**Міністерство Освіти І НАУКИ України**

**Національний університет "Львівська політехніка"**

Інститут **КНІТ**

Кафедра **ПЗ**

### ЗВІТ

До лабораторної роботи № 1

**З дисципліни:** *“Моделювання і аналіз програмного забезпечення”*

**На тему:** *“* *Шаблон інтерпретатор”*

**Лектор:**

к.т.н. доцент.

Сердюк П.В

**Виконав:**

ст. гр. ПЗ-22

Хамар Іван

**Прийняв:**

к.т.н. доцент

Білас О.Є.

« \_\_\_\_ » \_\_\_\_\_\_\_\_ 2020 р.

∑= \_\_\_\_\_ .

Львів – 2020

**Тема:** Шаблон інтерпретатор

**Мета:** Навчитися розробляти, створювати і використовувати   
мову програмування та її інтерпретатор.

**ЗАВДАННЯ**

Розробити імперативну мову, описати її синтаксис. Мова повинна підтримувати лінійне виконання та умовні оператори та цикли (щонайменше один вид циклів). Розробити інтерпретатор/транслятор для неї. Виводити синтаксичне дерево, що будує інтерпретатор з вхідного коду. Оптимізувати дерево.

Варіант 35.

Інтерпретатор формул, який би надавав можливість тримати у собі формули, та дозволяв різні операції з ними. Повинна бути можливість оголошувати змінні, використовувати функції, диференціювання, тощо.

**ХІД ВИКОНАННЯ**

На рис.1 зображено розроблений інтерпретатор, синтаксичне дерево, яке він будує та лінійне виконання команд в ньому.

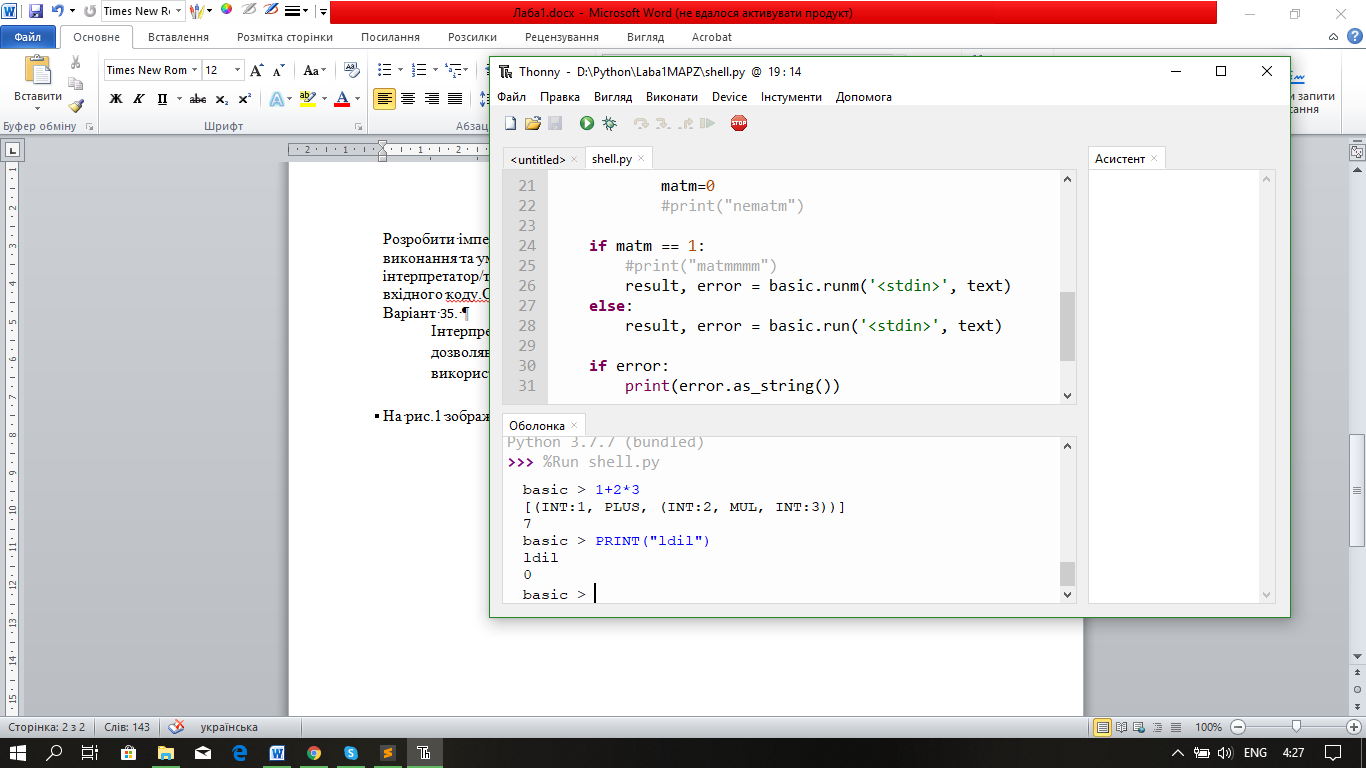


Рис. 1. Інтерпретатор

На рис.2 зображена підтримка змінних і умовного оператора IF THEN.

