General Information:

- Python 3.9 was used in this project
- Necessary Python libraries were used (String, Os, Math, etc.)
- The project was coded on Visual Studio Code
- The grammar of the language is in BNF format.
- The project has three layers
- Lexer and Parser classes was coded in this project
- Constant, Errors, Position, Token, Nodes, Values, List, Context, Symbol Table, Interpreter structures was design in this programming language
- Assignment operations, basic string operations, basic math operations, basic list operations can be done

Contents

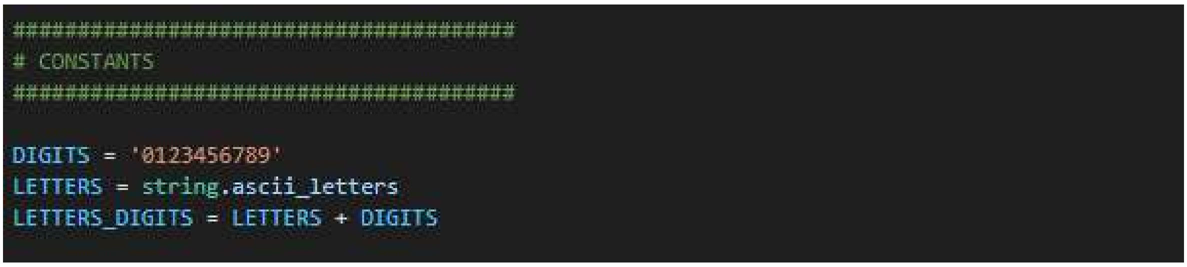
Visual Studio Code

1- System Architecture

1.1 - Fundamental

Here are the codes where almost all of our structures are created. I added screenshots and description of most classes.

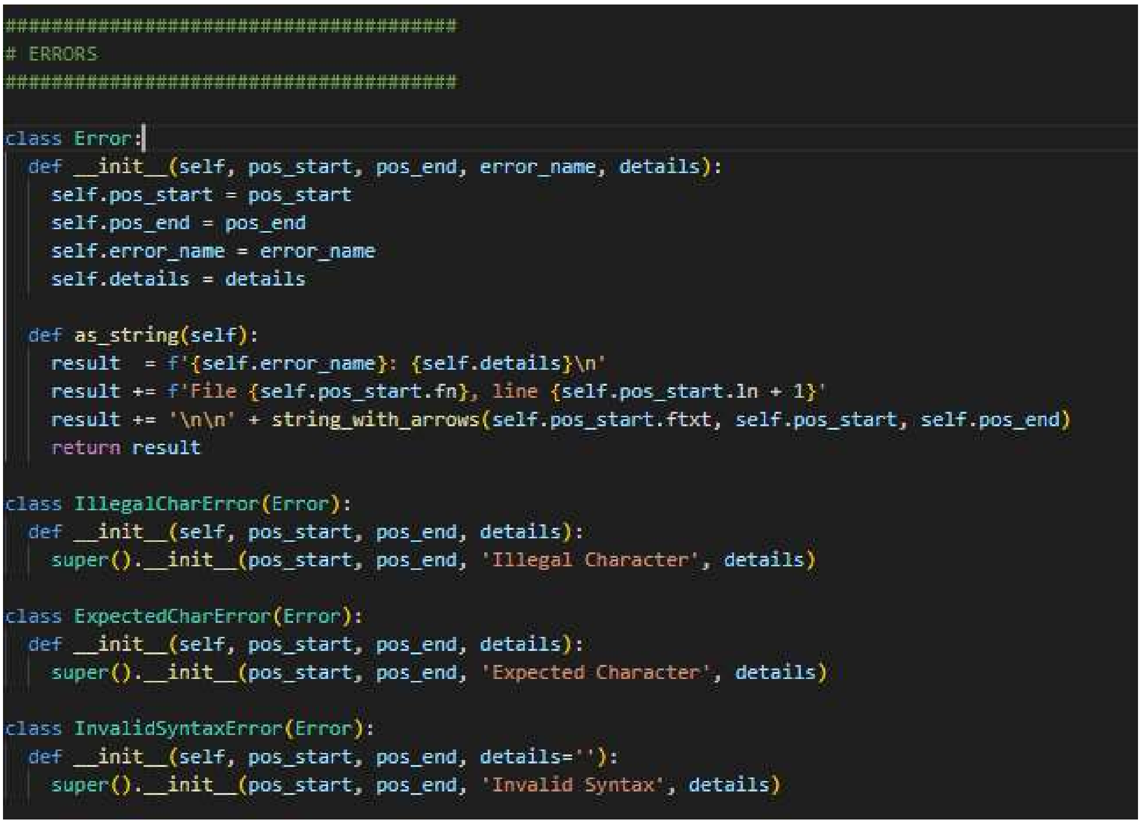
Constant:



```
########################################
# CONSTANTS
########################################

DIGITS = '0123456789'
LETTERS = string.ascii_letters
LETTERS_DIGITS = LETTERS + DIGITS
```

The Digits was given and ASCII characters was used for the letters.

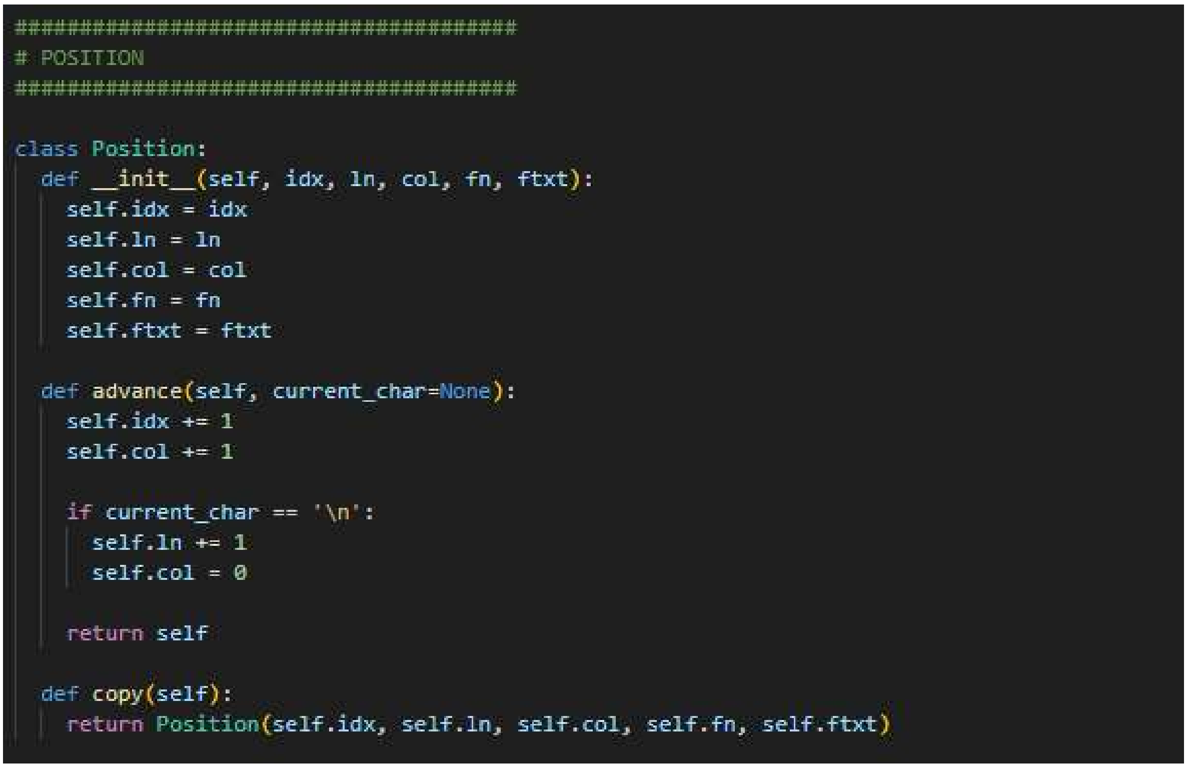
Error:



```
########################################
# ERRORS
########################################

class Error:
  def __init__(self, pos_start, pos_end, error_name, details):
    self.pos_start = pos_start
    self.pos_end = pos_end
    self.error_name = error_name
    self.details = details

  def as_string(self):
    result  = f'{self.error_name}: {self.details}\n'
    result += f'File {self.pos_start.fn}, line {self.pos_start.ln + 1}'
    result += '\n\n' + string_with_arrows(self.pos_start.ftxt, self.pos_start, self.pos_end)
    return result

class IllegalCharError(Error):
  def __init__(self, pos_start, pos_end, details):
    super().__init__(pos_start, pos_end, 'Illegal Character', details)

class ExpectedCharError(Error):
  def __init__(self, pos_start, pos_end, details):
    super().__init__(pos_start, pos_end, 'Expected Character', details)

class InvalidSyntaxError(Error):
  def __init__(self, pos_start, pos_end, details=''):
    super().__init__(pos_start, pos_end, 'Invalid Syntax', details)
```

The basic errors that should be in a programming language are coded. The System throw an exception while we coded with illegal character (more on that in the code)

Position:



```python
#######################################
# POSITION
#######################################

class Position:
  def __init__(self, idx, ln, col, fn, ftxt):
    self.idx = idx
    self.ln = ln
    self.col = col
    self.fn = fn
    self.ftxt = ftxt

  def advance(self, current_char=None):
    self.idx += 1
    self.col += 1

    if current_char == '\n':
      self.ln += 1
      self.col = 0

    return self

  def copy(self):
    return Position(self.idx, self.ln, self.col, self.fn, self.ftxt)
```

My programming language works indents like python. The codes here provide us the indent structure.

Tokens:

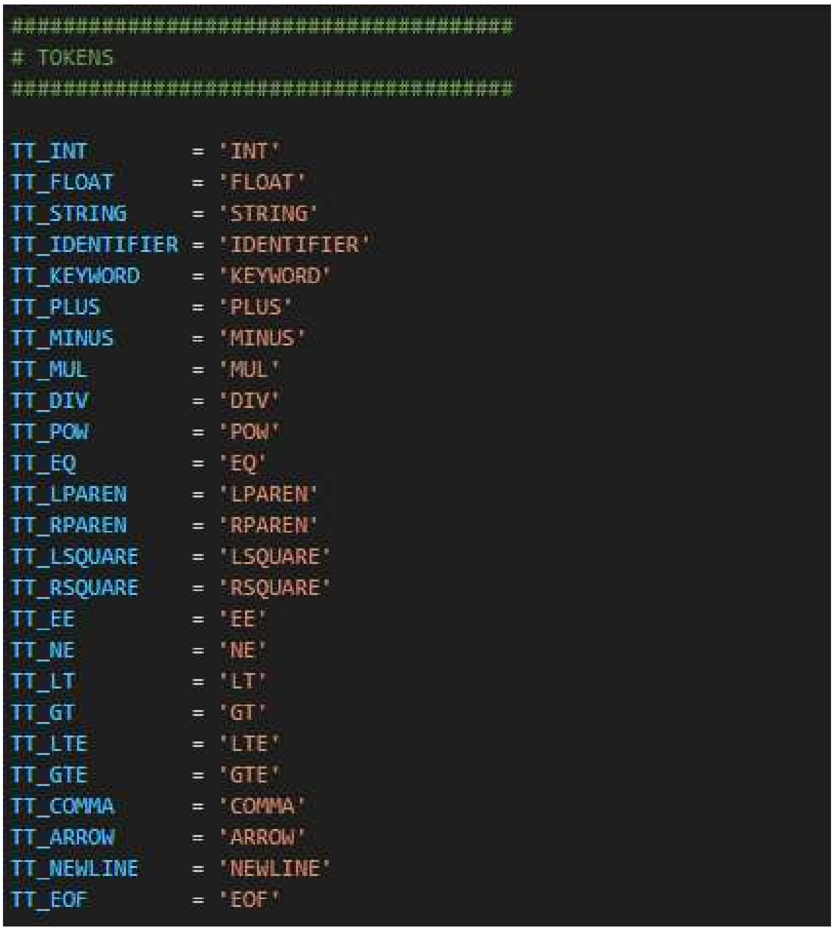
What is token?

In general, a token is an object that represents something else, such as another object (either physical or virtual), or an abstract concept as, for example, a gift is sometimes referred to as a token of the giver's esteem for the recipient. In computers, there are a number of types of tokens.

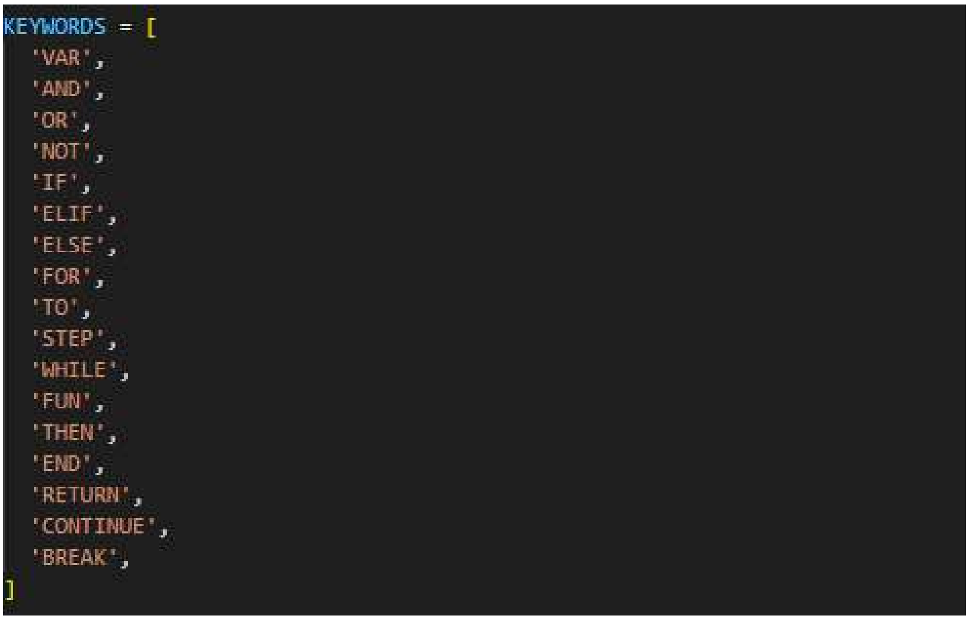
What is keyword?

