Department of Computer Science & Technology

MINI PROJECT REPORT

ON

" Interpreter for the BASIC language written in Python 3"

SUBMITTED TO THE 6th SEMESTER COMPILER DESIGN AND SYSTEM SOFTWARE LABORATORY (21CT3606)

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE & TECHNOLOGY

Submitted by

Allan Dsouza - (ENG21CT0002)
Hemal S - (ENG21CT0009)
Jaice Joseph - (ENG21CT0011)
Swathi S - (ENG21CT0043)

Under the supervision of
Dr. D. Sudha,
Associate Professor, Dept. Of. CST.

DAYANANDA SAGAR UNIVERSITY

SCHOOL OF ENGINEERING

Department of Computer Science & Technology

CERTIFICATE

This is to certify that ***Mr./Ms. Allan Dsouza, Hemal S, Jaice Joseph, Swathi S*** bearing USN ***ENG21CT0002, ENG21CT0009 , ENG21CT0011 , ENG21CT0043*** has satisfactorily completed his/her Mini Project as prescribed by the University for the *6th semester* B.Tech. programme in *Computer Science & Technology* during the year *2024* at the School of Engineering, Dayananda Sagar University., Bangalore.

Date: .

Signature of the faculty in-charge

Signature of Chairperson

Department of Computer Science & Technology

DECLARATION

We hereby declare that the work presented in this mini project entitled "***Interpreter for the BASIC language written in Python 3***" has been carried out by us and it has not been submitted for the award of any degree, diploma or the mini project of any other college or university.

ALLAN DSOUZA- (ENG21CT0002)
HEMAL S - (ENG21CT0009)
JAICE JOSEPH - (ENG21CT0011)
SWATHI S - (ENG21CT0043)

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of task would be incomplete without the mention of the people who made it possible and whose constant guidance and encouragement crown all the efforts with success.

We are especially thankful to our **Chairperson, Dr. M. Shahina Parveen**, for providing necessary departmental facilities, moral support and encouragement.

We are very much thankful to our guide **Dr.D.Sudha** , for providing help and suggestions in completion of this project successfully.

We wish to thank all that have directly or indirectly helped us in the successful completion of this project work.

ALLAN DSOUZA- (ENG21CT0002)
HEMAL S - (ENG21CT0009)
JAICE JOSEPH - (ENG21CT0011)
SWATHI S - (ENG21CT0043)

TABLE OF CONTENTS

ABSTRACT

This project presents a fully functional interpreter for the BASIC programming language, implemented in Python 3. The interpreter is designed to parse and execute code written in BASIC, a language known for its simplicity and historical significance in the early days of personal computing. The implementation focuses on supporting the core features of BASIC, including line-numbered statements, control flow constructs (such as GOTO, IF-THEN, and FOR-NEXT loops), and basic input/output operations. By leveraging Python's robust parsing libraries and its dynamic typing system, the interpreter achieves a balance between ease of implementation and performance.

Key components of the interpreter include a tokenizer to break down the BASIC source code into lexical tokens, a parser to construct a syntax tree from these tokens, and an evaluator to execute the parsed instructions. Additionally, the project incorporates error handling to provide informative feedback for common programming mistakes, enhancing the learning experience for users.

This project not only serves as a tool for executing BASIC programs but also as an educational resource, illustrating fundamental concepts in interpreter design, such as lexical analysis, parsing, and runtime evaluation. By implementing the interpreter in Python, the project highlights the versatility and readability of Python as a language for developing complex software systems

INTRODUCTION

The BASIC (Beginner's All-purpose Symbolic Instruction Code) programming language has played a pivotal role in the history of computing, particularly in making programming accessible to a broad audience during the early days of personal computers. Designed in the mid-1960s, BASIC's simplicity and ease of use made it a popular choice for beginners and hobbyists, leading to widespread adoption on early microcomputers.

Despite the evolution of programming languages and the advent of more modern development environments, BASIC remains a valuable educational tool. It provides an excellent platform for understanding fundamental programming concepts without the complexity of contemporary languages. This project aims to revive the essence of BASIC by creating an interpreter implemented in Python 3, offering both historical insight and practical learning opportunities.

