

GROUP 4

KYALO MAURICE MUNYASYA - P15/1642/2019

BERNARDS YVETTE NOAH - P15/136174/2019

DANIEL WANYORO MUIGAI - P15/1648/2019

MUSEE ALLAN MEISILAL - P15/137833/2019

MATIVO MARTIN MUTUNE - P15/137915/2019

GRAMMAR

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)

```

SCANNER SOURCE FILE

```

#####
# 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)

    def __repr__(self):
        return f'{self.details}'

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

    def __repr__(self):
        return f'{self.details}'

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

    def __repr__(self):
        return f'{self.details}'

class RuntimeError(Error):
    def __init__(self, pos_start, pos_end, details, context):
        super().__init__(pos_start, pos_end, 'Runtime Error', details)
        self.context = context

    def __repr__(self):
        return f'{self.details}'

    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()

def execute_run(fn):

    with open(fn, "r") as f:
        script = f.read()

    tokens, error = run(fn, script)

    return tokens, error

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

    return tokens, error

```

RUN SCANNER SOURCE FILE

```

#-----
#     MAIN SCANNER
#-----

#####
#     IMPORTS
#####

import basic_scanner

while True:
    text = input('basic scanner> ')
    if text.strip() == "": continue

    if 'RUN' in text:
        fn = text.strip("(\\"")RUN")
        result, error = basic_scanner.execute_run(fn)
    else:
        result, error = basic_scanner.run('<stdin>', text)

```

```

if error:
    print(error.as_string())
elif result:
    print(repr(result))

```

```

dmore@dmore: ~/Year-3.2/326_compiler_construction
dmore@dmore:~/Year-3.2/326_compiler_construction$ python3 main_scanner.py
basic scanner> RUN("test")
[NEWLINE, KEYWORD:FUN, IDENTIFIER:oopify, LPAREN, IDENTIFIER:prefix, RPAREN, ARROW, IDENTIFIER:prefix, PLUS, STRING:oop, NEWLINE, NEWLINE, KEYWORD:FUN, IDENTIFIER:join, LPAREN, IDENTIFIER:elements, COMMA, IDENTIFIER:separator, RPAREN, NEWLINE, KEYWORD:VAR, IDENTIFIER:result, EQ, STRING:..., EQ, INT, KEYWORD:TO, IDENTIFIER:len, KEYWORD:THEN, NEWLINE, KEYWORD:VAR, IDENTIFIER:result, EQ, IDENTIFIER:result, PLUS, IDENTIFIER:elements, DIV, IDENTIFIER:i, NEWLINE, KEYWORD:IF, IDENTIFIER:i, NE, IDENTIFIER:len, MINUS, INT:1, KEYWORD:THEN, KEYWORD:VAR, IDENTIFIER:result, EQ, IDENTIFIER:result, PLUS, IDENTIFIER:separator, NEWLINE, KEYWORD:END, NEWLINE, NEWLINE, KEYWORD:RETURN, IDENTIFIER:result, NEWLINE, KEYWORD:END, NEWLINE, NEWLINE, KEYWORD:FUN, IDENTIFIER:map, LPAREN, IDENTIFIER:elements, COMMA, IDENTIFIER:func, RPAREN, NEWLINE, KEYWORD:VAR, IDENTIFIER:new_elements, EQ, LSQUARE, RSQUARE, NEWLINE, NEWLINE, KEYWORD:FOR, IDENTIFIER:i, EQ, INT, KEYWORD:TO, IDENTIFIER:LEN, LPAREN, IDENTIFIER:elements, RPAREN, KEYWORD:THEN, NEWLINE, IDENTIFIER:APPEND, LPAREN, IDENTIFIER:new_elements, COMMA, IDENTIFIER:func, LPAREN, IDENTIFIER:elements, DIV, IDENTIFIER:i, RPAREN, RPAREN, NEWLINE, KEYWORD:END, NEWLINE, NEWLINE, KEYWORD:RETURN, IDENTIFIER:new_elements, NEWLINE, KEYWORD:END, NEWLINE, NEWLINE, IDENTIFIER:PRINT, LPAREN, STRING:Greetings universe!, RPAREN, NEWLINE, NEWLINE, KEYWORD:FOR, IDENTIFIER:i, EQ, INT, KEYWORD:TO, INT:5, KEYWORD:THEN, NEWLINE, IDENTIFIER:PRINT, LPAREN, IDENTIFIER:join, LPAREN, IDENTIFIER:map, LPAREN, LSQUARE, STRING:l, COMMA, STRING:sp, RSQUARE, COMMA, IDENTIFIER:oopify, RPAREN, COMMA, STRING:, , RPAREN, RPAREN, NEWLINE, KEYWORD:END, NEWLINE, EOF]
basic scanner>

```

:SCANNER OUTPUT ON SAMPLE CODE

PARSING STRATEGY DESCRIPTION

An LL parser (Left-to-right, leftmost derivation) is a top-down parser for a restricted context-free language. It parses the input from Left to right, performing Leftmost derivation of the sentence. An LL parser is called an LL(k) parser if it uses k tokens of lookahead when parsing a sentence.

The first L indicates that the input is read from left to right.

The second L says that it produces a left-to-right derivation.

And the 1 says that it uses one lookahead token

The parser needs to find a production to use for nonterminal N when it sees lookahead token t.

To select which production to use, it suffices to have a table that has, as a key, a pair (N, t) and gives the number of productions to use.

Justification:

LL1 parsing saves work, avoiding construction of errors. It allows for consistency-checking of the grammar and automatic error-detection and possibly recovery in the resulting parser

PARSER SOURCE FILE