Keywords are special reserved words that have specific meanings and purposes and can't be used for anything but those specific purposes. These keywords are always available—you'll never have to import them into your code.

These are the corresponding tokens and keywords in my programming language:



```
#######################################
# TOKENS
#######################################

TT_INT        = 'INT'
TT_FLOAT      = 'FLOAT'
TT_STRING     = 'STRING'
TT_IDENTIFIER = 'IDENTIFIER'
TT_KEYWORD    = 'KEYWORD'
TT_PLUS       = 'PLUS'
TT_MINUS      = 'MINUS'
TT_MUL        = 'MUL'
TT_DIV        = 'DIV'
TT_POW        = 'POW'
TT_EQ         = 'EQ'
TT_LPAREN     = 'LPAREN'
TT_RPAREN     = 'RPAREN'
TT_LSQUARE    = 'LSQUARE'
TT_RSQUARE    = 'RSQUARE'
TT_EE         = 'EE'
TT_NE         = 'NE'
TT_LT         = 'LT'
TT_GT         = 'GT'
TT_LTE        = 'LTE'
TT_GTE        = 'GTE'
TT_COMMA      = 'COMMA'
TT_ARROW      = 'ARROW'
TT_NEWLINE    = 'NEWLINE'
TT_EOF        = 'EOF'
```

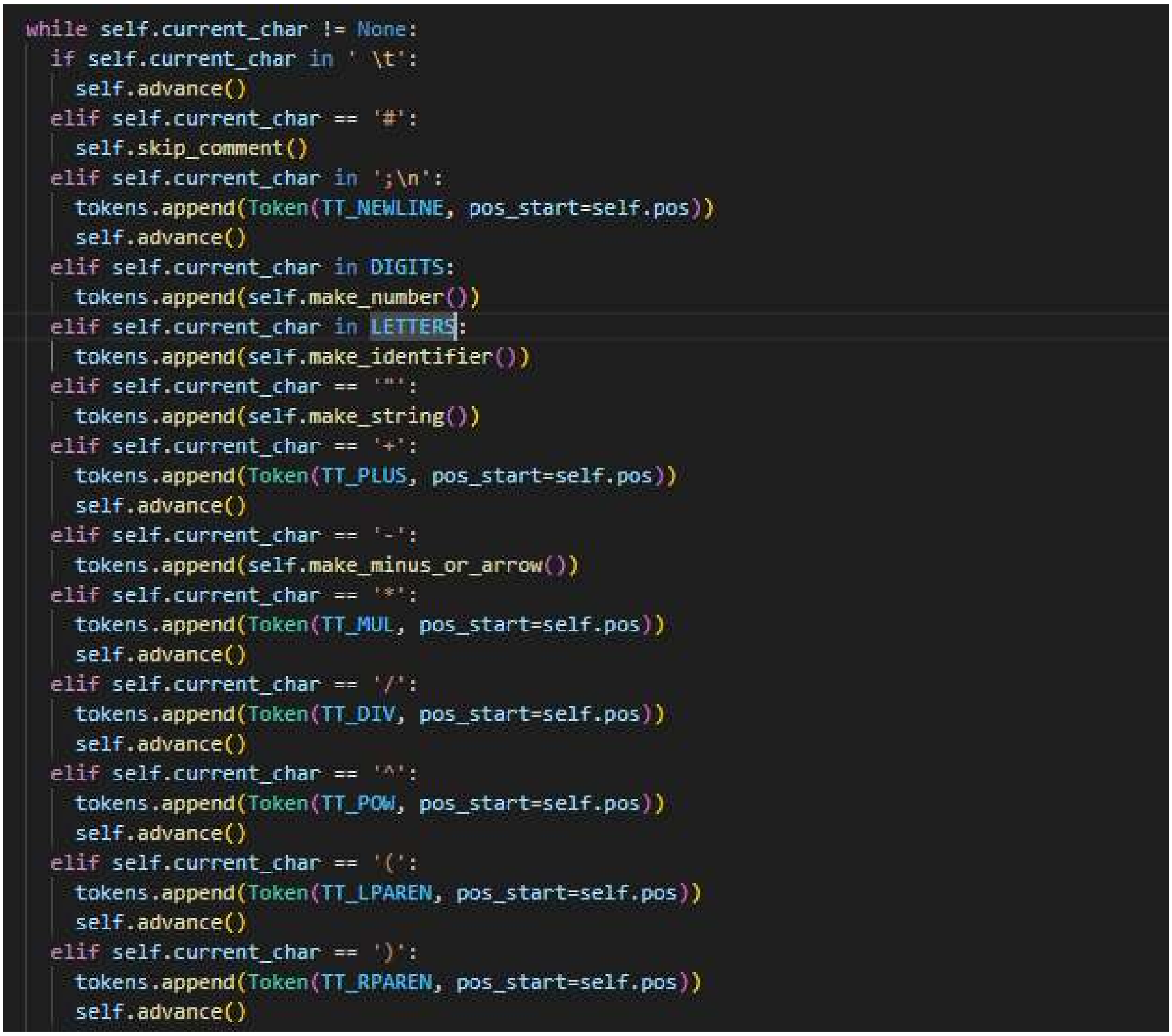


```
KEYWORDS = [
  'VAR',
  'AND',
  'OR',
  'NOT',
  'IF',
  'ELIF',
  'ELSE',
  'FOR',
  'TO',
  'STEP',
  'WHILE',
  'FUN',
  'THEN',
  'END',
  'RETURN',
  'CONTINUE',
  'BREAK',
]
```

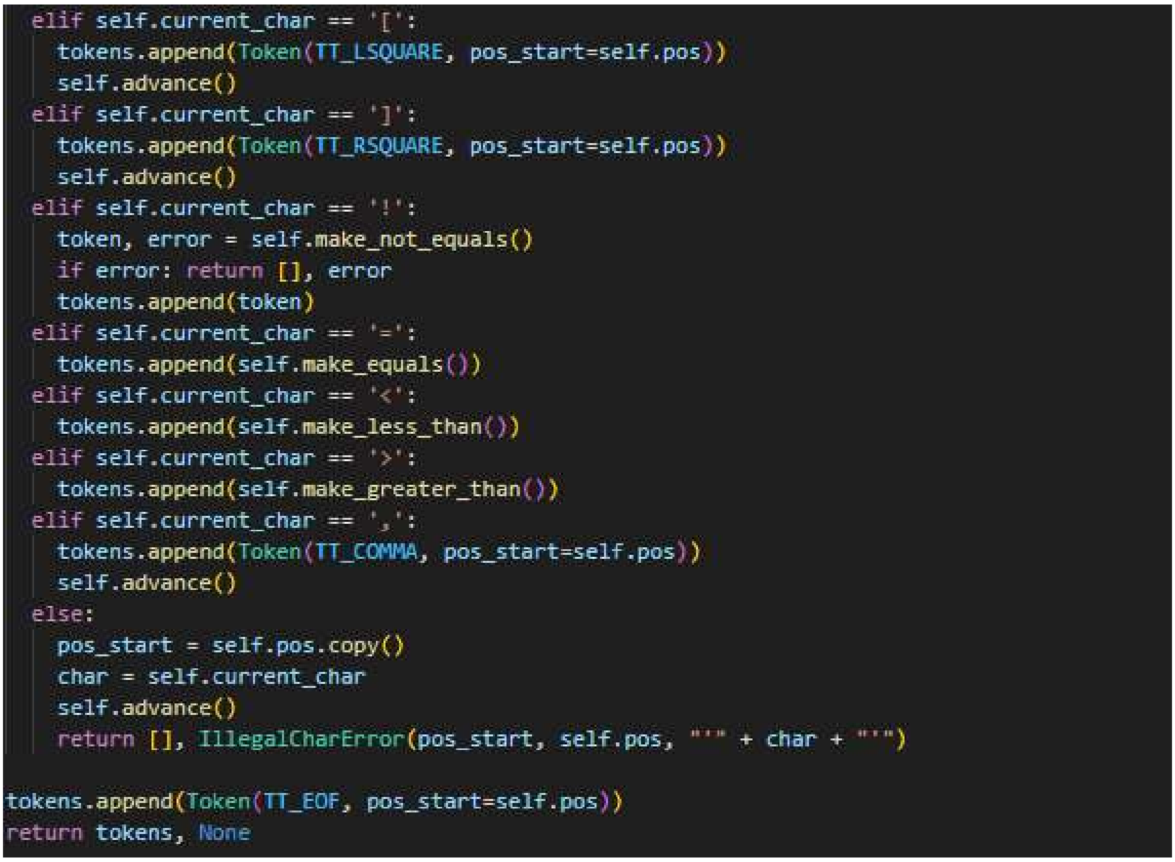
<u>Lexer:</u>

What is lexer in programming?

In computer science, lexical analysis, lexing or tokenization is the process of converting a sequence of characters into a sequence of tokens. A lexer is the part of an interpreter that turns a sequence of characters (plain text) into a sequence of 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))
    self.advance()
  elif self.current_char == '(':
    tokens.append(Token(TT_LPAREN, pos_start=self.pos))
    self.advance()
  elif self.current_char == ')':
    tokens.append(Token(TT_RPAREN, pos_start=self.pos))
    self.advance()
```

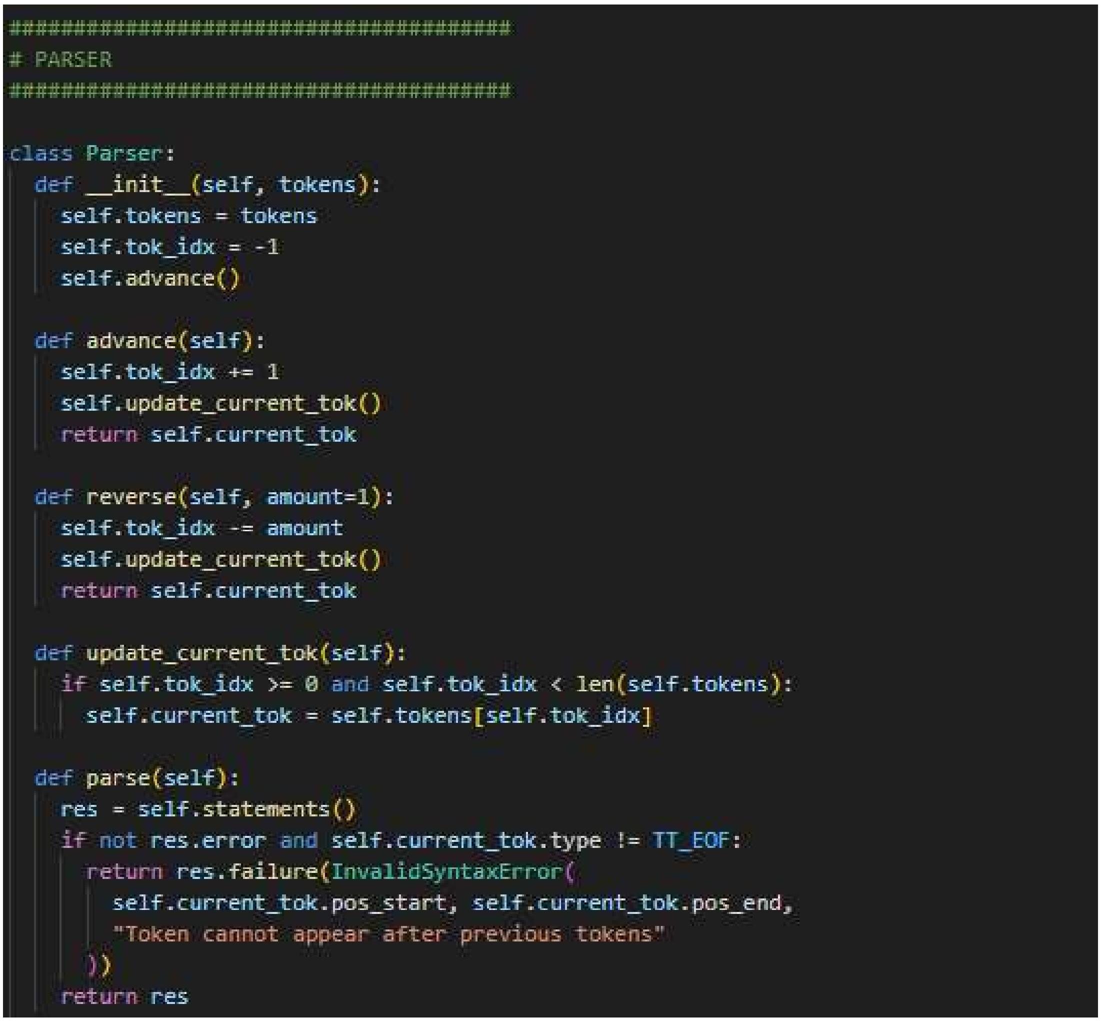