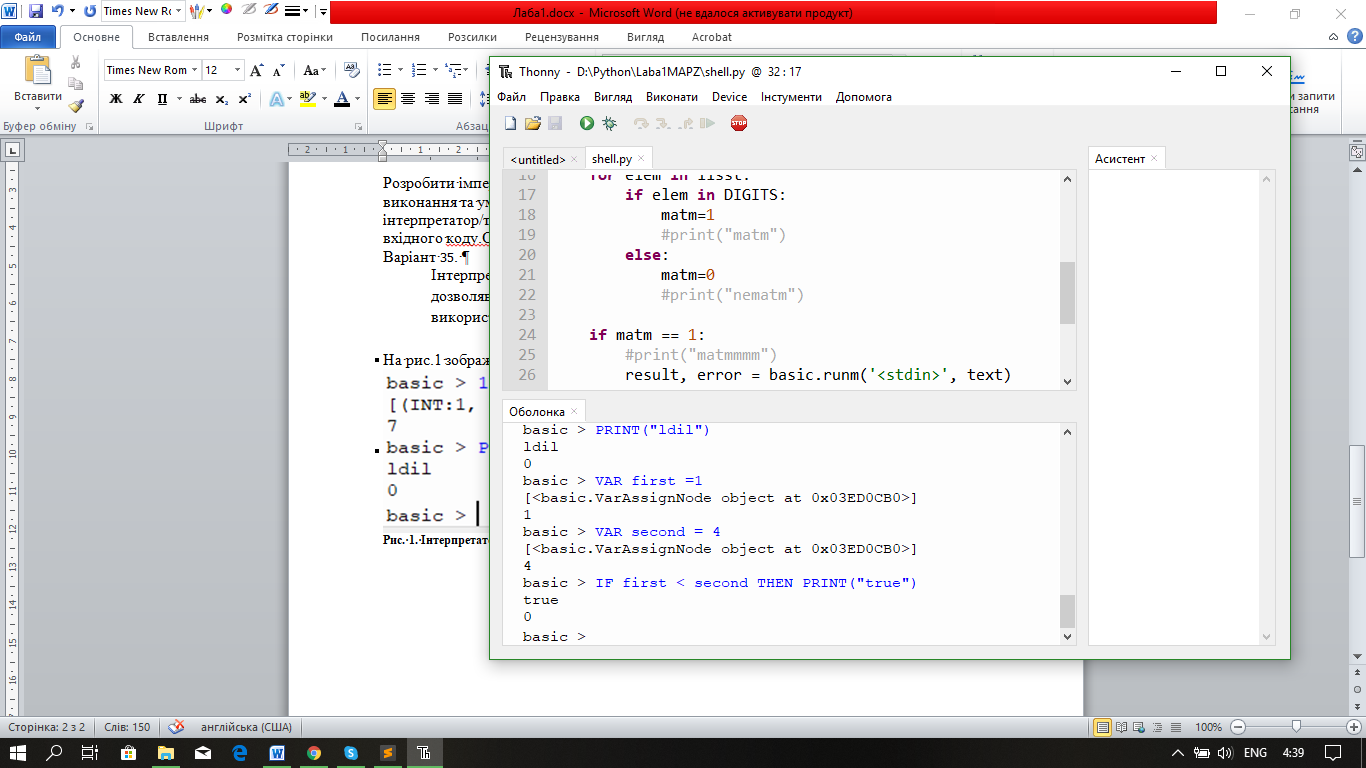


Рис. 2.Змінні і умовний оператор

На рис.3 зображена підтримка циклу FOR THEN.

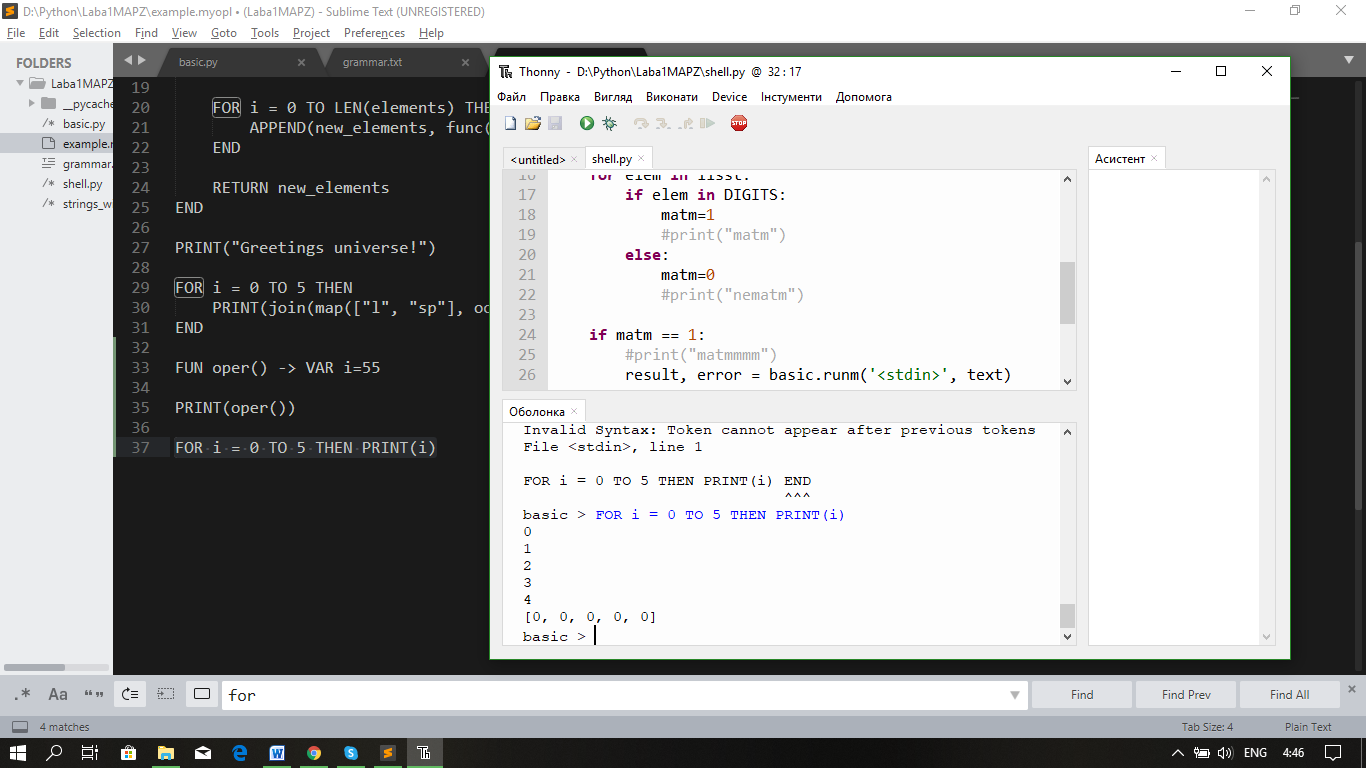


Рис. 3.Цикл FOR

На рис.4 зображена підтримка оголошення і використання функцій.