The interpreter is crafted to process and execute BASIC code, supporting its distinctive features such as line-numbered instructions, control flow constructs (GOTO, IF-THEN, FOR-NEXT loops), and basic input/output operations. Python 3, known for its readability and extensive standard library, serves as the ideal language for this implementation, facilitating the development of a clear and maintainable codebase.

In developing this interpreter, several key components are addressed:

Tokenizer: Converts the raw BASIC source code into a series of lexical tokens, representing the smallest units of meaning (e.g., keywords, operators, identifiers).

Parser: Analyzes the sequence of tokens to construct a syntax tree, representing the hierarchical structure of the program.

Evaluator: Traverses the syntax tree and executes the corresponding instructions, managing program state and control flow.

Additionally, the interpreter includes comprehensive error handling to assist users in identifying and correcting common mistakes, thereby enhancing the learning experience.

Software and Hardware Requirements

Software Requirements:

1. **Operating System:** Windows 10 or later
2. **Python Interpreter:** Python 3.7 or later is required to run the interpreter. It can be downloaded from the official Python website (https://www.python.org/).
3. **Libraries and Dependencies:**
    - **Standard Libraries:** like string , os , math
    - **Graphical User Interface Libraries:** tkinter.
4. **External Libraries:**

   basic: A custom module (assumed to be created as part of this project) that contains the core logic for the BASIC interpreter, including tokenizing, parsing, and evaluating BASIC code.
5. **Development Environment:** Visual Studio Code

Hardware Requirements:

Processor: Intel Core i3 or later

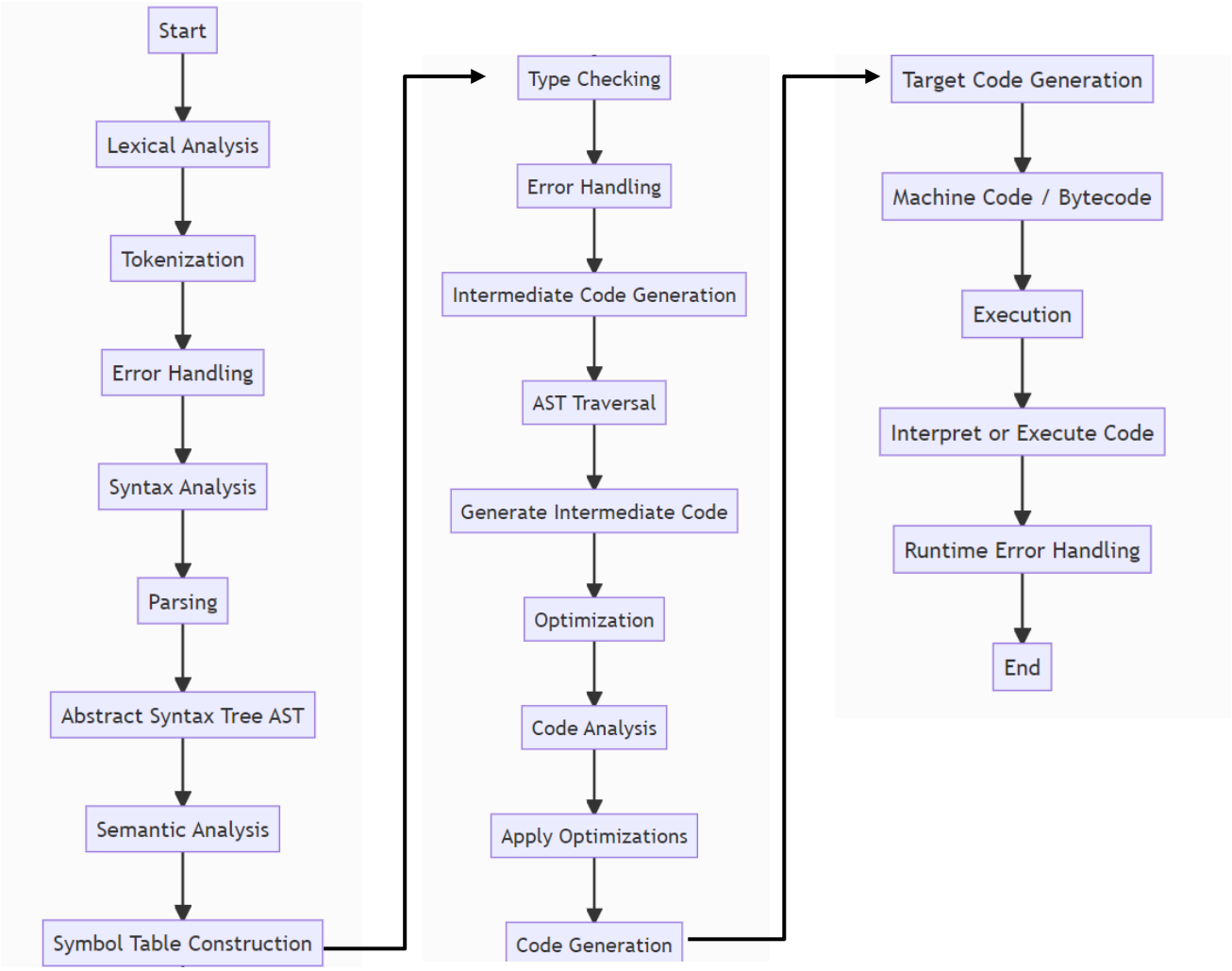
PROBLEM STATEMENT

The BASIC programming language, though historically significant and educationally valuable, lacks modern interpreters that can run on contemporary systems. As a result, there is a gap in accessible tools for learning and experimenting with BASIC, which limits opportunities for both historical exploration and educational purposes. Additionally, many existing BASIC interpreters are either outdated, lack comprehensive documentation, or are not designed with modern programming practices in mind.

This project aims to address this gap by developing a new interpreter for the BASIC language using Python 3. The interpreter will enable users to write, execute, and debug BASIC programs on current computing platforms. The key challenges to be addressed in this project include:

- **Lexical Analysis:** Implementing a tokenizer that accurately converts BASIC source code into tokens, recognizing keywords, operators, and various syntax elements.
- **Syntax Parsing:** Developing a parser that can construct a valid syntax tree from the tokens, handling the unique structure and syntax rules of BASIC.
- **Runtime Evaluation:** Creating an evaluator that can correctly execute the parsed syntax tree, managing program state, control flow, and input/output operations.

DESIGN


Start
Lexical Analysis
Tokenization
Error Handling
Syntax Analysis
Parsing
Abstract Syntax Tree AST
Semantic Analysis
Symbol Table Construction
Type Checking
Error Handling
Intermediate Code Generation
AST Traversal
Generate Intermediate Code
Optimization
Code Analysis
Apply Optimizations
Code Generation
Target Code Generation
Machine Code / Bytecode
Execution
Interpret or Execute Code
Runtime Error Handling
End

DESCRIPTION OF MODULES

Lexer Module:

The Lexer module, also known as the tokenizer, is responsible for breaking down the raw BASIC source code into a series of lexical tokens. These tokens represent the smallest units of meaning within the code, such as keywords, identifiers, operators, numbers, and punctuation marks. The Lexer scans the input line by line, recognizing patterns based on predefined rules and generating tokens accordingly. This process simplifies the subsequent parsing stage by converting the complex source code into a manageable and structured sequence of tokens. Effective error handling within the Lexer ensures that syntax errors are detected early, providing informative feedback to the user.

```
#######################################
# LEXER
#######################################

class Lexer:
  def __init__(self, fn, text):
    self.fn = fn
    self.text = text
    self.pos = Position(-1, 0, -1, fn, text)
    self.current_char = None
    self.advance()

  def advance(self):
    self.pos.advance(self.current_char)
    self.current_char = self.text[self.pos.idx] if self.pos.idx < len(self.text) el

  def make_tokens(self):
    tokens = []

    while self.current_char != None:
      if self.current_char in ' \t':
        self.advance()
      elif self.current_char == '#':
        self.skip_comment()
      elif self.current_char in ';\n':
        tokens.append(Token(TT_NEWLINE, pos_start=self.pos))
        self.advance()
      elif self.current_char in DIGITS:
        tokens.append(self.make_number())
      elif self.current_char in LETTERS:
        tokens.append(self.make_identifier())
      elif self.current_char == '"':
        tokens.append(self.make_string())
      elif self.current_char == '+':
        tokens.append(Token(TT_PLUS, pos_start=self.pos))
        self.advance()
      elif self.current_char == '-':
        tokens.append(self.make_minus_or_arrow())
      elif self.current_char == '*':
        tokens.append(Token(TT_MUL, pos_start=self.pos))
        self.advance()
      elif self.current_char == '/':
        tokens.append(Token(TT_DIV, pos_start=self.pos))
        self.advance()
      elif self.current_char == '^':
        tokens.append(Token(TT_POW, pos_start=self.pos))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

py-basicinterp/ep1> python3 shell.py
basic > 1 + 2
[INT:1, PLUS, INT:2]
basic > 2.5 * 2.5
[FLOAT:2.5, MUL, FLOAT:2.5]
basic > 2 * d
Illegal Character: 'd'
basic >
```

Parser Module:

The Parser module takes the list of tokens produced by the Lexer and organizes them into a syntax tree, also known as an abstract syntax tree (AST). This tree represents the hierarchical structure of the program, reflecting the grammatical

rules of the BASIC language. The parser ensures that the token sequence adheres to the correct syntax, identifying constructs such as variable declarations, control flow statements, and expressions. By constructing the AST, the Parser enables the interpreter to understand the logical flow and structure of the BASIC program. Any syntax errors detected during this stage are reported, allowing users to correct them before execution.

```
#######################################
# PARSER
#######################################

class Parser:
  def __init__(self, tokens):
    self.tokens = tokens
    self.tok_idx = -1
    self.advance()

  def advance(self):
    self.tok_idx += 1
    self.update_current_tok()
    return self.current_tok

  def reverse(self, amount=1):
    self.tok_idx -= amount
    self.update_current_tok()
    return self.current_tok

  def update_current_tok(self):
    if self.tok_idx >= 0 and self.tok_idx < len(self.tokens):
      self.current_tok = self.tokens[self.tok_idx]

  def parse(self):
    res = self.statements()
    if not res.error and self.current_tok.type != TT_EOF:
      return res.failure(InvalidSyntaxError(
        self.current_tok.pos_start, self.current_tok.pos_end,
        "Token cannot appear after previous tokens"
      ))
    return res

  ###################################

  def statements(self):
    res = ParseResult()
    statements = []
    pos_start = self.current_tok.pos_start.copy()

    while self.current_tok.type == TT_NEWLINE:
      res.register_advancement()
      self.advance()

    statement = res.register(self.statement())
    if res.error: return res
    statements.append(statement)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

myopl-ep2>python3 shell.py
basic > -5
(MINUS, INT:5)
basic > -5 + 6 * 7
((MINUS, INT:5), PLUS, (INT:6, MUL, INT:7))
```

Symbol Table Module:

The Symbol Table module is a crucial component for managing the scope and binding of variables within the BASIC program. It maintains a mapping between variable names and their corresponding values, along with other relevant information such as data types and memory locations. During both parsing and evaluation, the symbol table is consulted to resolve variable references and ensure that variables are used consistently and correctly throughout the program. The module handles variable declarations, updates, and lookups efficiently,