```
  elif self.current_char == '[':
    tokens.append(Token(TT_LSQUARE, pos_start=self.pos))
    self.advance()
  elif self.current_char == ']':
    tokens.append(Token(TT_RSQUARE, pos_start=self.pos))
    self.advance()
  elif self.current_char == '!':
    token, error = self.make_not_equals()
    if error: return [], error
    tokens.append(token)
  elif self.current_char == '=':
    tokens.append(self.make_equals())
  elif self.current_char == '<':
    tokens.append(self.make_less_than())
  elif self.current_char == '>':
    tokens.append(self.make_greater_than())
  elif self.current_char == ',':
    tokens.append(Token(TT_COMMA, pos_start=self.pos))
    self.advance()
  else:
    pos_start = self.pos.copy()
    char = self.current_char
    self.advance()
    return [], IllegalCharError(pos_start, self.pos, "'" + char + "'")

tokens.append(Token(TT_EOF, pos_start=self.pos))
return tokens, None
```

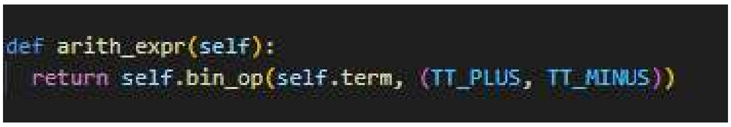
Parser:

In the case of programming languages, a parser is a component of a compiler or interpreter, which parses the source code of a computer programming language to create some form of internal representation; the parser is a key step in the compiler frontend.

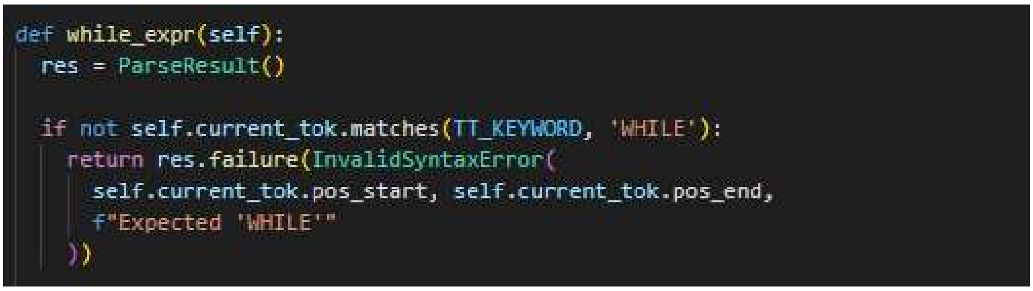
I have processed statement, expression, arithmetic expression, list, if else while statement structures into parser in programming language. You can browse the codes to see more status



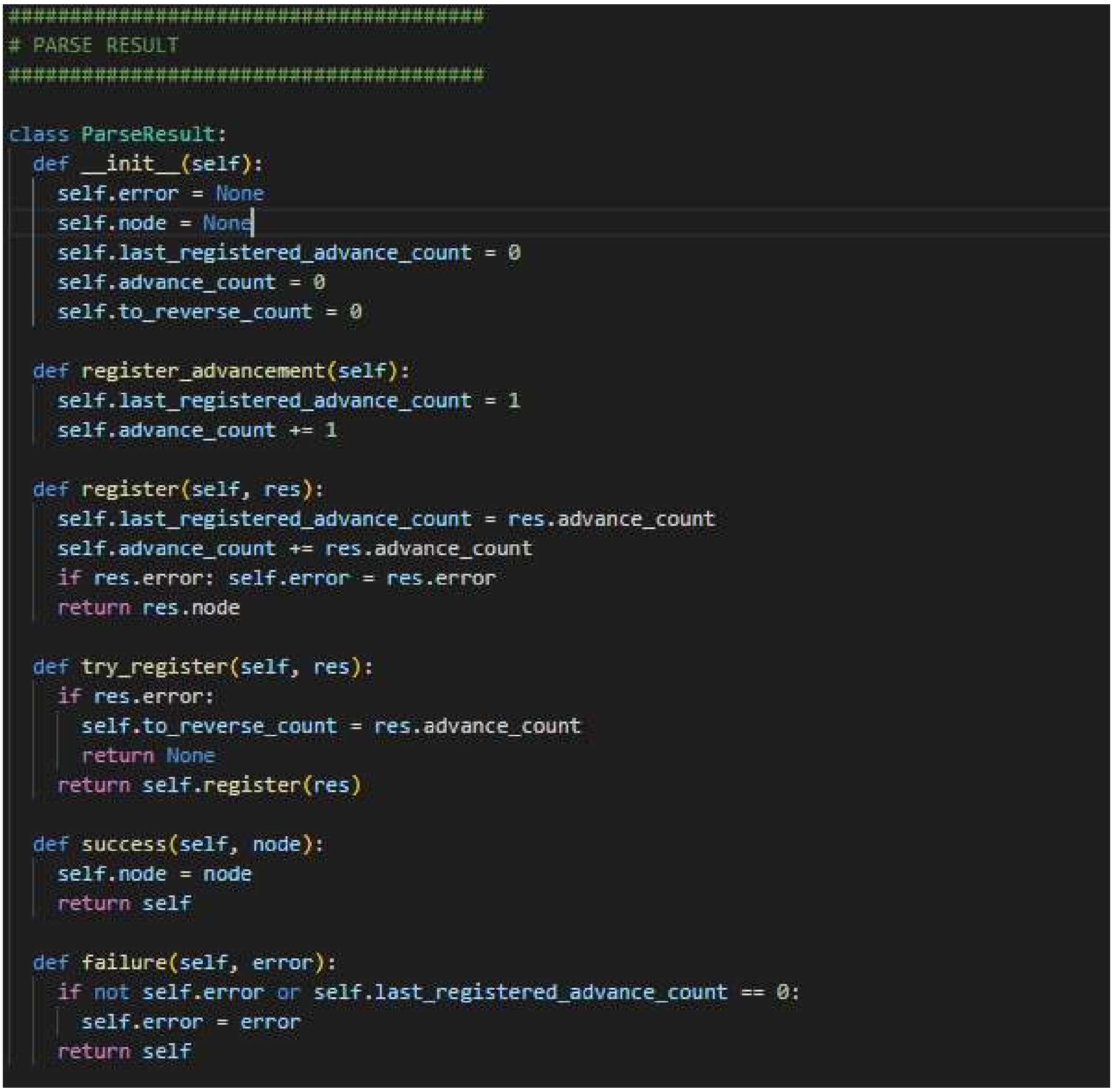
```
#######################################
# PARSER
#######################################

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
```



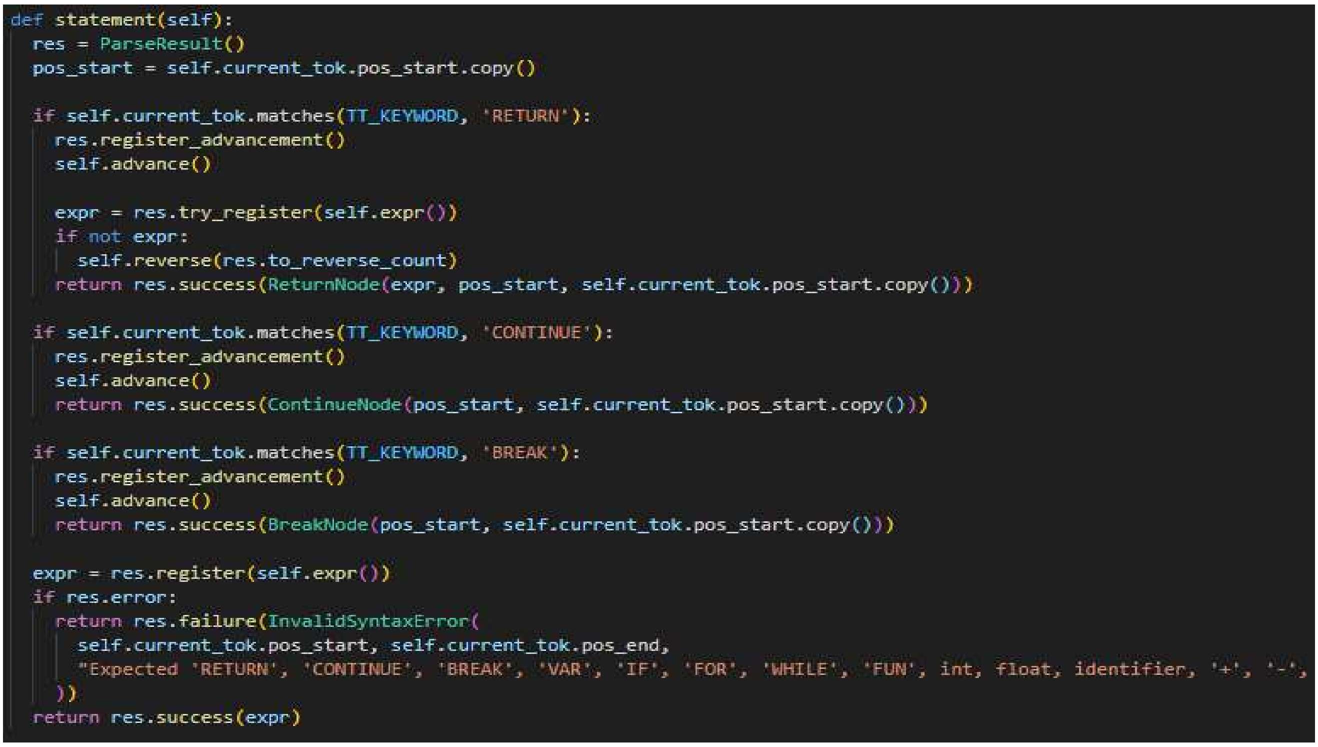
```
def arith_expr(self):
  return self.bin_op(self.term, (TT_PLUS, TT_MINUS))
```



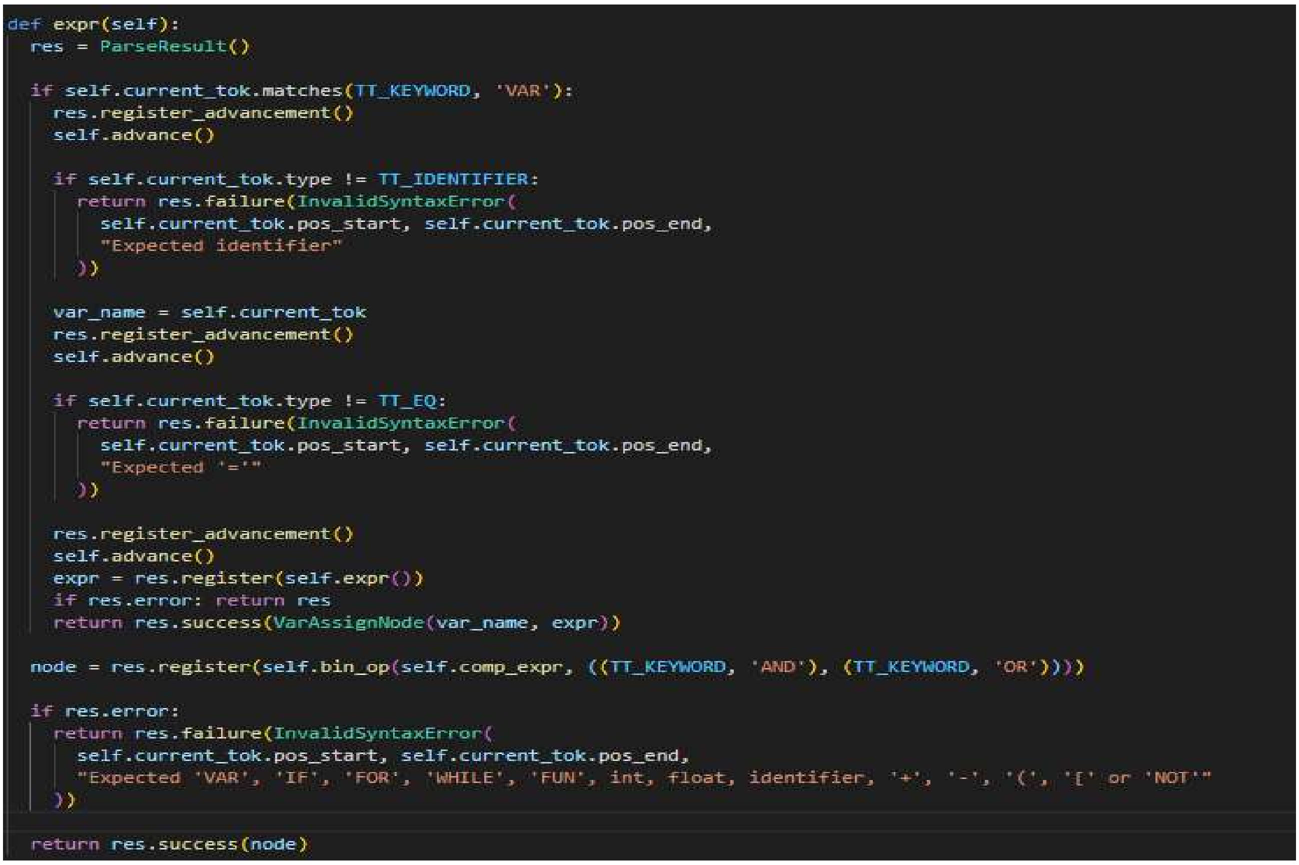
```
def while_expr(self):
  res = ParseResult()

  if not self.current_tok.matches(TT_KEYWORD, 'WHILE'):
    return res.failure(InvalidSyntaxError(
      self.current_tok.pos_start, self.current_tok.pos_end,
      f"Expected 'WHILE'"
    ))
```



```
#######################################
# PARSE RESULT
#######################################

class ParseResult:
  def __init__(self):
    self.error = None
    self.node = None
    self.last_registered_advance_count = 0
    self.advance_count = 0
    self.to_reverse_count = 0

  def register_advancement(self):
    self.last_registered_advance_count = 1
    self.advance_count += 1

  def register(self, res):
    self.last_registered_advance_count = res.advance_count
    self.advance_count += res.advance_count
    if res.error: self.error = res.error
    return res.node

  def try_register(self, res):
    if res.error:
      self.to_reverse_count = res.advance_count
      return None
    return self.register(res)