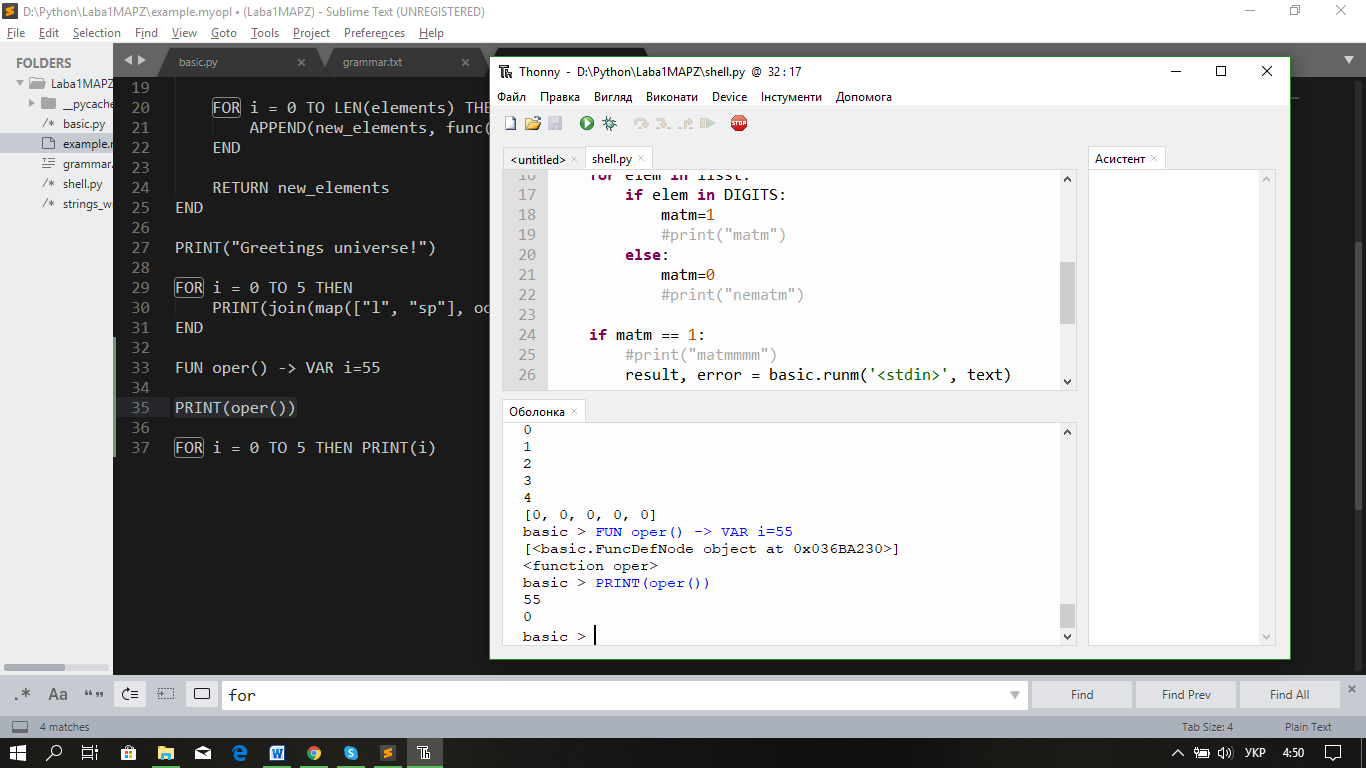


Рис. 4. Функція оголошення змінної

ВИСНОВОК

В результаті виконання цієї лабораторної роботи я навчився розробляти, створювати і використовувати мову програмування та її інтерпретатор

ДОДАТОК

**Shell.py:**

import basic

DIGITS = '0123456789+-\*/^'

matm = 0

def split(word):

return [char for char in word]

while True:

text = input('basic > ')

if text.strip() == "": continue

lisst = split(text)

#print(lisst)

for elem in lisst:

if elem in DIGITS:

matm=1

#print("matm")

else:

matm=0

#print("nematm")

if matm == 1:

#print("matmmmm")

result, error = basic.runm('<stdin>', text)

else:

result, error = basic.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))

**Basic.py:**

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

# IMPORTS

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

from strings\_with\_arrows import \*

import string

import os

import math

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

# CONSTANTS

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

DIGITS = '0123456789'

LETTERS = string.ascii\_letters

LETTERS\_DIGITS = LETTERS + DIGITS

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

# 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)

class RTError(Error):

def \_\_init\_\_(self, pos\_start, pos\_end, details, context):

super().\_\_init\_\_(pos\_start, pos\_end, 'Runtime Error', details)

self.context = context

def as\_string(self):

result = self.generate\_traceback()

result += f'{self.error\_name}: {self.details}'

result += '\n\n' + string\_with\_arrows(self.pos\_start.ftxt, self.pos\_start, self.pos\_end)

return result

def generate\_traceback(self):

result = ''

pos = self.pos\_start

ctx = self.context

while ctx:

result = f' File {pos.fn}, line {str(pos.ln + 1)}, in {ctx.display\_name}\n' + result

pos = ctx.parent\_entry\_pos

ctx = ctx.parent

return 'Traceback (most recent call last):\n' + result

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

# 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)

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

# 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',

]

class Token:

def \_\_init\_\_(self, type\_, value=None, pos\_start=None, pos\_end=None):

self.type = type\_

self.value = value

if pos\_start:

self.pos\_start = pos\_start.copy()

self.pos\_end = pos\_start.copy()

self.pos\_end.advance()

if pos\_end:

self.pos\_end = pos\_end.copy()

def matches(self, type\_, value):

return self.type == type\_ and self.value == value

def \_\_repr\_\_(self):

if self.value: return f'{self.type}:{self.value}'

return f'{self.type}'

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

# 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) else None

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))

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

def make\_number(self):

num\_str = ''

dot\_count = 0

pos\_start = self.pos.copy()

while self.current\_char != None and self.current\_char in DIGITS + '.':

if self.current\_char == '.':

if dot\_count == 1: break

dot\_count += 1

num\_str += self.current\_char

self.advance()

if dot\_count == 0:

return Token(TT\_INT, int(num\_str), pos\_start, self.pos)

else:

return Token(TT\_FLOAT, float(num\_str), pos\_start, self.pos)

def make\_string(self):

string = ''

pos\_start = self.pos.copy()

escape\_character = False

self.advance()

escape\_characters = {

'n': '\n',

't': '\t'

}

while self.current\_char != None and (self.current\_char != '"' or escape\_character):

if escape\_character:

string += escape\_characters.get(self.current\_char, self.current\_char)

else:

if self.current\_char == '\\':

escape\_character = True

else:

string += self.current\_char

self.advance()

escape\_character = False

self.advance()

return Token(TT\_STRING, string, pos\_start, self.pos)

def make\_identifier(self):

id\_str = ''

pos\_start = self.pos.copy()

while self.current\_char != None and self.current\_char in LETTERS\_DIGITS + '\_':

id\_str += self.current\_char

self.advance()

tok\_type = TT\_KEYWORD if id\_str in KEYWORDS else TT\_IDENTIFIER

return Token(tok\_type, id\_str, pos\_start, self.pos)

def make\_minus\_or\_arrow(self):

tok\_type = TT\_MINUS

pos\_start = self.pos.copy()

self.advance()

if self.current\_char == '>':

self.advance()

tok\_type = TT\_ARROW

return Token(tok\_type, pos\_start=pos\_start, pos\_end=self.pos)

def make\_not\_equals(self):

pos\_start = self.pos.copy()

self.advance()

if self.current\_char == '=':

self.advance()

return Token(TT\_NE, pos\_start=pos\_start, pos\_end=self.pos), None

self.advance()

return None, ExpectedCharError(pos\_start, self.pos, "'=' (after '!')")

def make\_equals(self):

tok\_type = TT\_EQ

pos\_start = self.pos.copy()

self.advance()

if self.current\_char == '=':

self.advance()

tok\_type = TT\_EE

return Token(tok\_type, pos\_start=pos\_start, pos\_end=self.pos)

def make\_less\_than(self):

tok\_type = TT\_LT

pos\_start = self.pos.copy()

self.advance()

if self.current\_char == '=':

self.advance()

tok\_type = TT\_LTE

return Token(tok\_type, pos\_start=pos\_start, pos\_end=self.pos)

def make\_greater\_than(self):

tok\_type = TT\_GT

pos\_start = self.pos.copy()

self.advance()

if self.current\_char == '=':

self.advance()

tok\_type = TT\_GTE

return Token(tok\_type, pos\_start=pos\_start, pos\_end=self.pos)

def skip\_comment(self):

self.advance()

while self.current\_char != '\n':

self.advance()

self.advance()