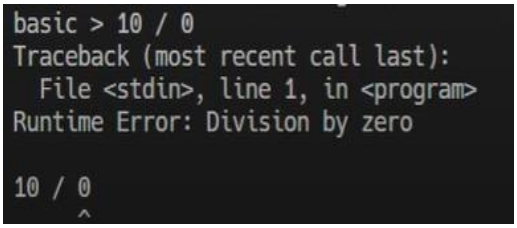
supporting the interpreter's execution by providing a reliable mechanism for variable management.

```
#####################################
# SYMBOL TABLE
#####################################

class SymbolTable:
  def __init__(self, parent=None):
    self.symbols = {}
    self.parent = parent

  def get(self, name):
    value = self.symbols.get(name, None)
    if value == None and self.parent:
      return self.parent.get(name)
    return value

  def set(self, name, value):
    self.symbols[name] = value

  def remove(self, name):
    del self.symbols[name]
```

Interpreter Module:

The Interpreter module is the core of the BASIC interpreter, responsible for executing the instructions defined by the syntax tree. It traverses the AST, evaluating expressions, executing statements, and managing the overall program state. The interpreter handles various control flow constructs such as loops, conditionals, and procedure calls, ensuring that the program logic is faithfully executed as intended. This module integrates closely with the Symbol Table to retrieve and update variable values during execution. The Interpreter also includes mechanisms for input/output operations, allowing BASIC programs to interact with the user and the system environment.

```
###################################
# INTERPRETER
###################################

class Interpreter:
  def visit(self, node, context):
    method_name = f'visit_{type(node).__name__}'
    method = getattr(self, method_name, self.no_visit_method)
    return method(node, context)

  def no_visit_method(self, node, context):
    raise Exception(f'No visit_{type(node).__name__} method defined')

  ###################################

  def visit_NumberNode(self, node, context):
    return RTResult().success(
      Number(node.tok.value).set_context(context).set_pos(node.pos_start, node.pos_end)
    )

  def visit_StringNode(self, node, context):
    return RTResult().success(
      String(node.tok.value).set_context(context).set_pos(node.pos_start, node.pos_end)
    )

  def visit_ListNode(self, node, context):
    res = RTResult()
    elements = []

    for element_node in node.element_nodes:
      elements.append(res.register(self.visit(element_node, context)))
      if res.should_return(): return res

    return res.success(
      List(elements).set_context(context).set_pos(node.pos_start, node.pos_end)
    )

  def visit_VarAccessNode(self, node, context):
    res = RTResult()
```



```
basic > 10 / 0
Traceback (most recent call last):
  File <stdin>, line 1, in <program>
Runtime Error: Division by zero

10 / 0
     ^
```

Run Module:

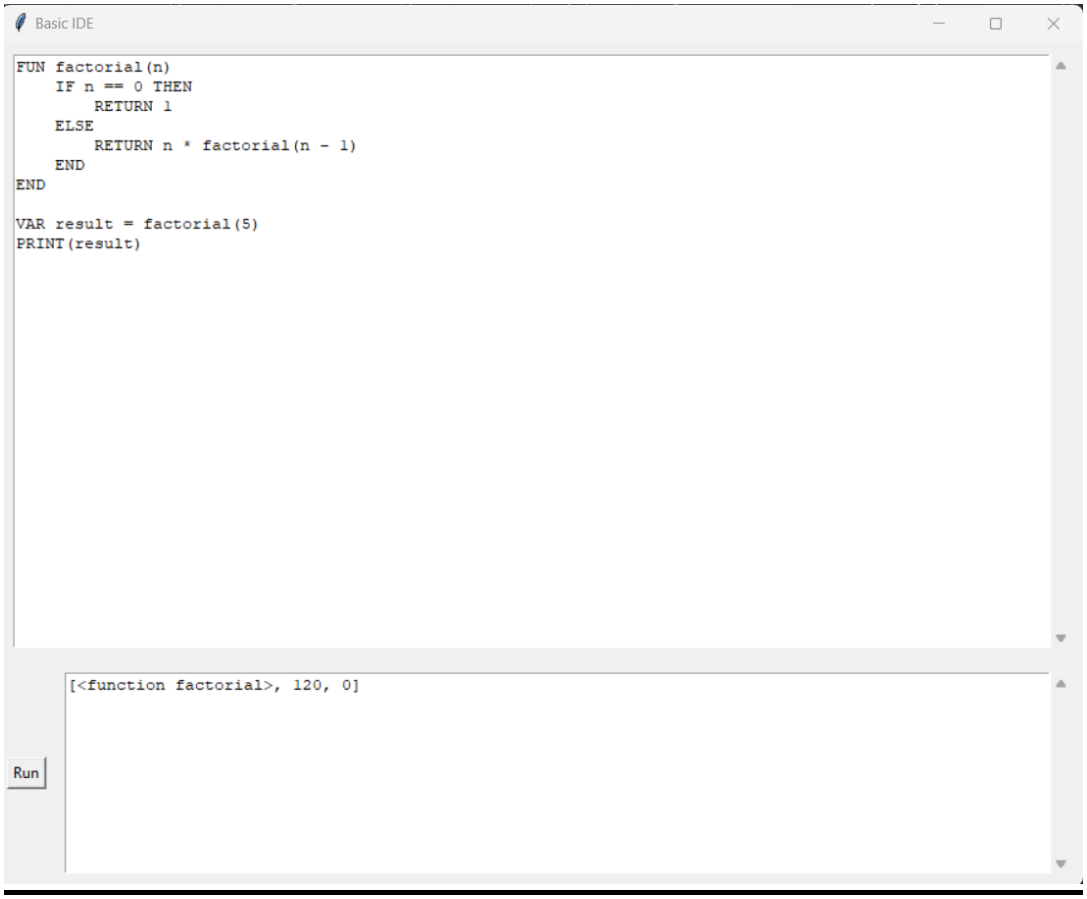
The Run module serves as the entry point for the interpreter, coordinating the overall process of reading, parsing, and executing BASIC programs. It handles the user interface, either command-line or graphical, and manages the workflow from loading source code to displaying results. The Run module invokes the Lexer to tokenize the input, the Parser to generate the syntax tree, and the Interpreter to execute the program. It also handles error reporting and user interactions, providing a seamless experience for running BASIC programs. This module ensures that the various components of the interpreter work together harmoniously, enabling users to write, debug, and execute their BASIC code efficiently.

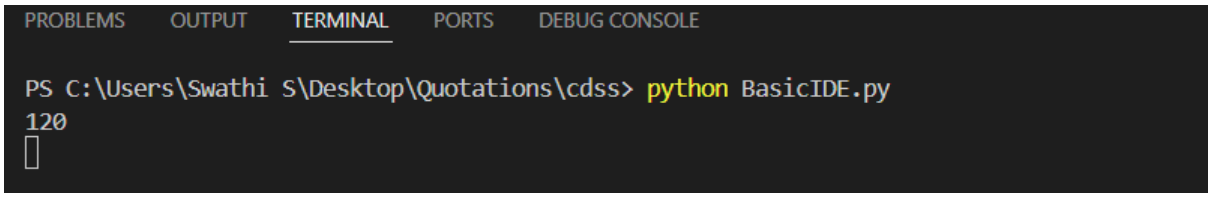
```
#######################################
# RUN
#######################################

global_symbol_table = SymbolTable()
global_symbol_table.set("NULL", Number.null)
global_symbol_table.set("FALSE", Number.false)
global_symbol_table.set("TRUE", Number.true)
global_symbol_table.set("MATH_PI", Number.math_PI)
global_symbol_table.set("PRINT", BuiltInFunction.print)
global_symbol_table.set("PRINT_RET", BuiltInFunction.print_ret)
global_symbol_table.set("INPUT", BuiltInFunction.input)
global_symbol_table.set("INPUT_INT", BuiltInFunction.input_int)
global_symbol_table.set("CLEAR", BuiltInFunction.clear)
global_symbol_table.set("CLS", BuiltInFunction.clear)
global_symbol_table.set("IS_NUM", BuiltInFunction.is_number)
global_symbol_table.set("IS_STR", BuiltInFunction.is_string)
global_symbol_table.set("IS_LIST", BuiltInFunction.is_list)
global_symbol_table.set("IS_FUN", BuiltInFunction.is_function)
global_symbol_table.set("APPEND", BuiltInFunction.append)
global_symbol_table.set("POP", BuiltInFunction.pop)
global_symbol_table.set("EXTEND", BuiltInFunction.extend)
global_symbol_table.set("LEN", BuiltInFunction.len)
global_symbol_table.set("RUN", BuiltInFunction.run)

def run(fn, text):
  # Generate tokens
  lexer = Lexer(fn, text)
  tokens, error = lexer.make_tokens()
  if error: return None, error

  # Generate AST
  parser = Parser(tokens)
  ast = parser.parse()
  if ast.error: return None, ast.error

  # Run program
  interpreter = Interpreter()
  context = Context('<program>')
  context.symbol_table = global_symbol_table
  result = interpreter.visit(ast.node, context)

  return result.value, result.error
```