  def success(self, node):
    self.node = node
    return self

  def failure(self, error):
    if not self.error or self.last_registered_advance_count == 0:
      self.error = error
    return self
```



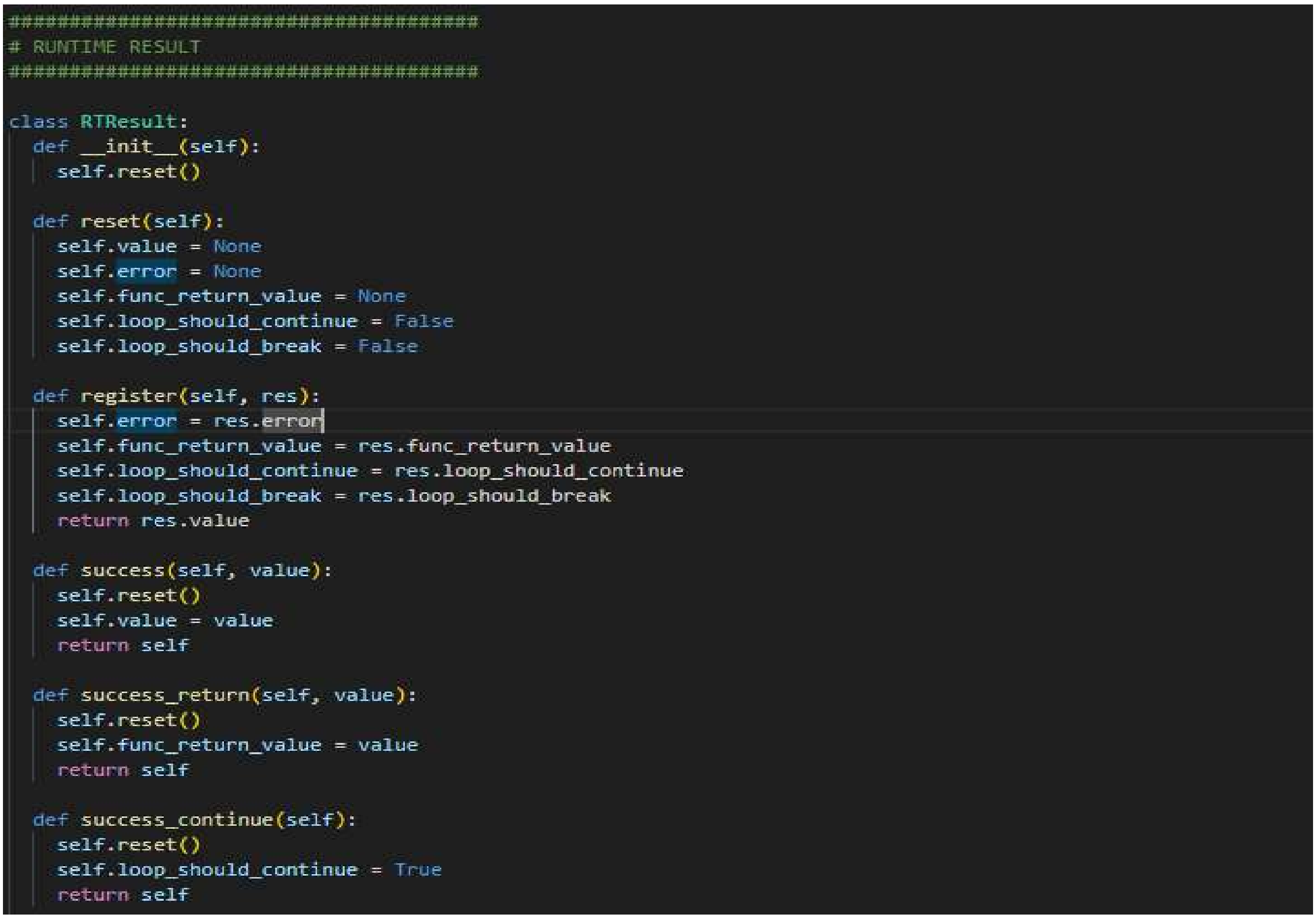
```python
def statement(self):
  res = ParseResult()
  pos_start = self.current_tok.pos_start.copy()

  if self.current_tok.matches(TT_KEYWORD, 'RETURN'):
    res.register_advancement()
    self.advance()

    expr = res.try_register(self.expr())
    if not expr:
      self.reverse(res.to_reverse_count)
    return res.success(ReturnNode(expr, pos_start, self.current_tok.pos_start.copy()))

  if self.current_tok.matches(TT_KEYWORD, 'CONTINUE'):
    res.register_advancement()
    self.advance()
    return res.success(ContinueNode(pos_start, self.current_tok.pos_start.copy()))

  if self.current_tok.matches(TT_KEYWORD, 'BREAK'):
    res.register_advancement()
    self.advance()
    return res.success(BreakNode(pos_start, self.current_tok.pos_start.copy()))

  expr = res.register(self.expr())
  if res.error:
    return res.failure(InvalidSyntaxError(
      self.current_tok.pos_start, self.current_tok.pos_end,
      "Expected 'RETURN', 'CONTINUE', 'BREAK', 'VAR', 'IF', 'FOR', 'WHILE', 'FUN', int, float, identifier, '+', '-',
    ))
  return res.success(expr)
```



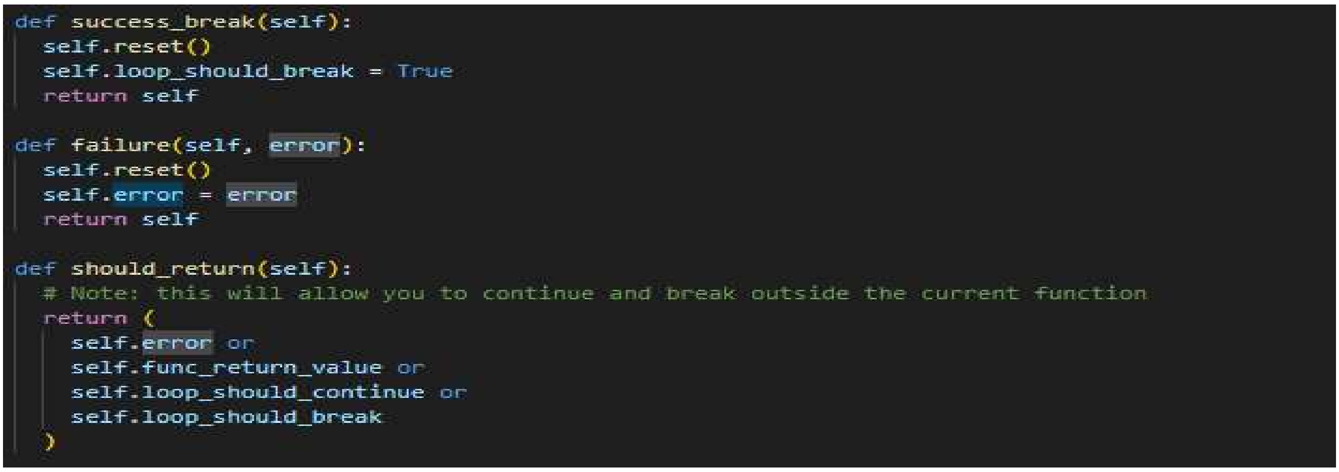
```python
def expr(self):
  res = ParseResult()

  if self.current_tok.matches(TT_KEYWORD, 'VAR'):
    res.register_advancement()
    self.advance()

    if self.current_tok.type != TT_IDENTIFIER:
      return res.failure(InvalidSyntaxError(
        self.current_tok.pos_start, self.current_tok.pos_end,
        "Expected identifier"
      ))

    var_name = self.current_tok
    res.register_advancement()
    self.advance()

    if self.current_tok.type != TT_EQ:
      return res.failure(InvalidSyntaxError(
        self.current_tok.pos_start, self.current_tok.pos_end,
        "Expected '='"
      ))

    res.register_advancement()
    self.advance()
    expr = res.register(self.expr())
    if res.error: return res
    return res.success(VarAssignNode(var_name, expr))

  node = res.register(self.bin_op(self.comp_expr, ((TT_KEYWORD, 'AND'), (TT_KEYWORD, 'OR'))))

  if res.error:
    return res.failure(InvalidSyntaxError(
      self.current_tok.pos_start, self.current_tok.pos_end,
      "Expected 'VAR', 'IF', 'FOR', 'WHILE', 'FUN', int, float, identifier, '+', '-', '(', '[' or 'NOT'"
    ))

  return res.success(node)
```

RunTime Result:

Runtime is a piece of code that implements portions of a programming language's execution model. In doing this, it allows the program to interact with the computing resources it needs to work. Runtimes are often integral parts of the programming language and don't need to be installed separately.



```
#######################################
# RUNTIME RESULT
#######################################

class RTResult:
  def __init__(self):
    self.reset()

  def reset(self):
    self.value = None
    self.error = None
    self.func_return_value = None
    self.loop_should_continue = False
    self.loop_should_break = False

  def register(self, res):
    self.error = res.error
    self.func_return_value = res.func_return_value
    self.loop_should_continue = res.loop_should_continue
    self.loop_should_break = res.loop_should_break
    return res.value

  def success(self, value):
    self.reset()
    self.value = value
    return self

  def success_return(self, value):
    self.reset()
    self.func_return_value = value
    return self

  def success_continue(self):
    self.reset()
    self.loop_should_continue = True
    return self
```

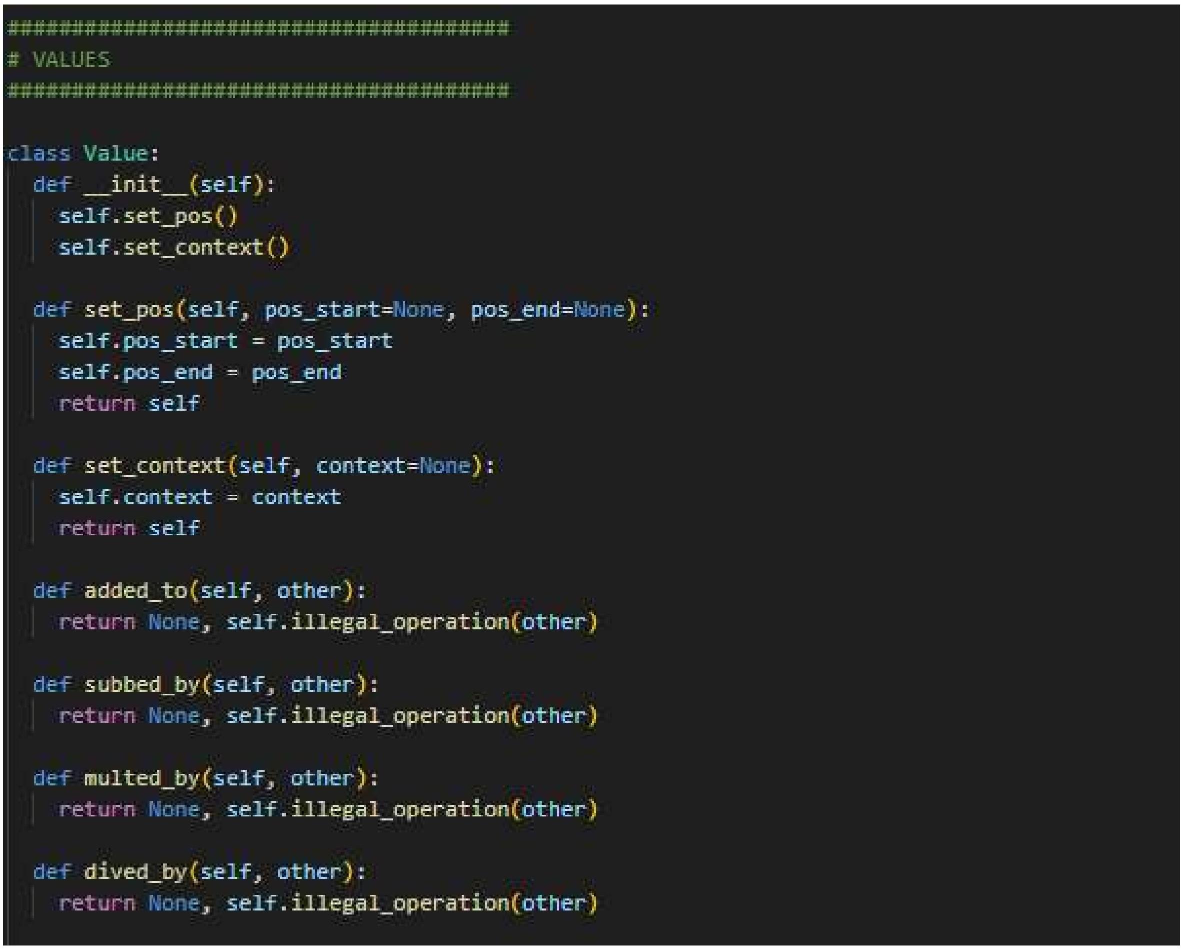


```
def success_break(self):
  self.reset()
  self.loop_should_break = True
  return self