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

# NODES

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

class NumberNode:

def \_\_init\_\_(self, tok):

self.tok = tok

self.pos\_start = self.tok.pos\_start

self.pos\_end = self.tok.pos\_end

def \_\_repr\_\_(self):

return f'{self.tok}'

class StringNode:

def \_\_init\_\_(self, tok):

self.tok = tok

self.pos\_start = self.tok.pos\_start

self.pos\_end = self.tok.pos\_end

def \_\_repr\_\_(self):

return f'{self.tok}'

class ListNode:

def \_\_init\_\_(self, element\_nodes, pos\_start, pos\_end):

self.element\_nodes = element\_nodes

self.pos\_start = pos\_start

self.pos\_end = pos\_end

class VarAccessNode:

def \_\_init\_\_(self, var\_name\_tok):

self.var\_name\_tok = var\_name\_tok

self.pos\_start = self.var\_name\_tok.pos\_start

self.pos\_end = self.var\_name\_tok.pos\_end

class VarAssignNode:

def \_\_init\_\_(self, var\_name\_tok, value\_node):

self.var\_name\_tok = var\_name\_tok

self.value\_node = value\_node

self.pos\_start = self.var\_name\_tok.pos\_start

self.pos\_end = self.value\_node.pos\_end

class BinOpNode:

def \_\_init\_\_(self, left\_node, op\_tok, right\_node):

self.left\_node = left\_node

self.op\_tok = op\_tok

self.right\_node = right\_node

self.pos\_start = self.left\_node.pos\_start

self.pos\_end = self.right\_node.pos\_end

def \_\_repr\_\_(self):

return f'({self.left\_node}, {self.op\_tok}, {self.right\_node})'

class UnaryOpNode:

def \_\_init\_\_(self, op\_tok, node):

self.op\_tok = op\_tok

self.node = node

self.pos\_start = self.op\_tok.pos\_start

self.pos\_end = node.pos\_end

def \_\_repr\_\_(self):

return f'({self.op\_tok}, {self.node})'

class IfNode:

def \_\_init\_\_(self, cases, else\_case):

self.cases = cases

self.else\_case = else\_case

self.pos\_start = self.cases[0][0].pos\_start

self.pos\_end = (self.else\_case or self.cases[len(self.cases) - 1])[0].pos\_end

class ForNode:

def \_\_init\_\_(self, var\_name\_tok, start\_value\_node, end\_value\_node, step\_value\_node, body\_node, should\_return\_null):

self.var\_name\_tok = var\_name\_tok

self.start\_value\_node = start\_value\_node

self.end\_value\_node = end\_value\_node

self.step\_value\_node = step\_value\_node

self.body\_node = body\_node

self.should\_return\_null = should\_return\_null

self.pos\_start = self.var\_name\_tok.pos\_start

self.pos\_end = self.body\_node.pos\_end

class WhileNode:

def \_\_init\_\_(self, condition\_node, body\_node, should\_return\_null):

self.condition\_node = condition\_node

self.body\_node = body\_node

self.should\_return\_null = should\_return\_null

self.pos\_start = self.condition\_node.pos\_start

self.pos\_end = self.body\_node.pos\_end

class FuncDefNode:

def \_\_init\_\_(self, var\_name\_tok, arg\_name\_toks, body\_node, should\_auto\_return):

self.var\_name\_tok = var\_name\_tok

self.arg\_name\_toks = arg\_name\_toks

self.body\_node = body\_node

self.should\_auto\_return = should\_auto\_return

if self.var\_name\_tok:

self.pos\_start = self.var\_name\_tok.pos\_start

elif len(self.arg\_name\_toks) > 0:

self.pos\_start = self.arg\_name\_toks[0].pos\_start

else:

self.pos\_start = self.body\_node.pos\_start

self.pos\_end = self.body\_node.pos\_end

class CallNode:

def \_\_init\_\_(self, node\_to\_call, arg\_nodes):

self.node\_to\_call = node\_to\_call

self.arg\_nodes = arg\_nodes

self.pos\_start = self.node\_to\_call.pos\_start

if len(self.arg\_nodes) > 0:

self.pos\_end = self.arg\_nodes[len(self.arg\_nodes) - 1].pos\_end

else:

self.pos\_end = self.node\_to\_call.pos\_end

class ReturnNode:

def \_\_init\_\_(self, node\_to\_return, pos\_start, pos\_end):

self.node\_to\_return = node\_to\_return

self.pos\_start = pos\_start

self.pos\_end = pos\_end

class ContinueNode:

def \_\_init\_\_(self, pos\_start, pos\_end):

self.pos\_start = pos\_start

self.pos\_end = pos\_end

class BreakNode:

def \_\_init\_\_(self, pos\_start, pos\_end):

self.pos\_start = pos\_start

self.pos\_end = pos\_end

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

# 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

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

# 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)

more\_statements = True

while True:

newline\_count = 0

while self.current\_tok.type == TT\_NEWLINE:

res.register\_advancement()

self.advance()

newline\_count += 1

if newline\_count == 0:

more\_statements = False

if not more\_statements: break

statement = res.try\_register(self.statement())

if not statement:

self.reverse(res.to\_reverse\_count)

more\_statements = False

continue

statements.append(statement)

return res.success(ListNode(

statements,

pos\_start,

self.current\_tok.pos\_end.copy()

))

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, '+', '-', '(', '[' or 'NOT'"

))

return res.success(expr)

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)

def comp\_expr(self):

res = ParseResult()

if self.current\_tok.matches(TT\_KEYWORD, 'NOT'):

op\_tok = self.current\_tok

res.register\_advancement()

self.advance()

node = res.register(self.comp\_expr())

if res.error: return res

return res.success(UnaryOpNode(op\_tok, node))

node = res.register(self.bin\_op(self.arith\_expr, (TT\_EE, TT\_NE, TT\_LT, TT\_GT, TT\_LTE, TT\_GTE)))

if res.error:

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

"Expected int, float, identifier, '+', '-', '(', '[', 'IF', 'FOR', 'WHILE', 'FUN' or 'NOT'"

))

return res.success(node)

def arith\_expr(self):

return self.bin\_op(self.term, (TT\_PLUS, TT\_MINUS))

def term(self):

return self.bin\_op(self.factor, (TT\_MUL, TT\_DIV))

def factor(self):

res = ParseResult()

tok = self.current\_tok

if tok.type in (TT\_PLUS, TT\_MINUS):

res.register\_advancement()

self.advance()

factor = res.register(self.factor())

if res.error: return res

return res.success(UnaryOpNode(tok, factor))

return self.power()

def power(self):

return self.bin\_op(self.call, (TT\_POW, ), self.factor)

def call(self):

res = ParseResult()

atom = res.register(self.atom())

if res.error: return res

if self.current\_tok.type == TT\_LPAREN:

res.register\_advancement()

self.advance()

arg\_nodes = []

if self.current\_tok.type == TT\_RPAREN:

res.register\_advancement()

self.advance()

else:

arg\_nodes.append(res.register(self.expr()))

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'"

))

while self.current\_tok.type == TT\_COMMA:

res.register\_advancement()

self.advance()

arg\_nodes.append(res.register(self.expr()))

if res.error: return res

if self.current\_tok.type != TT\_RPAREN:

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected ',' or ')'"

))

res.register\_advancement()

self.advance()

return res.success(CallNode(atom, arg\_nodes))

return res.success(atom)

def atom(self):

res = ParseResult()

tok = self.current\_tok

if tok.type in (TT\_INT, TT\_FLOAT):

res.register\_advancement()

self.advance()

return res.success(NumberNode(tok))

elif tok.type == TT\_STRING:

res.register\_advancement()

self.advance()

return res.success(StringNode(tok))

elif tok.type == TT\_IDENTIFIER:

res.register\_advancement()

self.advance()

return res.success(VarAccessNode(tok))

elif tok.type == TT\_LPAREN:

res.register\_advancement()

self.advance()

expr = res.register(self.expr())

if res.error: return res

if self.current\_tok.type == TT\_RPAREN:

res.register\_advancement()

self.advance()

return res.success(expr)

else:

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

"Expected ')'"

))

elif tok.type == TT\_LSQUARE:

list\_expr = res.register(self.list\_expr())

if res.error: return res

return res.success(list\_expr)

elif tok.matches(TT\_KEYWORD, 'IF'):

if\_expr = res.register(self.if\_expr())

if res.error: return res

return res.success(if\_expr)

elif tok.matches(TT\_KEYWORD, 'FOR'):

for\_expr = res.register(self.for\_expr())

if res.error: return res

return res.success(for\_expr)

elif tok.matches(TT\_KEYWORD, 'WHILE'):

while\_expr = res.register(self.while\_expr())

if res.error: return res

return res.success(while\_expr)

elif tok.matches(TT\_KEYWORD, 'FUN'):

func\_def = res.register(self.func\_def())

if res.error: return res

return res.success(func\_def)

return res.failure(InvalidSyntaxError(

tok.pos\_start, tok.pos\_end,

"Expected int, float, identifier, '+', '-', '(', '[', IF', 'FOR', 'WHILE', 'FUN'"

))

def list\_expr(self):

res = ParseResult()

element\_nodes = []

pos\_start = self.current\_tok.pos\_start.copy()

if self.current\_tok.type != TT\_LSQUARE:

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected '['"

))

res.register\_advancement()

self.advance()

if self.current\_tok.type == TT\_RSQUARE:

res.register\_advancement()

self.advance()

else:

element\_nodes.append(res.register(self.expr()))

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'"

))

while self.current\_tok.type == TT\_COMMA:

res.register\_advancement()

self.advance()

element\_nodes.append(res.register(self.expr()))

if res.error: return res

if self.current\_tok.type != TT\_RSQUARE:

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected ',' or ']'"

))

res.register\_advancement()

self.advance()

return res.success(ListNode(

element\_nodes,

pos\_start,

self.current\_tok.pos\_end.copy()

))

def if\_expr(self):

res = ParseResult()

all\_cases = res.register(self.if\_expr\_cases('IF'))

if res.error: return res

cases, else\_case = all\_cases

return res.success(IfNode(cases, else\_case))

def if\_expr\_b(self):

return self.if\_expr\_cases('ELIF')

def if\_expr\_c(self):

res = ParseResult()

else\_case = None

if self.current\_tok.matches(TT\_KEYWORD, 'ELSE'):

res.register\_advancement()

self.advance()

if self.current\_tok.type == TT\_NEWLINE:

res.register\_advancement()

self.advance()

statements = res.register(self.statements())

if res.error: return res

else\_case = (statements, True)

if self.current\_tok.matches(TT\_KEYWORD, 'END'):

res.register\_advancement()

self.advance()

else:

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

"Expected 'END'"

))

else:

expr = res.register(self.statement())

if res.error: return res

else\_case = (expr, False)

return res.success(else\_case)

def if\_expr\_b\_or\_c(self):

res = ParseResult()

cases, else\_case = [], None

if self.current\_tok.matches(TT\_KEYWORD, 'ELIF'):

all\_cases = res.register(self.if\_expr\_b())

if res.error: return res

cases, else\_case = all\_cases

else:

else\_case = res.register(self.if\_expr\_c())

if res.error: return res

return res.success((cases, else\_case))

def if\_expr\_cases(self, case\_keyword):

res = ParseResult()

cases = []

else\_case = None

if not self.current\_tok.matches(TT\_KEYWORD, case\_keyword):

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected '{case\_keyword}'"

))

res.register\_advancement()

self.advance()

condition = res.register(self.expr())

if res.error: return res

if not self.current\_tok.matches(TT\_KEYWORD, 'THEN'):

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected 'THEN'"

))

res.register\_advancement()

self.advance()

if self.current\_tok.type == TT\_NEWLINE:

res.register\_advancement()

self.advance()

statements = res.register(self.statements())

if res.error: return res

cases.append((condition, statements, True))

if self.current\_tok.matches(TT\_KEYWORD, 'END'):

res.register\_advancement()

self.advance()

else:

all\_cases = res.register(self.if\_expr\_b\_or\_c())

if res.error: return res

new\_cases, else\_case = all\_cases

cases.extend(new\_cases)

else:

expr = res.register(self.statement())

if res.error: return res

cases.append((condition, expr, False))

all\_cases = res.register(self.if\_expr\_b\_or\_c())

if res.error: return res

new\_cases, else\_case = all\_cases

cases.extend(new\_cases)

return res.success((cases, else\_case))

def for\_expr(self):

res = ParseResult()

if not self.current\_tok.matches(TT\_KEYWORD, 'FOR'):

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected 'FOR'"

))

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,

f"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,

f"Expected '='"

))

res.register\_advancement()

self.advance()

start\_value = res.register(self.expr())

if res.error: return res

if not self.current\_tok.matches(TT\_KEYWORD, 'TO'):

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected 'TO'"

))

res.register\_advancement()

self.advance()

end\_value = res.register(self.expr())

if res.error: return res

if self.current\_tok.matches(TT\_KEYWORD, 'STEP'):

res.register\_advancement()

self.advance()

step\_value = res.register(self.expr())

if res.error: return res

else:

step\_value = None

if not self.current\_tok.matches(TT\_KEYWORD, 'THEN'):

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected 'THEN'"

))

res.register\_advancement()

self.advance()

if self.current\_tok.type == TT\_NEWLINE:

res.register\_advancement()

self.advance()

body = res.register(self.statements())

if res.error: return res

if not self.current\_tok.matches(TT\_KEYWORD, 'END'):

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected 'END'"

))

res.register\_advancement()

self.advance()

return res.success(ForNode(var\_name, start\_value, end\_value, step\_value, body, True))

body = res.register(self.statement())

if res.error: return res

return res.success(ForNode(var\_name, start\_value, end\_value, step\_value, body, False))

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'"

))

res.register\_advancement()

self.advance()

condition = res.register(self.expr())

if res.error: return res

if not self.current\_tok.matches(TT\_KEYWORD, 'THEN'):

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected 'THEN'"

))

res.register\_advancement()

self.advance()

if self.current\_tok.type == TT\_NEWLINE:

res.register\_advancement()

self.advance()

body = res.register(self.statements())

if res.error: return res

if not self.current\_tok.matches(TT\_KEYWORD, 'END'):

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected 'END'"

))

res.register\_advancement()

self.advance()

return res.success(WhileNode(condition, body, True))

body = res.register(self.statement())

if res.error: return res

return res.success(WhileNode(condition, body, False))

def func\_def(self):

res = ParseResult()

if not self.current\_tok.matches(TT\_KEYWORD, 'FUN'):

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected 'FUN'"

))

res.register\_advancement()

self.advance()

if self.current\_tok.type == TT\_IDENTIFIER:

var\_name\_tok = self.current\_tok

res.register\_advancement()

self.advance()

if self.current\_tok.type != TT\_LPAREN:

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected '('"

))

else:

var\_name\_tok = None

if self.current\_tok.type != TT\_LPAREN:

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected identifier or '('"