```
#####
# IMPORTS
#####

from basic_scanner import *

#####
# 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

    def __repr__(self):
        return f'{self.element_nodes}'

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

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

```

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

    def __repr__(self):
        return f'{self.var_name_tok, self.value_node}'

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

    def __repr__(self):
        return f'({self.cases}, {self.else_case})'

```

```

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

    def __repr__(self):
        return f'{self.var_name_tok, self.start_value_node,
self.end_value_node,self.step_value_node, self.body_node,
self.should_return_null}'

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

    def __repr__(self):
        return f'{self.condition_node, self.body_node, self.should_return_null}'

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

```

```

    def __repr__(self):
        return f'{self.var_name_tok, self.arg_name_toks, self.body_node,
self.should_auto_return}'

```

```

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

    def __repr__(self):
        return f'{self.node_to_call, self.arg_nodes}'

```

```

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

    def __repr__(self):
        return f'{self.node_to_return}'

```

```

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)

def execute_run(fn):

    with open(fn, "r") as f:
        script = f.read()

    ast = run(fn, script)

    return ast[0], ast[1]

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

    return ast.node, ast.error
```

RUN PARSER SOURCE FILE

```
#-----
#     MAIN Parser
#-----
```

```
#####
#     IMPORTS
#####

import basic_parser

while True:
    text = input('basic scanner> ')
    if text.strip() == "": continue

    if 'RUN' in text:
        fn = text.strip("\n")
        result, error = basic_parser.execute_run(fn)
    else:
        result, error = basic_parser.run('<stdin>', text)

    if error:
        print(error.as_string())
    elif result:
        print(repr(result))
```

```
dmore@dmore: ~/Year-3.2/326_compiler_construction
dmore@dmore: ~/Year-3.2/326_compiler_construction 142x37
dmore@dmore:~/Year-3.2/326_compiler_construction$ python3 main_parser.py
basic scanner> RUN("test")
[(IDENTIFIER:oopify, [IDENTIFIER:prefix], (IDENTIFIER:prefix, PLUS, STRING:oop), True), (IDENTIFIER:join, [IDENTIFIER:elements, IDENTIFIER:separator], [(IDENTIFIER:result, STRING), (IDENTIFIER:len, (IDENTIFIER:LEN, [IDENTIFIER:elements])), (IDENTIFIER:l, INT, IDENTIFIER:len, None, [(IDENTIFIER:result, (IDENTIFIER:result, PLUS, (IDENTIFIER:elements, DIV, IDENTIFIER:l))), [(IDENTIFIER:l, NE, (IDENTIFIER:len, MINUS, INT:1)), (IDENTIFIER:result, (IDENTIFIER:result, PLUS, IDENTIFIER:separator)), False], None)], True), IDENTIFIER:result], False), (IDENTIFIER:map, [IDENTIFIER:elements, IDENTIFIER:func], [(IDENTIFIER:new_elements, []), (IDENTIFIER:l, INT, (IDENTIFIER:LEN, [IDENTIFIER:elements]), None, [(IDENTIFIER:APPEND, [IDENTIFIER:new_elements, (IDENTIFIER:func, [(IDENTIFIER:elements, DIV, IDENTIFIER:l)]))), True), IDENTIFIER:new_elements], False), (IDENTIFIER:PRINT, [STRING:Greetings universe!]), (IDENTIFIER:l, INT, INT:5, None, [(IDENTIFIER:PRINT, [(IDENTIFIER:join, [(IDENTIFIER:map, [[STRING:l, STRING:sp], IDENTIFIER:oopify]), STRING:, ]]]), True])], True)]
basic scanner>
```

:PARSER OUTPUT ON SAMPLE CODE