def failure(self, error):
  self.reset()
  self.error = error
  return self

def should_return(self):
  # Note: this will allow you to continue and break outside the current function
  return (
    self.error or
    self.func_return_value or
    self.loop_should_continue or
    self.loop_should_break
  )
```

Values:

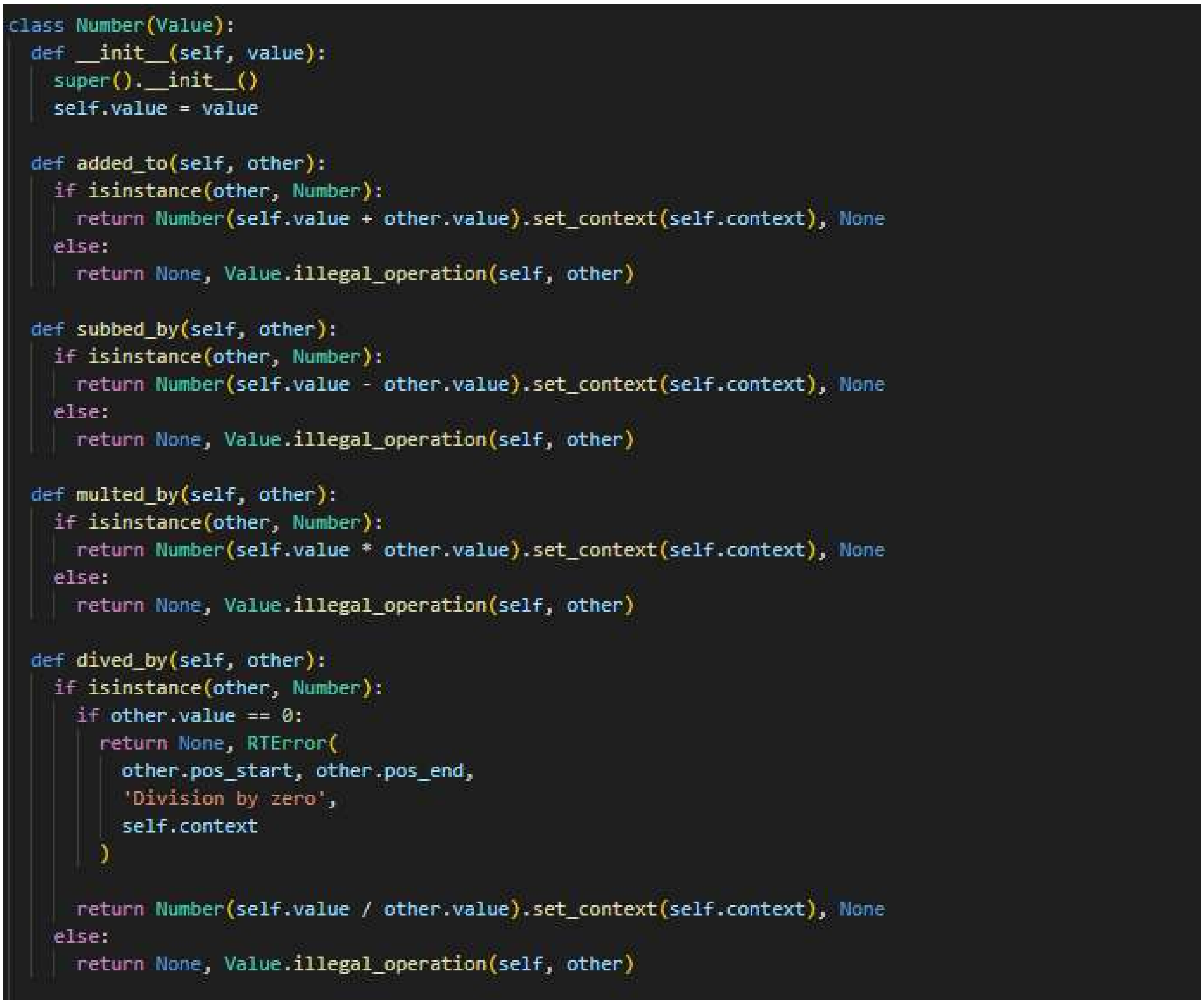
In computer science, a value is the representation of some entity that can be manipulated by a program. The members of a type are the values of that type. The "value of a variable" is given by the corresponding mapping in the environment.

This programming language has number, string, list, function values.



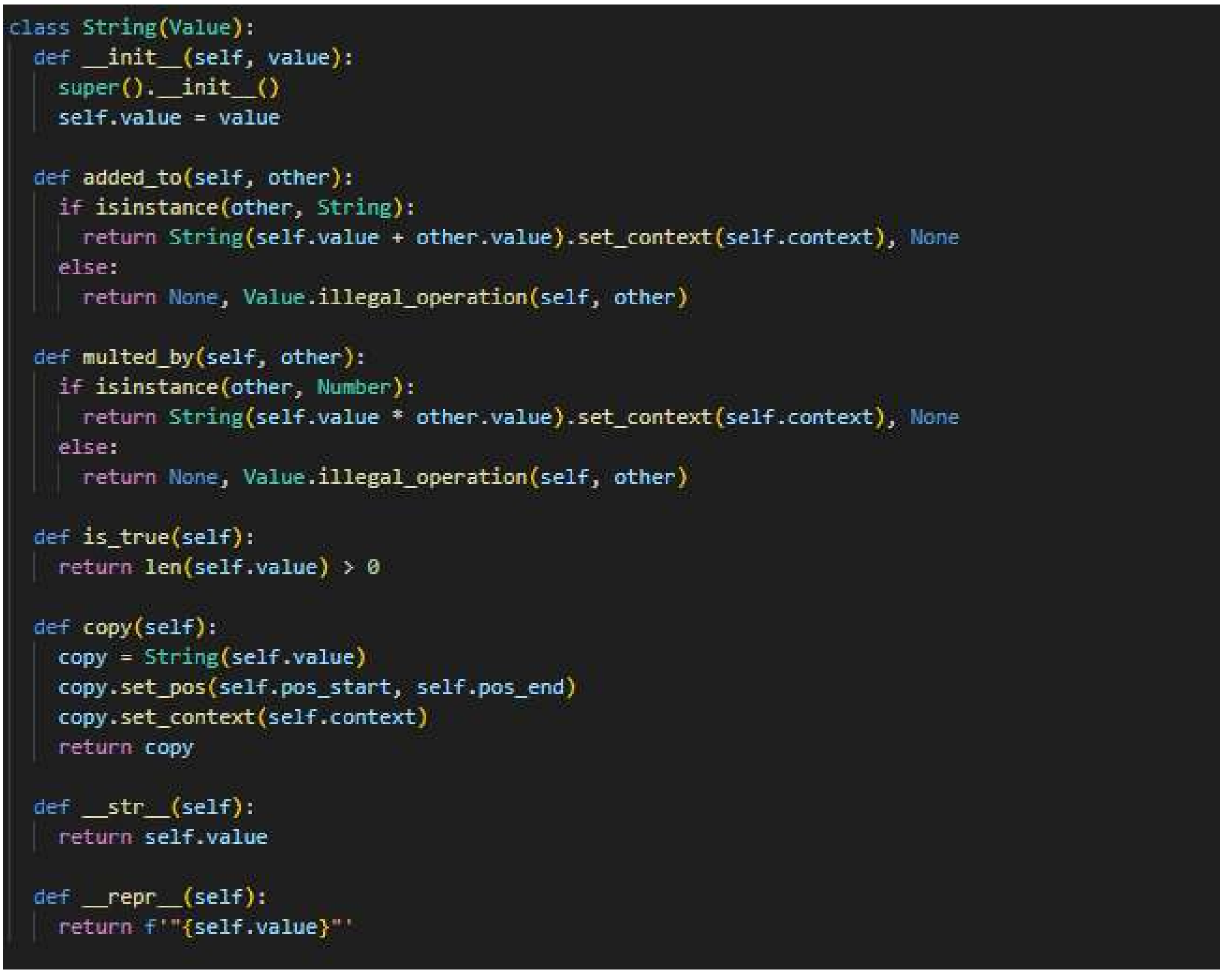
```
########################################
# VALUES
########################################

class Value:
  def __init__(self):
    self.set_pos()
    self.set_context()

  def set_pos(self, pos_start=None, pos_end=None):
    self.pos_start = pos_start
    self.pos_end = pos_end
    return self

  def set_context(self, context=None):
    self.context = context
    return self

  def added_to(self, other):
    return None, self.illegal_operation(other)

  def subbed_by(self, other):
    return None, self.illegal_operation(other)

  def multed_by(self, other):
    return None, self.illegal_operation(other)

  def dived_by(self, other):
    return None, self.illegal_operation(other)
```

Class of Number take the Value as parameter because Value class is the base structure of all value type and it has its own special functions.



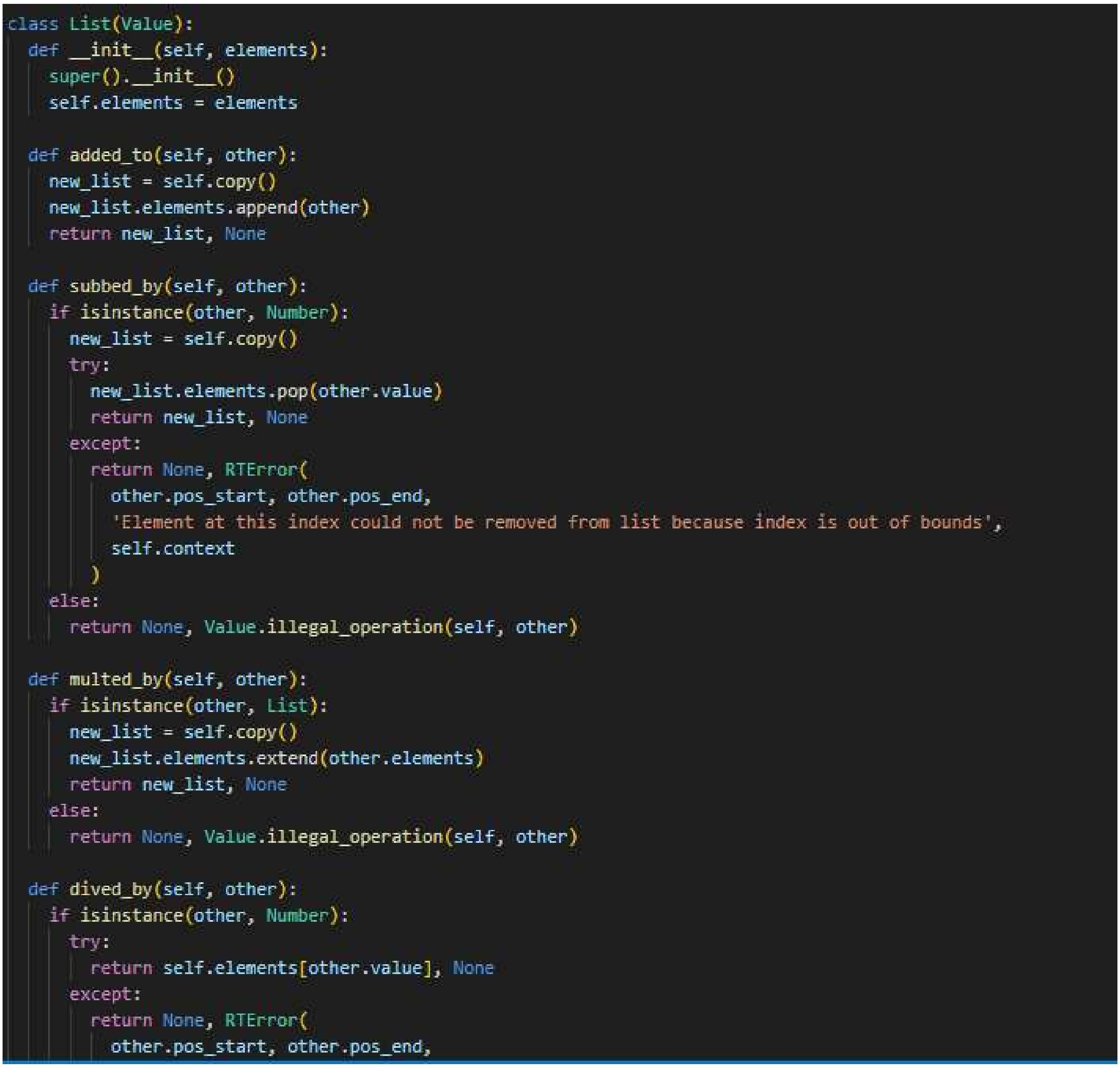
```
class Number(Value):
  def __init__(self, value):
    super().__init__()
    self.value = value

  def added_to(self, other):
    if isinstance(other, Number):
      return Number(self.value + other.value).set_context(self.context), None
    else:
      return None, Value.illegal_operation(self, other)

  def subbed_by(self, other):
    if isinstance(other, Number):
      return Number(self.value - other.value).set_context(self.context), None
    else:
      return None, Value.illegal_operation(self, other)

  def multed_by(self, other):
    if isinstance(other, Number):
      return Number(self.value * other.value).set_context(self.context), None
    else:
      return None, Value.illegal_operation(self, other)

  def dived_by(self, other):
    if isinstance(other, Number):
      if other.value == 0:
        return None, RTError(
          other.pos_start, other.pos_end,
          'Division by zero',
          self.context
        )

      return Number(self.value / other.value).set_context(self.context), None
    else:
      return None, Value.illegal_operation(self, other)
```

Class of String take the Value as parameter because Value class is the base structure of all value type and it has its own special functions.



```
class String(Value):
  def __init__(self, value):
    super().__init__()
    self.value = value

  def added_to(self, other):
    if isinstance(other, String):
      return String(self.value + other.value).set_context(self.context), None
    else:
      return None, Value.illegal_operation(self, other)

  def multed_by(self, other):
    if isinstance(other, Number):
      return String(self.value * other.value).set_context(self.context), None
    else:
      return None, Value.illegal_operation(self, other)

  def is_true(self):
    return len(self.value) > 0

  def copy(self):
    copy = String(self.value)
    copy.set_pos(self.pos_start, self.pos_end)
    copy.set_context(self.context)
    return copy

  def __str__(self):
    return self.value

  def __repr__(self):
    return f'"{self.value}"'
```

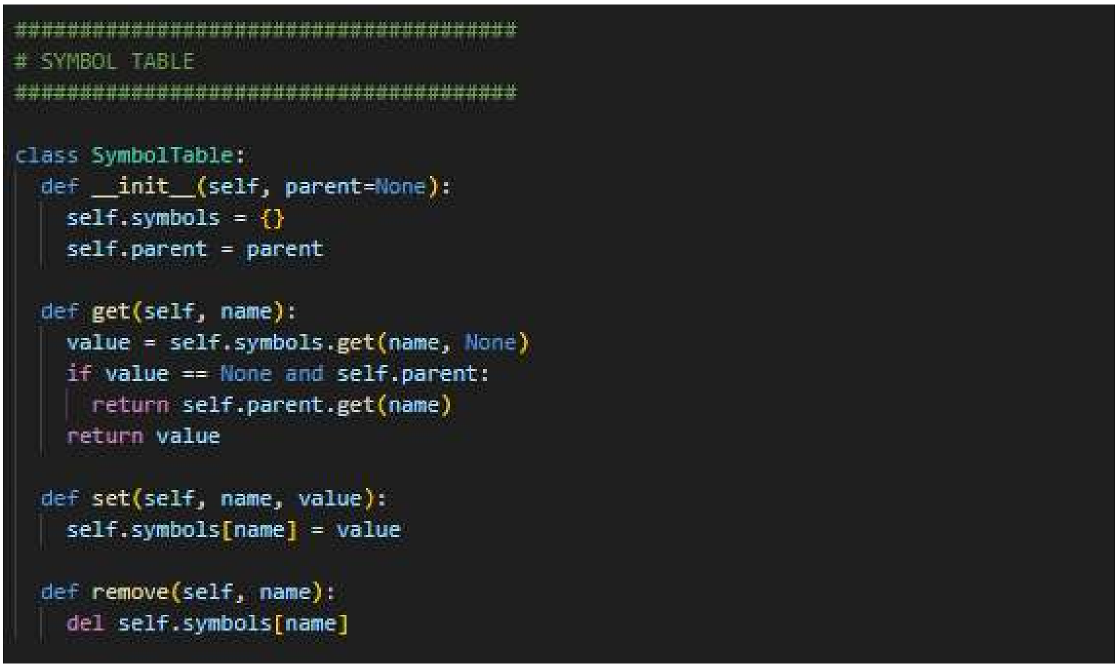
Class of List take the Value as parameter because Value class is the base structure of all value type and it has its own special functions.