Grammar.txt

```
statements  : NEWLINE* statement (NEWLINE+ statement)* NEWLINE*

statement   : KEYWORD:RETURN expr?
            : KEYWORD:CONTINUE
            : KEYWORD:BREAK
            : expr

expr        : KEYWORD:VAR IDENTIFIER EQ expr
            : comp-expr ((KEYWORD:AND|KEYWORD:OR) comp-expr)*

comp-expr   : NOT comp-expr
            : arith-expr ((EE|LT|GT|LTE|GTE) arith-expr)*

arith-expr  : term ((PLUS|MINUS) term)*

term        : factor ((MUL|DIV) factor)*

factor      : (PLUS|MINUS) factor
            : power

power       : call (POW factor)*

call        : atom (LPAREN (expr (COMMA expr)*)? RPAREN)?

atom        : INT|FLOAT|STRING|IDENTIFIER
            : LPAREN expr RPAREN
            : list-expr
            : if-expr
            : for-expr
            : while-expr
            : func-def

list-expr   : LSQUARE (expr (COMMA expr)*)? RSQUARE

if-expr     : KEYWORD:IF expr KEYWORD:THEN
              (statement if-expr-b|if-expr-c?)
            | (NEWLINE statements KEYWORD:END|if-expr-b|if-expr-c)

if-expr-b   : KEYWORD:ELIF expr KEYWORD:THEN
              (statement if-expr-b|if-expr-c?)
            | (NEWLINE statements KEYWORD:END|if-expr-b|if-expr-c)

if-expr-c   : KEYWORD:ELSE
              statement
            | (NEWLINE statements KEYWORD:END)

for-expr    : KEYWORD:FOR IDENTIFIER EQ expr KEYWORD:TO expr
              (KEYWORD:STEP expr)? KEYWORD:THEN
              statement
            | (NEWLINE statements KEYWORD:END)

while-expr  : KEYWORD:WHILE expr KEYWORD:THEN
              statement
            | (NEWLINE statements KEYWORD:END)

func-def    : KEYWORD:FUN IDENTIFIER?
              LPAREN (IDENTIFIER (COMMA IDENTIFIER)*)? RPAREN
              (ARROW expr)
            | (NEWLINE statements KEYWORD:END)
```

OUTPUT SCREENSHOTS





CONCLUSION

The development of a BASIC interpreter in Python 3 encapsulates both a tribute to the historical significance of the BASIC language and an educational endeavor to illustrate fundamental concepts in programming language design. By implementing the interpreter through distinct modules for lexical analysis, parsing, symbol management, and execution, this project offers a comprehensive look at the inner workings of language interpreters.

The Lexer module breaks down complex source code into manageable tokens, while the Parser constructs a meaningful syntax tree that represents the program's structure. The Symbol Table ensures efficient and accurate management of variables, and the Interpreter executes the program by traversing the syntax tree and evaluating each statement. The Run module ties all these components together, providing a user-friendly interface for loading, running, and debugging BASIC programs.

Throughout this project, the use of Python 3 underscores the language's versatility and readability, making the interpreter accessible to both beginners and experienced programmers. The project not only revives the simplicity and accessibility of BASIC but also serves as a practical guide for understanding the design and implementation of programming languages.

REFERENCES

- https://en.wikipedia.org/wiki/BASIC_interpreter