))

res.register\_advancement()

self.advance()

arg\_name\_toks = []

if self.current\_tok.type == TT\_IDENTIFIER:

arg\_name\_toks.append(self.current\_tok)

res.register\_advancement()

self.advance()

while self.current\_tok.type == TT\_COMMA:

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,

f"Expected identifier"

))

arg\_name\_toks.append(self.current\_tok)

res.register\_advancement()

self.advance()

if self.current\_tok.type != TT\_RPAREN:

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected ',' or ')'"

))

else:

if self.current\_tok.type != TT\_RPAREN:

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected identifier or ')'"

))

res.register\_advancement()

self.advance()

if self.current\_tok.type == TT\_ARROW:

res.register\_advancement()

self.advance()

body = res.register(self.expr())

if res.error: return res

return res.success(FuncDefNode(

var\_name\_tok,

arg\_name\_toks,

body,

True

))

if self.current\_tok.type != TT\_NEWLINE:

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected '->' or NEWLINE"

))

res.register\_advancement()

self.advance()

body = res.register(self.statements())

if res.error: return res

if not self.current\_tok.matches(TT\_KEYWORD, 'END'):

return res.failure(InvalidSyntaxError(

self.current\_tok.pos\_start, self.current\_tok.pos\_end,

f"Expected 'END'"

))

res.register\_advancement()

self.advance()

return res.success(FuncDefNode(

var\_name\_tok,

arg\_name\_toks,

body,

False

))

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

def bin\_op(self, func\_a, ops, func\_b=None):

if func\_b == None:

func\_b = func\_a

res = ParseResult()

left = res.register(func\_a())

if res.error: return res

while self.current\_tok.type in ops or (self.current\_tok.type, self.current\_tok.value) in ops:

op\_tok = self.current\_tok

res.register\_advancement()

self.advance()

right = res.register(func\_b())

if res.error: return res

left = BinOpNode(left, op\_tok, right)

return res.success(left)

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

# 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

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

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)

def powed\_by(self, other):

return None, self.illegal\_operation(other)

def get\_comparison\_eq(self, other):

return None, self.illegal\_operation(other)

def get\_comparison\_ne(self, other):

return None, self.illegal\_operation(other)

def get\_comparison\_lt(self, other):

return None, self.illegal\_operation(other)

def get\_comparison\_gt(self, other):

return None, self.illegal\_operation(other)

def get\_comparison\_lte(self, other):

return None, self.illegal\_operation(other)

def get\_comparison\_gte(self, other):

return None, self.illegal\_operation(other)

def anded\_by(self, other):

return None, self.illegal\_operation(other)

def ored\_by(self, other):

return None, self.illegal\_operation(other)

def notted(self):

return None, self.illegal\_operation(other)

def execute(self, args):

return RTResult().failure(self.illegal\_operation())

def copy(self):

raise Exception('No copy method defined')

def is\_true(self):

return False

def illegal\_operation(self, other=None):

if not other: other = self

return RTError(

self.pos\_start, other.pos\_end,

'Illegal operation',

self.context

)

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)

def powed\_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 get\_comparison\_eq(self, other):

if isinstance(other, Number):

return Number(int(self.value == other.value)).set\_context(self.context), None

else:

return None, Value.illegal\_operation(self, other)

def get\_comparison\_ne(self, other):

if isinstance(other, Number):

return Number(int(self.value != other.value)).set\_context(self.context), None

else:

return None, Value.illegal\_operation(self, other)

def get\_comparison\_lt(self, other):

if isinstance(other, Number):

return Number(int(self.value < other.value)).set\_context(self.context), None

else:

return None, Value.illegal\_operation(self, other)

def get\_comparison\_gt(self, other):

if isinstance(other, Number):

return Number(int(self.value > other.value)).set\_context(self.context), None

else:

return None, Value.illegal\_operation(self, other)

def get\_comparison\_lte(self, other):

if isinstance(other, Number):

return Number(int(self.value <= other.value)).set\_context(self.context), None

else:

return None, Value.illegal\_operation(self, other)

def get\_comparison\_gte(self, other):

if isinstance(other, Number):

return Number(int(self.value >= other.value)).set\_context(self.context), None

else:

return None, Value.illegal\_operation(self, other)

def anded\_by(self, other):

if isinstance(other, Number):

return Number(int(self.value and other.value)).set\_context(self.context), None

else:

return None, Value.illegal\_operation(self, other)

def ored\_by(self, other):

if isinstance(other, Number):

return Number(int(self.value or other.value)).set\_context(self.context), None

else:

return None, Value.illegal\_operation(self, other)

def notted(self):

return Number(1 if self.value == 0 else 0).set\_context(self.context), None

def copy(self):

copy = Number(self.value)

copy.set\_pos(self.pos\_start, self.pos\_end)

copy.set\_context(self.context)

return copy

def is\_true(self):

return self.value != 0

def \_\_str\_\_(self):

return str(self.value)

def \_\_repr\_\_(self):

return str(self.value)

Number.null = Number(0)

Number.false = Number(0)

Number.true = Number(1)

Number.math\_PI = Number(math.pi)

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 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,

'Element at this index could not be retrieved from list because index is out of bounds',

self.context

)

else:

return None, Value.illegal\_operation(self, other)

def copy(self):

copy = List(self.elements)

copy.set\_pos(self.pos\_start, self.pos\_end)

copy.set\_context(self.context)

return copy

def \_\_str\_\_(self):

return ", ".join([str(x) for x in self.elements])

def \_\_repr\_\_(self):

return f'[{", ".join([repr(x) for x in self.elements])}]'

class BaseFunction(Value):

def \_\_init\_\_(self, name):

super().\_\_init\_\_()

self.name = name or "<anonymous>"

def generate\_new\_context(self):

new\_context = Context(self.name, self.context, self.pos\_start)

new\_context.symbol\_table = SymbolTable(new\_context.parent.symbol\_table)

return new\_context

def check\_args(self, arg\_names, args):

res = RTResult()

if len(args) > len(arg\_names):

return res.failure(RTError(

self.pos\_start, self.pos\_end,

f"{len(args) - len(arg\_names)} too many args passed into {self}",

self.context

))

if len(args) < len(arg\_names):

return res.failure(RTError(

self.pos\_start, self.pos\_end,

f"{len(arg\_names) - len(args)} too few args passed into {self}",

self.context

))

return res.success(None)

def populate\_args(self, arg\_names, args, exec\_ctx):

for i in range(len(args)):

arg\_name = arg\_names[i]

arg\_value = args[i]

arg\_value.set\_context(exec\_ctx)

exec\_ctx.symbol\_table.set(arg\_name, arg\_value)

def check\_and\_populate\_args(self, arg\_names, args, exec\_ctx):

res = RTResult()

res.register(self.check\_args(arg\_names, args))

if res.should\_return(): return res

self.populate\_args(arg\_names, args, exec\_ctx)

return res.success(None)

class Function(BaseFunction):

def \_\_init\_\_(self, name, body\_node, arg\_names, should\_auto\_return):

super().\_\_init\_\_(name)

self.body\_node = body\_node

self.arg\_names = arg\_names

self.should\_auto\_return = should\_auto\_return

def execute(self, args):

res = RTResult()

interpreter = Interpreter()