```
class List(Value):
  def __init__(self, elements):
    super().__init__()
    self.elements = elements

  def added_to(self, other):
    new_list = self.copy()
    new_list.elements.append(other)
    return new_list, None

  def subbed_by(self, other):
    if isinstance(other, Number):
      new_list = self.copy()
      try:
        new_list.elements.pop(other.value)
        return new_list, None
      except:
        return None, RTError(
          other.pos_start, other.pos_end,
          'Element at this index could not be removed from list because index is out of bounds',
          self.context
        )
    else:
      return None, Value.illegal_operation(self, other)

  def multed_by(self, other):
    if isinstance(other, List):
      new_list = self.copy()
      new_list.elements.extend(other.elements)
      return new_list, None
    else:
      return None, Value.illegal_operation(self, other)

  def dived_by(self, other):
    if isinstance(other, Number):
      try:
        return self.elements[other.value], None
      except:
        return None, RTError(
          other.pos_start, other.pos_end,
```

Symbol Table:

In computer science, a symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier (or symbols), constants, procedures and functions in a program's source code is associated with information relating to its declaration or appearance in the source.

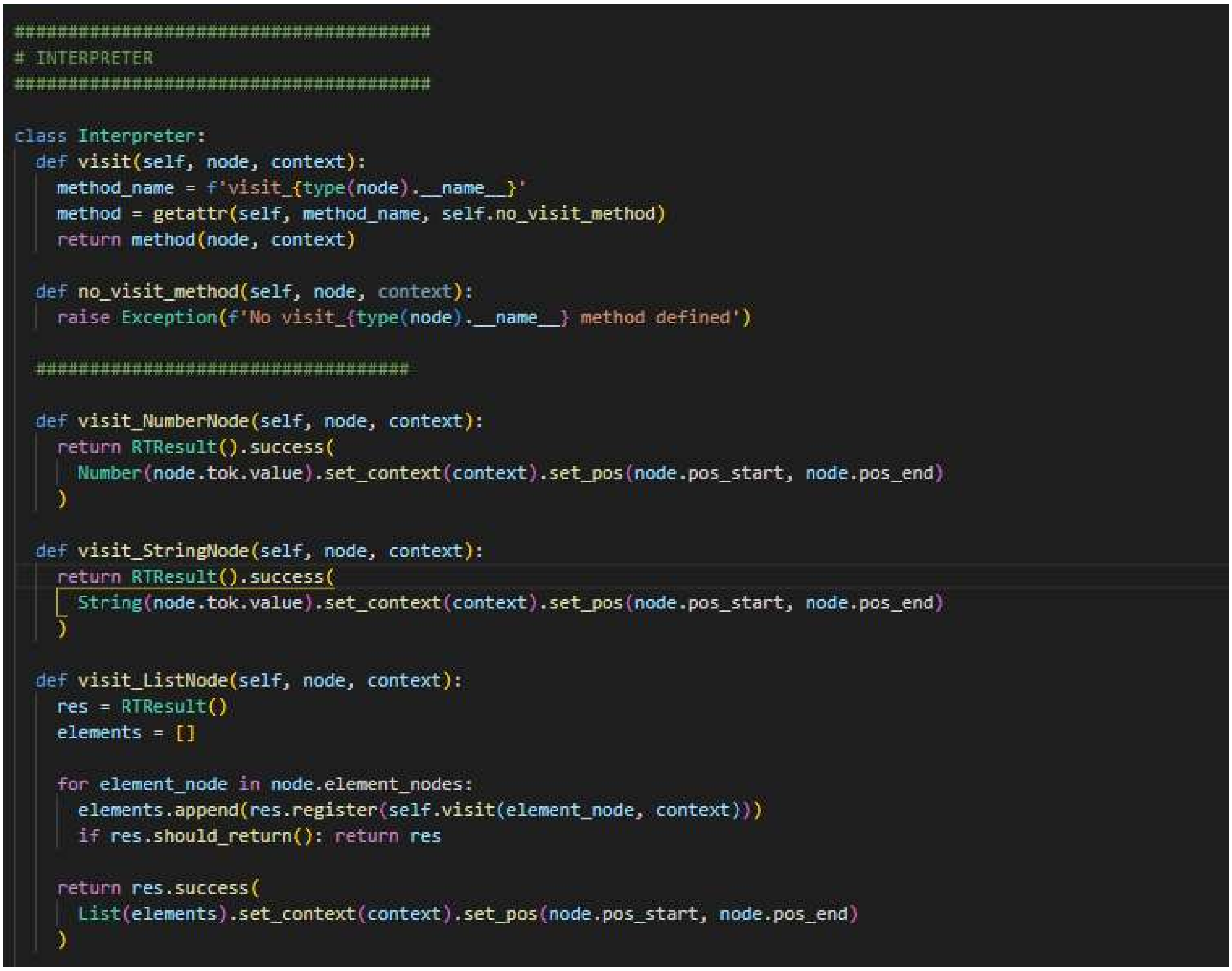


```
#######################################
# SYMBOL TABLE
#######################################

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:

In computer science, an interpreter is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program. An interpreter generally uses one of the following strategies for program execution:

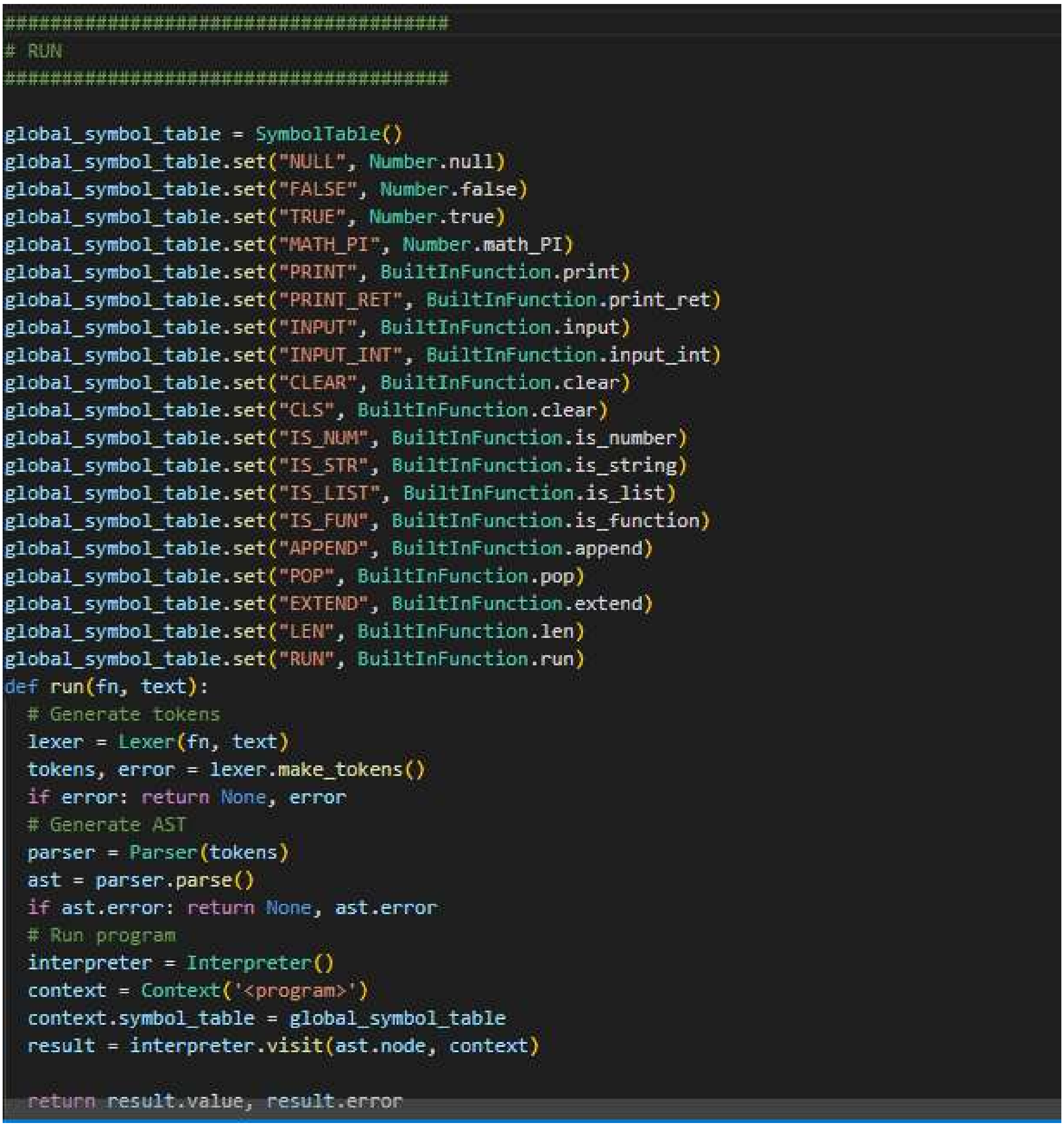
- Parse the source code and perform its behaviour directly;
- Translate source code into some efficient intermediate representation or object code and immediately execute that;
- Explicitly execute stored precompiled bytecode made by a compiler and matched with the interpreter Virtual Machine.



```
#######################################
# 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)
    )
```

Run:

We have come to the end of the design of the programming language I am designing, and here are the codes needed to run it next. We call and use the symbol table, lexer, parser, error, interpreter, context, token classes that I created in this function.

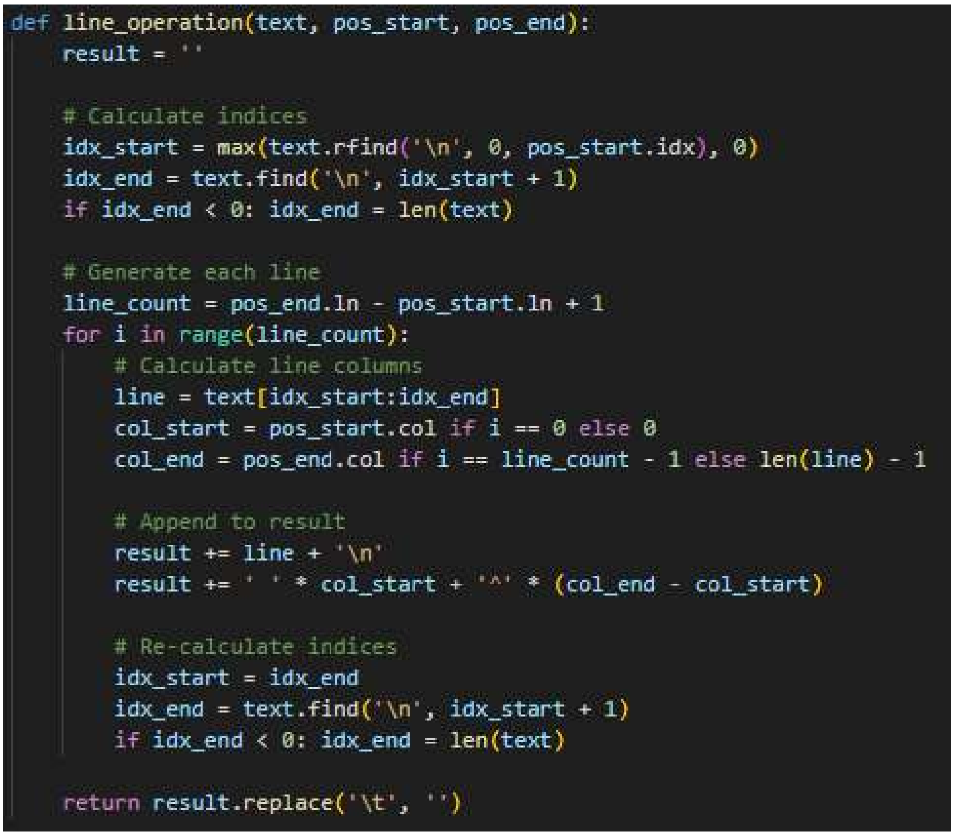


```
#######################################
# RUN
#######################################

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
```

1.2 – Line_operation

This part contains only one function. The purposes of this function are

- Calculate indices
- Generate each line
- Calculate line columns

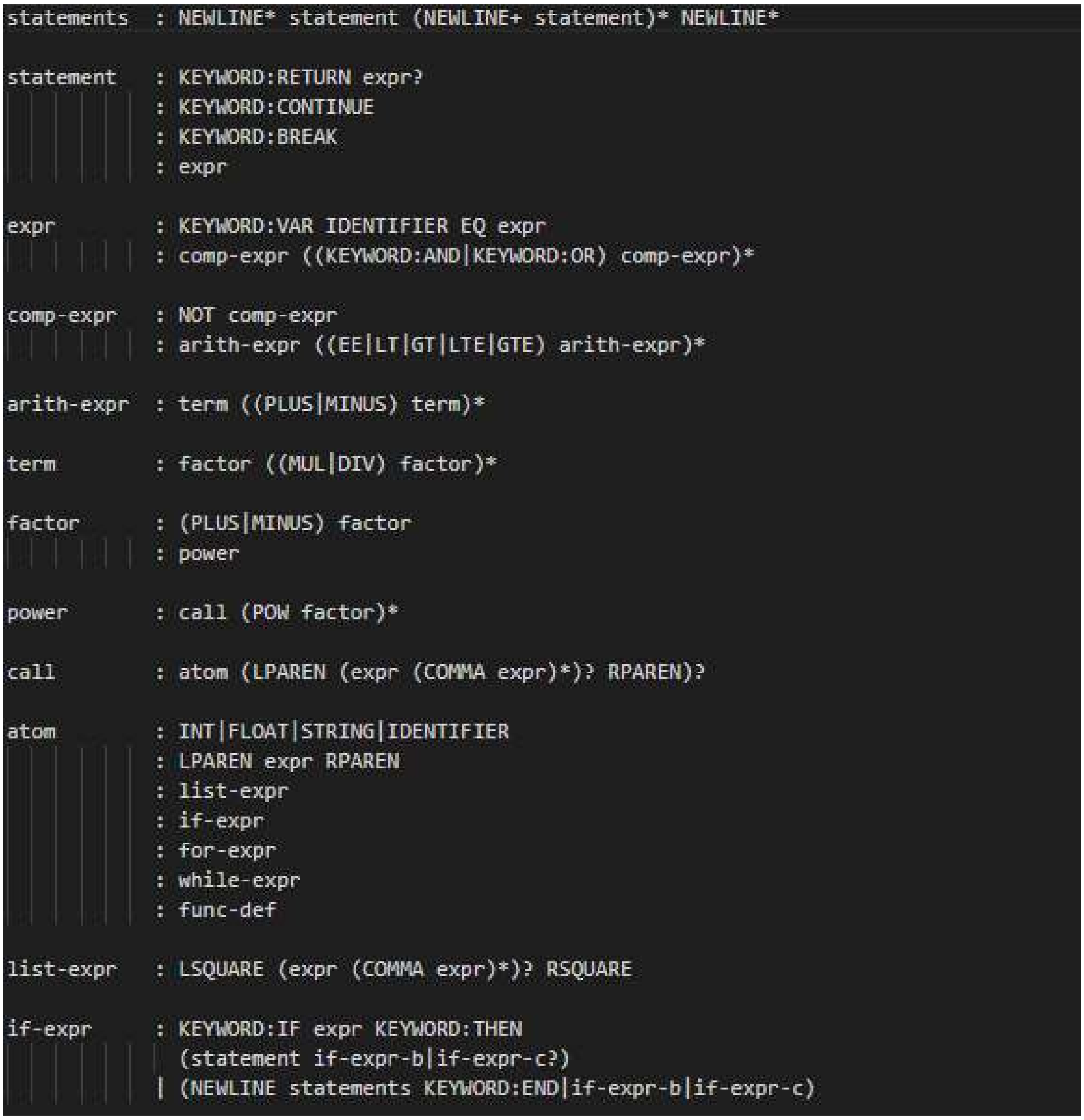


```python
def line_operation(text, pos_start, pos_end):
    result = ''

    # Calculate indices
    idx_start = max(text.rfind('\n', 0, pos_start.idx), 0)
    idx_end = text.find('\n', idx_start + 1)
    if idx_end < 0: idx_end = len(text)

    # Generate each line
    line_count = pos_end.ln - pos_start.ln + 1
    for i in range(line_count):
        # Calculate line columns
        line = text[idx_start:idx_end]
        col_start = pos_start.col if i == 0 else 0
        col_end = pos_end.col if i == line_count - 1 else len(line) - 1

        # Append to result
        result += line + '\n'
        result += ' ' * col_start + '^' * (col_end - col_start)

        # Re-calculate indices
        idx_start = idx_end
        idx_end = text.find('\n', idx_start + 1)
        if idx_end < 0: idx_end = len(text)

    return result.replace('\t', '')
```

