux:~/Desktop/Viper$ python3 shell.py
> RUN("1.vip")
1
39

-----Your code has run successfully-----
>
```

Competitive Programming Code:

Codeforces: 4A Watermelon

https://codeforces.com/problemset/problem/4/A

```
1
2 VAR x = INPUT_INT()
3
4 JODI x%2==0 AND x>2 TAHOLEY LEKHO("YES") NAHOLEY LEKHO("NO")
```

```
> RUN("code.vip")
8
YES
0
> RUN("code.vip")
5
NO
0
>
```

Codeforces: 231A Team

https://codeforces.com/problemset/problem/231/A

```
1 VAR sum = 0
2 VAR ans = 0
3 VAR x = INPUT_INT()
4
5
6 FOR i=0 THEKE x TAHOLEY
7 VAR sum = 0
8 FOR j=0 THEKE 3 TAHOLEY
9 VAR z = INPUT_INT()
10 JODI z==1 TAHOLEY VAR sum=sum+1
11 END
12 JODI sum>=2 TAHOLEY VAR ans = ans+1
13 END
14 LEKHO(ans)
```

```
> RUN("code.vip")
3
1
1
0
1
1
1
1
0
0
2
0
>
```

22

Codeforces: 71A Way Too Long Word

https://codeforces.com/problemset/problem/71/A
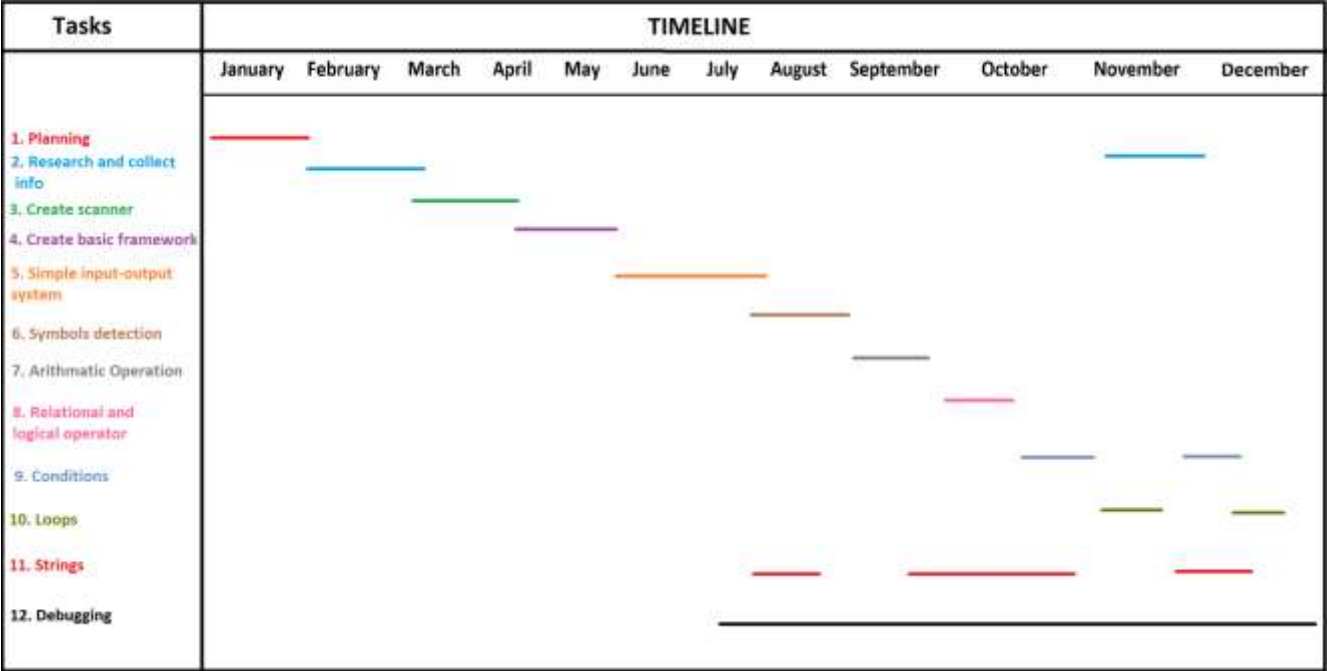


```
 1 VAR n = INPUT_INT()
 2
 3 FOR i = 0 THEKE n TAHOLEY
 4 VAR s = INPUT()
 5 JODI LENGTH(s) < 10 TAHOLEY
 6 LIKHI(s)
 7 NAHOLEY
 8 LEKHO(s/0)
 9 LEKHO(LENGTH(s)-2)
10 LIKHI(s/(LENGTH(s)-1))
11 END
12 END
13
```

```
> RUN("code.vip")
2
RAFID
RAFID
ratunrahman
r9n

-----Your code has run successfully-----
>
```

TIMELINE

V. CONCLUSIONS AND FUTURE WORK

A . CONCLUSIONS

In this paper, the design of an interpreter for the programming language in the context of a software engineering project has been presented. Some requirements include, but are not limited to, writing a complete project using challenging algorithms and data structures, use of different development tools, object oriented design, and team management which is an important issue to consider given that only team work in software engineering and database courses.

B . FUTURE WORK

1. Future work will focus on creating an interactive source level debugger for the language that enables the use of command lines to interact with the interpreter as well as an Integrated Development Environment (IDE) with a graphical user interface (GUI).
2. Add some more software terms such as object oriented, design pattern, UML diagram etc.
3. More specific error handling system: Better anticipation, detection and resolution of programming, application and communication error. Establish safety critical systems and exception handling.

Reference:

[1] Sommerville, I. Software Engineering. Addisson Wesley, 9th edition, 2010

[2] C. Xing. "How Interpreters Work: An Overlooked Topic in Undergraduate Computer Science Education," Proc. In CCSC Southern Eastern Conference, JCSC Vol. 25,Iissue 2. December 2009

[3] R. Mak. Writing Compilers and Interpreters: A Modern Software Engineering Approach Using Java. Wiley, 3rd edition, 2009

[4] H. Deitel, Java: How to Program (early projects), 10th edition, Prentice Hall Inc. , 2014.

[5] R. Sebesta, Concepts of Programming Languages, 10th Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2012.

[6] S. Reiss, and T. Davis. "Experiences Writing Object-Oriented Compiler Front Ends". Tech. Rep., Brown University, January 1995.

[7] B. Malloy., J. Power, and J. Waldron. "Applying Software Engineering Techniques to Parser Design: The Development of a C# Parser," in Proc. of SAICSIT 2002, pp. 74–81

[8] A. Ghuloum, "An Incremental Approach to Compiler Construction," in Proc. of the 2006 Scheme and Functional Programming Workshop, 2006, pp. 28.

[9] A. Demaille. "Making Compiler Construction Projects Relevant to Core Curriculums," In proceeding of: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2005, Caparica, Portugal, June 27-29, 2005.

Github Link:

https://github.com/Ratun011/VIPER-new-programming-language