exec\_ctx = self.generate\_new\_context()

res.register(self.check\_and\_populate\_args(self.arg\_names, args, exec\_ctx))

if res.should\_return(): return res

value = res.register(interpreter.visit(self.body\_node, exec\_ctx))

if res.should\_return() and res.func\_return\_value == None: return res

ret\_value = (value if self.should\_auto\_return else None) or res.func\_return\_value or Number.null

return res.success(ret\_value)

def copy(self):

copy = Function(self.name, self.body\_node, self.arg\_names, self.should\_auto\_return)

copy.set\_context(self.context)

copy.set\_pos(self.pos\_start, self.pos\_end)

return copy

def \_\_repr\_\_(self):

return f"<function {self.name}>"

class BuiltInFunction(BaseFunction):

def \_\_init\_\_(self, name):

super().\_\_init\_\_(name)

def execute(self, args):

res = RTResult()

exec\_ctx = self.generate\_new\_context()

method\_name = f'execute\_{self.name}'

method = getattr(self, method\_name, self.no\_visit\_method)

res.register(self.check\_and\_populate\_args(method.arg\_names, args, exec\_ctx))

if res.should\_return(): return res

return\_value = res.register(method(exec\_ctx))

if res.should\_return(): return res

return res.success(return\_value)

def no\_visit\_method(self, node, context):

raise Exception(f'No execute\_{self.name} method defined')

def copy(self):

copy = BuiltInFunction(self.name)

copy.set\_context(self.context)

copy.set\_pos(self.pos\_start, self.pos\_end)

return copy

def \_\_repr\_\_(self):

return f"<built-in function {self.name}>"

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

def execute\_print(self, exec\_ctx):

print(str(exec\_ctx.symbol\_table.get('value')))

return RTResult().success(Number.null)

execute\_print.arg\_names = ['value']

def execute\_print\_ret(self, exec\_ctx):

return RTResult().success(String(str(exec\_ctx.symbol\_table.get('value'))))

execute\_print\_ret.arg\_names = ['value']

def execute\_input(self, exec\_ctx):

text = input()

return RTResult().success(String(text))

execute\_input.arg\_names = []

def execute\_input\_int(self, exec\_ctx):

while True:

text = input()

try:

number = int(text)

break

except ValueError:

print(f"'{text}' must be an integer. Try again!")

return RTResult().success(Number(number))

execute\_input\_int.arg\_names = []

def execute\_clear(self, exec\_ctx):

os.system('cls' if os.name == 'nt' else 'cls')

return RTResult().success(Number.null)

execute\_clear.arg\_names = []

def execute\_is\_number(self, exec\_ctx):

is\_number = isinstance(exec\_ctx.symbol\_table.get("value"), Number)

return RTResult().success(Number.true if is\_number else Number.false)

execute\_is\_number.arg\_names = ["value"]

def execute\_is\_string(self, exec\_ctx):

is\_number = isinstance(exec\_ctx.symbol\_table.get("value"), String)

return RTResult().success(Number.true if is\_number else Number.false)

execute\_is\_string.arg\_names = ["value"]

def execute\_is\_list(self, exec\_ctx):

is\_number = isinstance(exec\_ctx.symbol\_table.get("value"), List)

return RTResult().success(Number.true if is\_number else Number.false)

execute\_is\_list.arg\_names = ["value"]

def execute\_is\_function(self, exec\_ctx):

is\_number = isinstance(exec\_ctx.symbol\_table.get("value"), BaseFunction)

return RTResult().success(Number.true if is\_number else Number.false)

execute\_is\_function.arg\_names = ["value"]

def execute\_append(self, exec\_ctx):

list\_ = exec\_ctx.symbol\_table.get("list")

value = exec\_ctx.symbol\_table.get("value")

if not isinstance(list\_, List):

return RTResult().failure(RTError(

self.pos\_start, self.pos\_end,

"First argument must be list",

exec\_ctx

))

list\_.elements.append(value)

return RTResult().success(Number.null)

execute\_append.arg\_names = ["list", "value"]

def execute\_pop(self, exec\_ctx):

list\_ = exec\_ctx.symbol\_table.get("list")

index = exec\_ctx.symbol\_table.get("index")

if not isinstance(list\_, List):

return RTResult().failure(RTError(

self.pos\_start, self.pos\_end,

"First argument must be list",

exec\_ctx

))

if not isinstance(index, Number):

return RTResult().failure(RTError(

self.pos\_start, self.pos\_end,

"Second argument must be number",

exec\_ctx

))

try:

element = list\_.elements.pop(index.value)

except:

return RTResult().failure(RTError(

self.pos\_start, self.pos\_end,

'Element at this index could not be removed from list because index is out of bounds',

exec\_ctx

))

return RTResult().success(element)

execute\_pop.arg\_names = ["list", "index"]

def execute\_extend(self, exec\_ctx):

listA = exec\_ctx.symbol\_table.get("listA")

listB = exec\_ctx.symbol\_table.get("listB")

if not isinstance(listA, List):

return RTResult().failure(RTError(

self.pos\_start, self.pos\_end,

"First argument must be list",

exec\_ctx

))

if not isinstance(listB, List):

return RTResult().failure(RTError(

self.pos\_start, self.pos\_end,

"Second argument must be list",

exec\_ctx

))

listA.elements.extend(listB.elements)

return RTResult().success(Number.null)

execute\_extend.arg\_names = ["listA", "listB"]

def execute\_len(self, exec\_ctx):

list\_ = exec\_ctx.symbol\_table.get("list")

if not isinstance(list\_, List):

return RTResult().failure(RTError(

self.pos\_start, self.pos\_end,

"Argument must be list",

exec\_ctx

))

return RTResult().success(Number(len(list\_.elements)))

execute\_len.arg\_names = ["list"]

def execute\_run(self, exec\_ctx):

fn = exec\_ctx.symbol\_table.get("fn")

if not isinstance(fn, String):

return RTResult().failure(RTError(

self.pos\_start, self.pos\_end,

"Second argument must be string",

exec\_ctx

))

fn = fn.value

try:

with open(fn, "r") as f:

script = f.read()

except Exception as e:

return RTResult().failure(RTError(

self.pos\_start, self.pos\_end,

f"Failed to load script \"{fn}\"\n" + str(e),

exec\_ctx

))

\_, error = run(fn, script)

if error:

return RTResult().failure(RTError(

self.pos\_start, self.pos\_end,

f"Failed to finish executing script \"{fn}\"\n" +

error.as\_string(),

exec\_ctx

))

return RTResult().success(Number.null)

execute\_run.arg\_names = ["fn"]

BuiltInFunction.print = BuiltInFunction("print")

BuiltInFunction.print\_ret = BuiltInFunction("print\_ret")

BuiltInFunction.input = BuiltInFunction("input")

BuiltInFunction.input\_int = BuiltInFunction("input\_int")

BuiltInFunction.clear = BuiltInFunction("clear")

BuiltInFunction.is\_number = BuiltInFunction("is\_number")

BuiltInFunction.is\_string = BuiltInFunction("is\_string")

BuiltInFunction.is\_list = BuiltInFunction("is\_list")

BuiltInFunction.is\_function = BuiltInFunction("is\_function")

BuiltInFunction.append = BuiltInFunction("append")