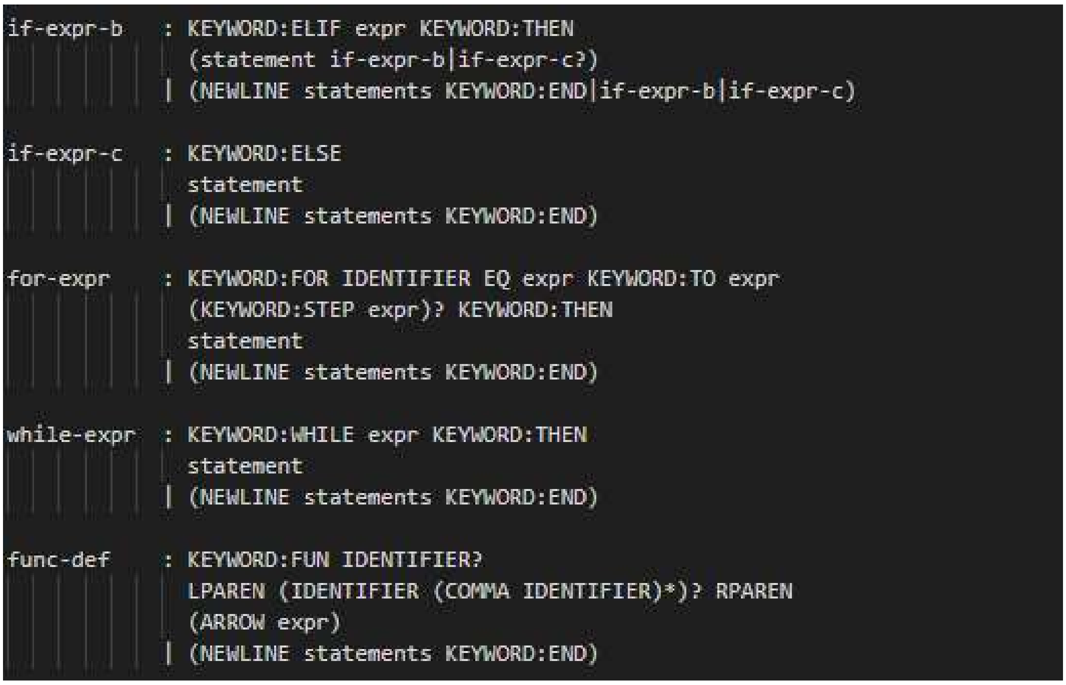
1.3 - Grammar

A Programming Language Grammar is a set of instructions about how to write statements that are valid for that programming language.

The instructions are given in the form of rules that specify how characters and words can be put one after the other, to form valid statements (also called sentences).



```
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)
```

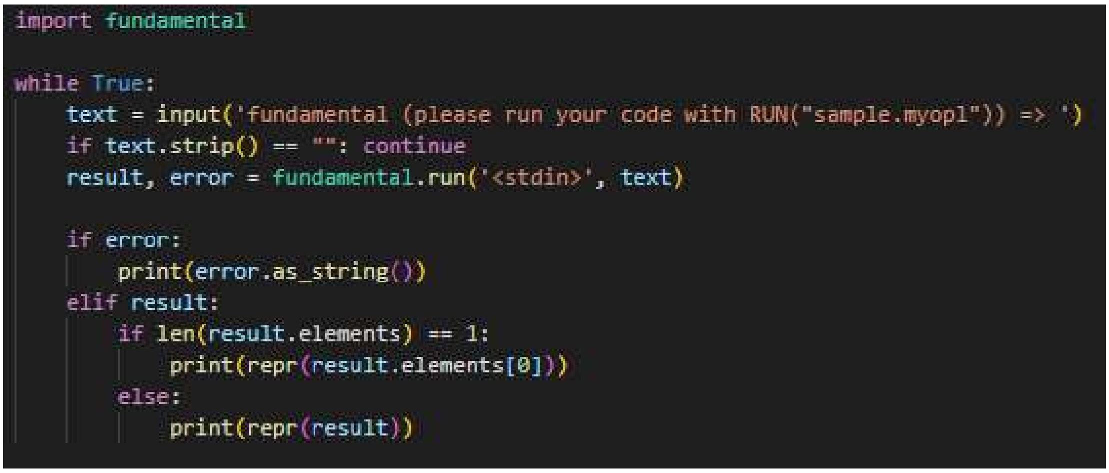


```
if-expr-b   : KEYWORD:ELIF expr KEYWORD:THEN
              (statement if-expr-b|if-expr-c?)
            | (NEWLINE statements KEYWORD:END|if-expr-b|if-expr-c)

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)
```

In computer science, Backus–Naur form or Backus normal form (BNF) is a metasyntax notation for context-free grammars, often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols.

1.4 - Initiator

I wrote a shell program to run my programming language and this program imports the basic file and makes it controllable from the terminal.



```python
import fundamental

while True:
    text = input('fundamental (please run your code with RUN("sample.myopl")) => ')
    if text.strip() == "": continue
    result, error = fundamental.run('<stdin>', text)

    if error:
        print(error.as_string())
    elif result:
        if len(result.elements) == 1:
            print(repr(result.elements[0]))
        else:
            print(repr(result))
```

2 – Test and Output

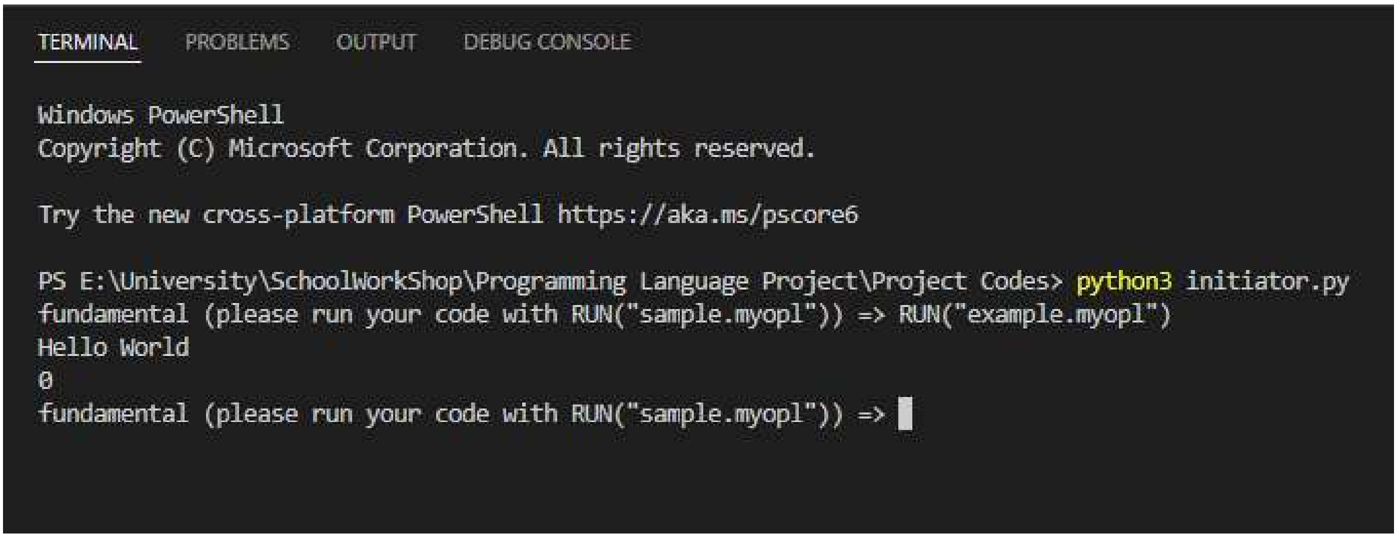
The project is fully compilable and runnable.

These project code files read by your modified lex and syntax analysis source codes.

The modified code gives a proper output for the input source codes of my language

To test the programming language, you need to do the following steps:

- open terminal on visual studio code
- write this code in terminal python3 shell.py and thanks to shell programming language starts
- We run the file created with this programming language with this command RUN("example.myopl") and get the output.



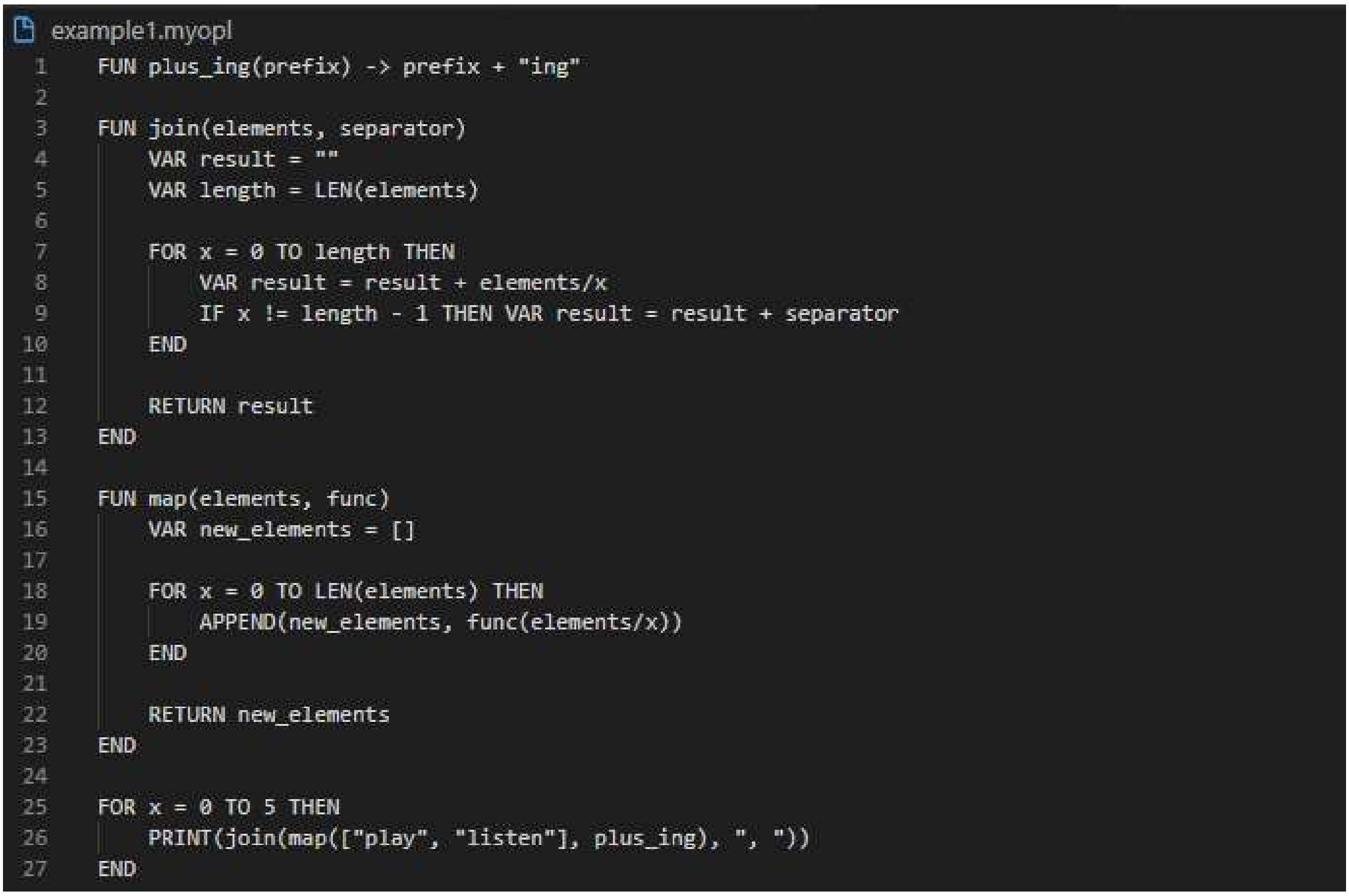
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS E:\University\SchoolWorkShop\Programming Language Project\Project Codes> python3 initiator.py
fundamental (please run your code with RUN("sample.myopl")) => RUN("example.myopl")
Hello World
0
fundamental (please run your code with RUN("sample.myopl")) => 
```

3 – Information of the Programming Language

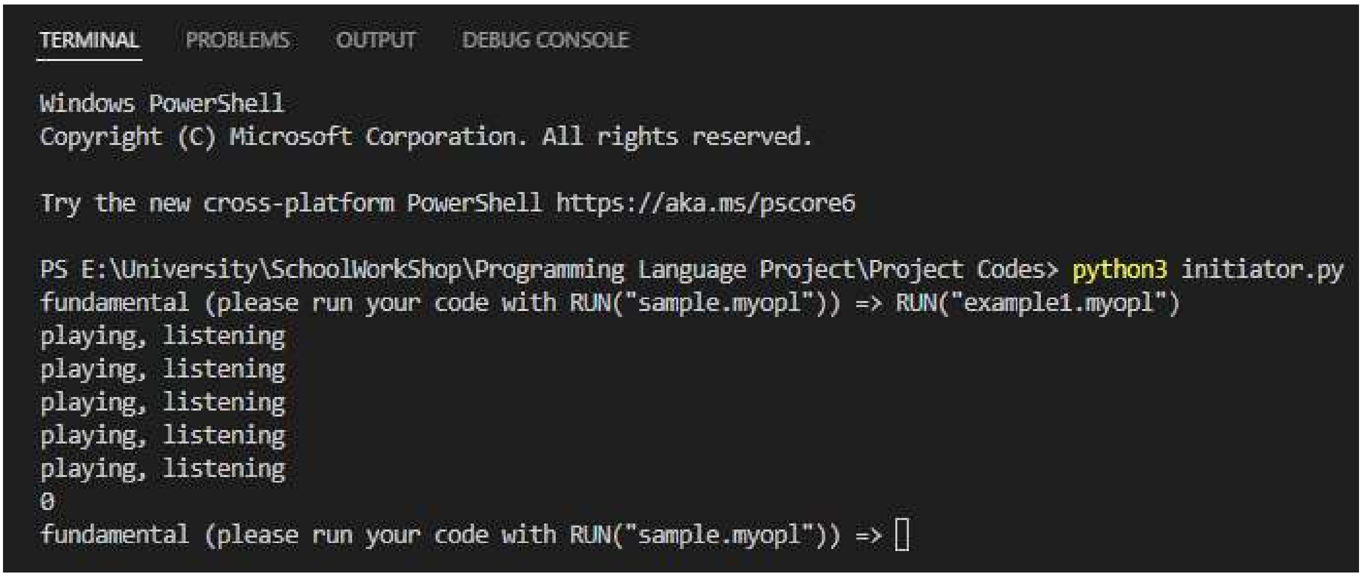
- The programming language has a comment line and is created with the # sign.
- We can use \t and \n for rows operation.
- We should write it in an **indent** structure with function and loop.
- We must use VAR and = keywords to assignment an object.
- Addition + , subtraction - , multiplication * , division / , power ^ , equal , greater > , less < , greater /less and equal >= , <= operations can be done.
- The function must be created with FUN and closed with END.
- We can use RETURN at the end of the function.
- The loop must be created with FOR , WHILE and it closed with END also TO, THEN is can be used in the loop.
- The statements must be created with IF , ELIF (else-if) , ELSE also TO, THEN is can be used in the statement.
- We can use == (if equal) , != (if not equal) , AND , OR in the statements.
- We can use BREAK and CONTINUE keywords in the loop whenever we want.
- We can create a list with [ ] like Python.
- The list class has APPEND(list_name,variable) function to add a variable in the list
- The list class has POP (list_name,variable) function to remove a variable in the list.
- The list class has EXTEND (list_name,variable) function to extend the list.
- The list class has CLEAR (list_name,variable) function to clear the list.
- The programming language has LEN (object) function for the learning the length.
- We can use NULL , FALSE , TRUE structure whenever we need these.
- We get Boolean results about the type of object with these functions IS_NUM(object) , IS_STR(object) , IS_LIST(object) , IS_FUN(object).
- We get data from the users with INPUT() , INPUT_INT().
- We must use PRINT(object) function to print something on the terminal.

3 – Examples

3.1 - Example-1

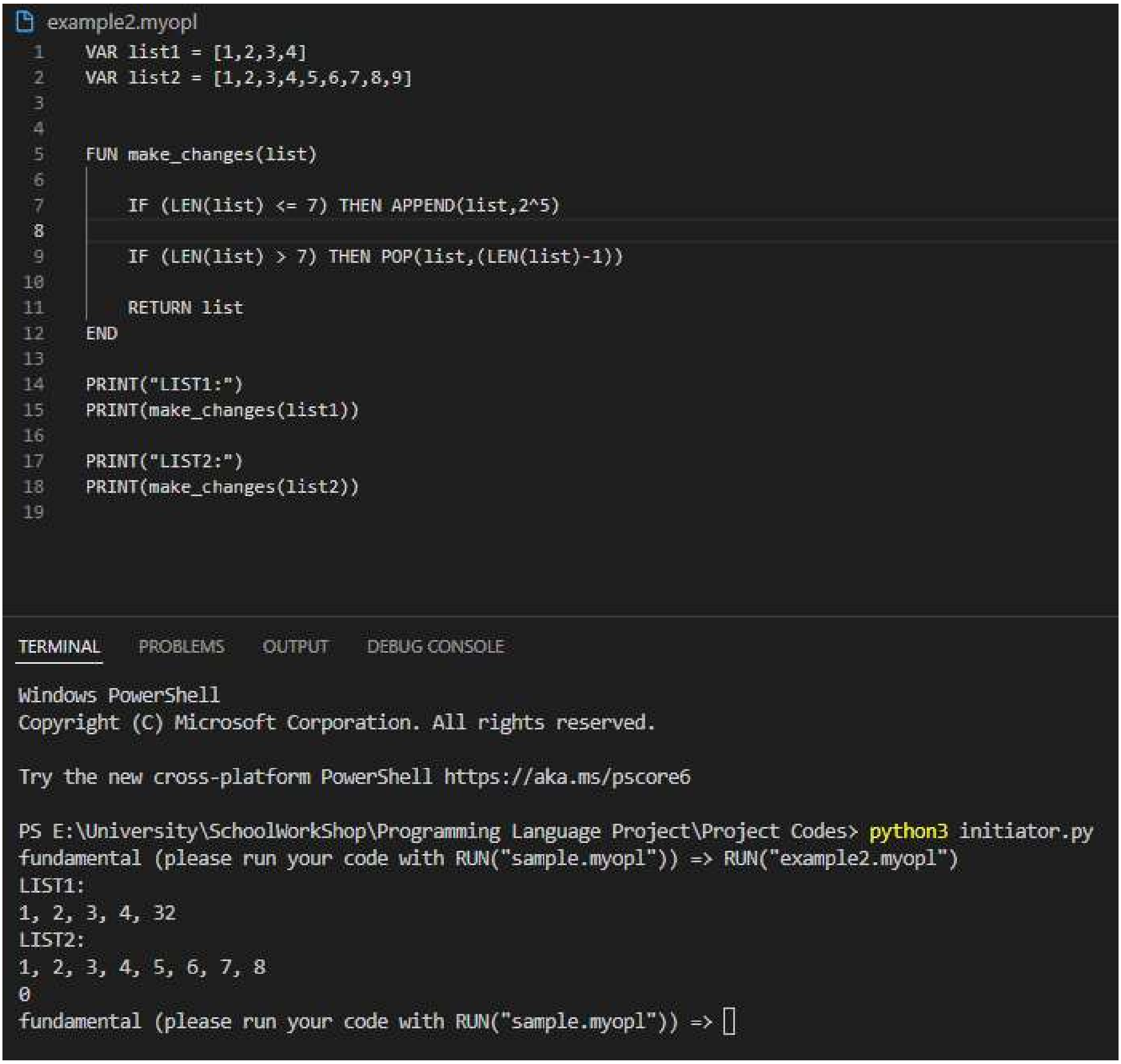


```
FUN plus_ing(prefix) -> prefix + "ing"

FUN join(elements, separator)
    VAR result = ""
    VAR length = LEN(elements)

    FOR x = 0 TO length THEN
        VAR result = result + elements/x
        IF x != length - 1 THEN VAR result = result + separator
    END

    RETURN result
END

FUN map(elements, func)
    VAR new_elements = []

    FOR x = 0 TO LEN(elements) THEN
        APPEND(new_elements, func(elements/x))
    END

    RETURN new_elements
END

FOR x = 0 TO 5 THEN
    PRINT(join(map(["play", "listen"], plus_ing), ", "))
END
```



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS E:\University\SchoolWorkShop\Programming Language Project\Project Codes> python3 initiator.py
fundamental (please run your code with RUN("sample.myopl")) => RUN("example1.myopl")
playing, listening
playing, listening
playing, listening
playing, listening
playing, listening
0
fundamental (please run your code with RUN("sample.myopl")) => 
```

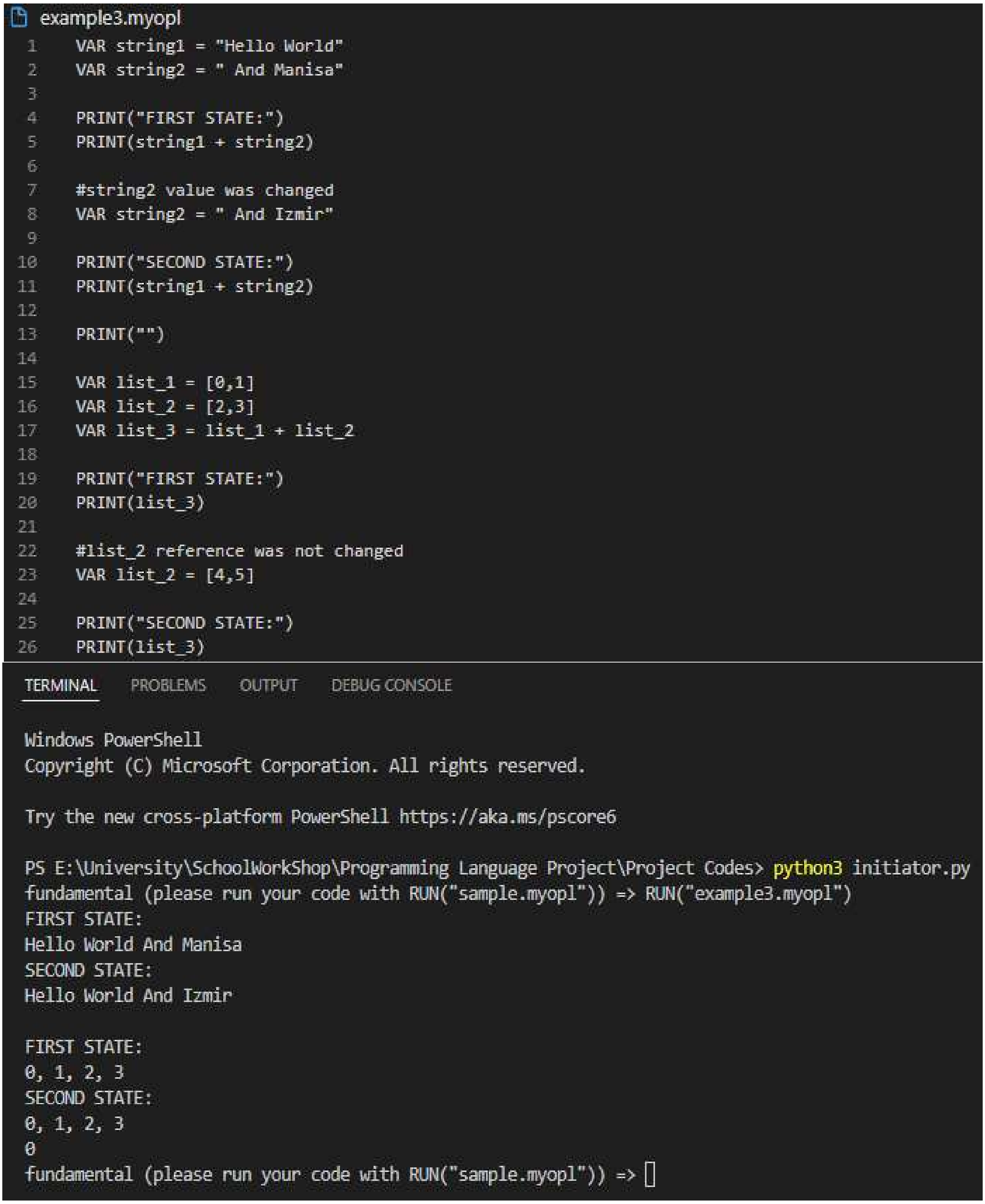
3.2 - Example-2



```
VAR list1 = [1,2,3,4]
VAR list2 = [1,2,3,4,5,6,7,8,9]


FUN make_changes(list)

    IF (LEN(list) <= 7) THEN APPEND(list,2^5)

    IF (LEN(list) > 7) THEN POP(list,(LEN(list)-1))

    RETURN list
END

PRINT("LIST1:")
PRINT(make_changes(list1))

PRINT("LIST2:")
PRINT(make_changes(list2))

```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS E:\University\SchoolWorkShop\Programming Language Project\Project Codes> python3 initiator.py
fundamental (please run your code with RUN("sample.myopl")) => RUN("example2.myopl")
LIST1:
1, 2, 3, 4, 32
LIST2:
1, 2, 3, 4, 5, 6, 7, 8
0
fundamental (please run your code with RUN("sample.myopl")) => 
```

3.3 - Example-3



```
VAR string1 = "Hello World"
VAR string2 = " And Manisa"

PRINT("FIRST STATE:")
PRINT(string1 + string2)

#string2 value was changed
VAR string2 = " And Izmir"

PRINT("SECOND STATE:")
PRINT(string1 + string2)

PRINT("")

VAR list_1 = [0,1]
VAR list_2 = [2,3]
VAR list_3 = list_1 + list_2

PRINT("FIRST STATE:")
PRINT(list_3)

#list_2 reference was not changed
VAR list_2 = [4,5]

PRINT("SECOND STATE:")
PRINT(list_3)
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS E:\University\SchoolWorkShop\Programming Language Project\Project Codes> python3 initiator.py
fundamental (please run your code with RUN("sample.myopl")) => RUN("example3.myopl")
FIRST STATE:
Hello World And Manisa
SECOND STATE:
Hello World And Izmir

FIRST STATE:
0, 1, 2, 3
SECOND STATE:
0, 1, 2, 3
0
fundamental (please run your code with RUN("sample.myopl")) =>
```

I explained my project and programming language in detail. I showed you how to test and finally finished my report with 3 examples.

Thank you for reading this far