BuiltInFunction.pop = BuiltInFunction("pop")

BuiltInFunction.extend = BuiltInFunction("extend")

BuiltInFunction.len = BuiltInFunction("len")

BuiltInFunction.run = BuiltInFunction("run")

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

# CONTEXT

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

class Context:

def \_\_init\_\_(self, display\_name, parent=None, parent\_entry\_pos=None):

self.display\_name = display\_name

self.parent = parent

self.parent\_entry\_pos = parent\_entry\_pos

self.symbol\_table = None

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

# 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

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

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()

var\_name = node.var\_name\_tok.value

value = context.symbol\_table.get(var\_name)

if not value:

return res.failure(RTError(

node.pos\_start, node.pos\_end,

f"'{var\_name}' is not defined",

context

))

value = value.copy().set\_pos(node.pos\_start, node.pos\_end).set\_context(context)

return res.success(value)

def visit\_VarAssignNode(self, node, context):

res = RTResult()

var\_name = node.var\_name\_tok.value

value = res.register(self.visit(node.value\_node, context))

if res.should\_return(): return res

context.symbol\_table.set(var\_name, value)

return res.success(value)

def visit\_BinOpNode(self, node, context):

res = RTResult()

left = res.register(self.visit(node.left\_node, context))

if res.should\_return(): return res

right = res.register(self.visit(node.right\_node, context))

if res.should\_return(): return res

if node.op\_tok.type == TT\_PLUS:

result, error = left.added\_to(right)

elif node.op\_tok.type == TT\_MINUS:

result, error = left.subbed\_by(right)

elif node.op\_tok.type == TT\_MUL:

result, error = left.multed\_by(right)

elif node.op\_tok.type == TT\_DIV:

result, error = left.dived\_by(right)

elif node.op\_tok.type == TT\_POW:

result, error = left.powed\_by(right)

elif node.op\_tok.type == TT\_EE:

result, error = left.get\_comparison\_eq(right)

elif node.op\_tok.type == TT\_NE:

result, error = left.get\_comparison\_ne(right)

elif node.op\_tok.type == TT\_LT:

result, error = left.get\_comparison\_lt(right)

elif node.op\_tok.type == TT\_GT:

result, error = left.get\_comparison\_gt(right)

elif node.op\_tok.type == TT\_LTE:

result, error = left.get\_comparison\_lte(right)

elif node.op\_tok.type == TT\_GTE:

result, error = left.get\_comparison\_gte(right)

elif node.op\_tok.matches(TT\_KEYWORD, 'AND'):

result, error = left.anded\_by(right)

elif node.op\_tok.matches(TT\_KEYWORD, 'OR'):

result, error = left.ored\_by(right)

if error:

return res.failure(error)

else:

return res.success(result.set\_pos(node.pos\_start, node.pos\_end))

def visit\_UnaryOpNode(self, node, context):

res = RTResult()

number = res.register(self.visit(node.node, context))

if res.should\_return(): return res

error = None

if node.op\_tok.type == TT\_MINUS:

number, error = number.multed\_by(Number(-1))

elif node.op\_tok.matches(TT\_KEYWORD, 'NOT'):

number, error = number.notted()

if error:

return res.failure(error)

else:

return res.success(number.set\_pos(node.pos\_start, node.pos\_end))

def visit\_IfNode(self, node, context):

res = RTResult()

for condition, expr, should\_return\_null in node.cases:

condition\_value = res.register(self.visit(condition, context))

if res.should\_return(): return res

if condition\_value.is\_true():

expr\_value = res.register(self.visit(expr, context))

if res.should\_return(): return res

return res.success(Number.null if should\_return\_null else expr\_value)

if node.else\_case:

expr, should\_return\_null = node.else\_case

expr\_value = res.register(self.visit(expr, context))

if res.should\_return(): return res

return res.success(Number.null if should\_return\_null else expr\_value)

return res.success(Number.null)

def visit\_ForNode(self, node, context):

res = RTResult()

elements = []

start\_value = res.register(self.visit(node.start\_value\_node, context))

if res.should\_return(): return res

end\_value = res.register(self.visit(node.end\_value\_node, context))

if res.should\_return(): return res

if node.step\_value\_node:

step\_value = res.register(self.visit(node.step\_value\_node, context))

if res.should\_return(): return res

else:

step\_value = Number(1)

i = start\_value.value

if step\_value.value >= 0:

condition = lambda: i < end\_value.value

else:

condition = lambda: i > end\_value.value

while condition():

context.symbol\_table.set(node.var\_name\_tok.value, Number(i))

i += step\_value.value

value = res.register(self.visit(node.body\_node, context))

if res.should\_return() and res.loop\_should\_continue == False and res.loop\_should\_break == False: return res

if res.loop\_should\_continue:

continue

if res.loop\_should\_break:

break

elements.append(value)

return res.success(

Number.null if node.should\_return\_null else

List(elements).set\_context(context).set\_pos(node.pos\_start, node.pos\_end)

)

def visit\_WhileNode(self, node, context):

res = RTResult()

elements = []

while True:

condition = res.register(self.visit(node.condition\_node, context))

if res.should\_return(): return res

if not condition.is\_true():

break

value = res.register(self.visit(node.body\_node, context))

if res.should\_return() and res.loop\_should\_continue == False and res.loop\_should\_break == False: return res

if res.loop\_should\_continue:

continue

if res.loop\_should\_break:

break

elements.append(value)

return res.success(

Number.null if node.should\_return\_null else

List(elements).set\_context(context).set\_pos(node.pos\_start, node.pos\_end)

)

def visit\_FuncDefNode(self, node, context):

res = RTResult()

func\_name = node.var\_name\_tok.value if node.var\_name\_tok else None

body\_node = node.body\_node

arg\_names = [arg\_name.value for arg\_name in node.arg\_name\_toks]

func\_value = Function(func\_name, body\_node, arg\_names, node.should\_auto\_return).set\_context(context).set\_pos(node.pos\_start, node.pos\_end)

if node.var\_name\_tok:

context.symbol\_table.set(func\_name, func\_value)

return res.success(func\_value)

def visit\_CallNode(self, node, context):

res = RTResult()

args = []

value\_to\_call = res.register(self.visit(node.node\_to\_call, context))

if res.should\_return(): return res

value\_to\_call = value\_to\_call.copy().set\_pos(node.pos\_start, node.pos\_end)

for arg\_node in node.arg\_nodes:

args.append(res.register(self.visit(arg\_node, context)))

if res.should\_return(): return res

return\_value = res.register(value\_to\_call.execute(args))

if res.should\_return(): return res

return\_value = return\_value.copy().set\_pos(node.pos\_start, node.pos\_end).set\_context(context)

return res.success(return\_value)

def visit\_ReturnNode(self, node, context):

res = RTResult()

if node.node\_to\_return:

value = res.register(self.visit(node.node\_to\_return, context))

if res.should\_return(): return res

else:

value = Number.null

return res.success\_return(value)

def visit\_ContinueNode(self, node, context):

return RTResult().success\_continue()

def visit\_BreakNode(self, node, context):

return RTResult().success\_break()

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

# 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()

#print(ast.node.element\_nodes)

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

def runm(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()

print(ast.node.element\_nodes)

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)