CS 609: Final Project

Fall 2024

Dr. Indranil Roy

Group Members:

Sri Pavan Kalyan Reddy Gottam

Elaheh Beheshti

Ashok Chennareddy

Sai Prashanth Reddy Dyapa

Project Overview:

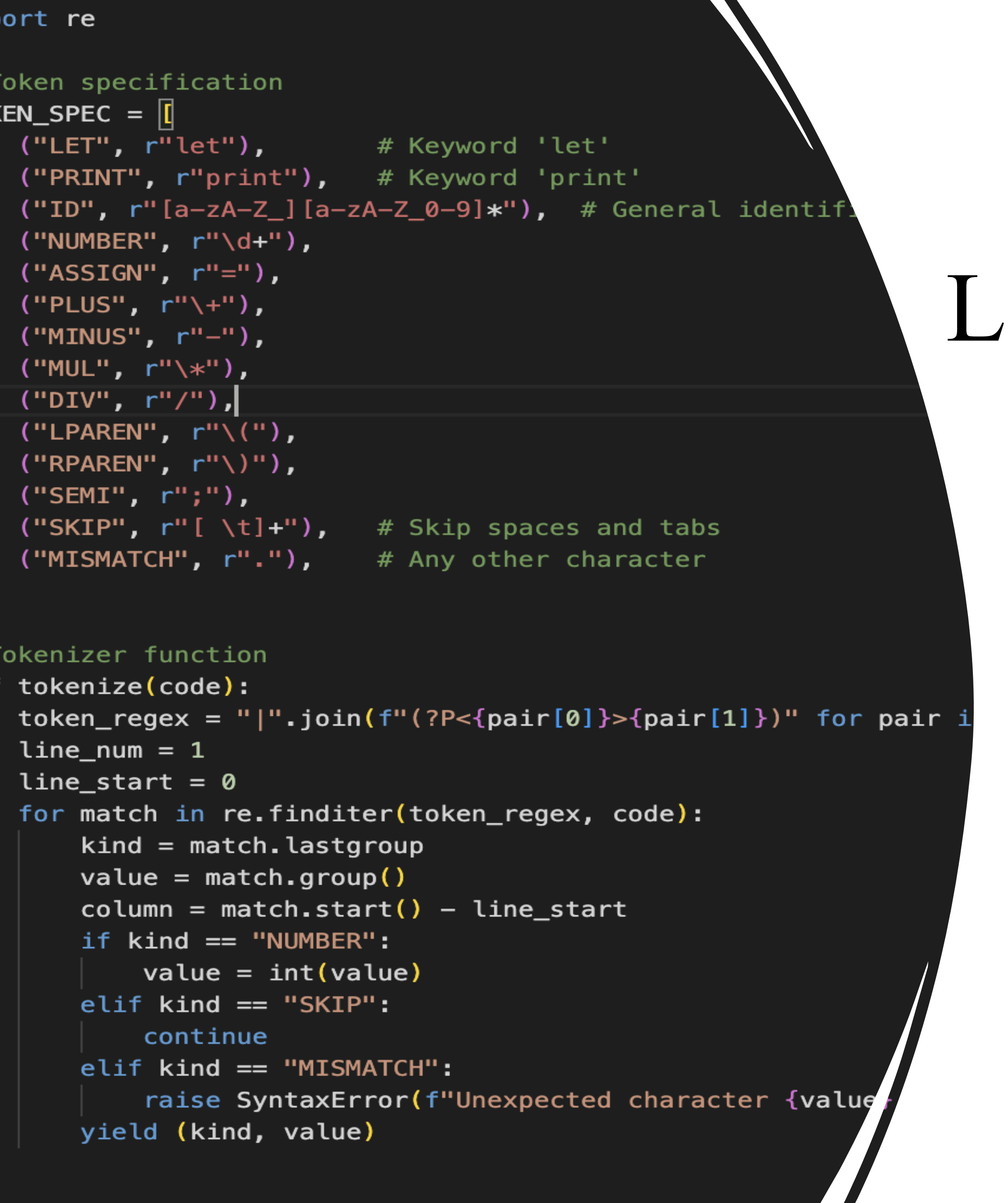
Goal: Learn how interpreters work, focusing on tokenization, parsing, and interpreting Abstract Syntax Trees (AST).

Key Components:

- Lexer (Tokenization) **(Ashok Chennareddy)**
- Parser (AST Generation) **(Sri Pavan Kalyan Reddy Gottam)**
- AST Nodes (Logical Representation) **(Sai Prashanth Reddy Dyapa)**
- Interpreter (Execution) **(Elaheh Beheshti)**

Lexer (Tokenization)

- **File:** lexer.py
- **Purpose:** Convert the input code into a sequence of tokens.
- **Key Features:**
    - Recognizes LET, PRINT, identifiers, numbers, operators, and semicolons.
    - Skips whitespace and raises errors for unexpected characters.
- **Code Snippet:**

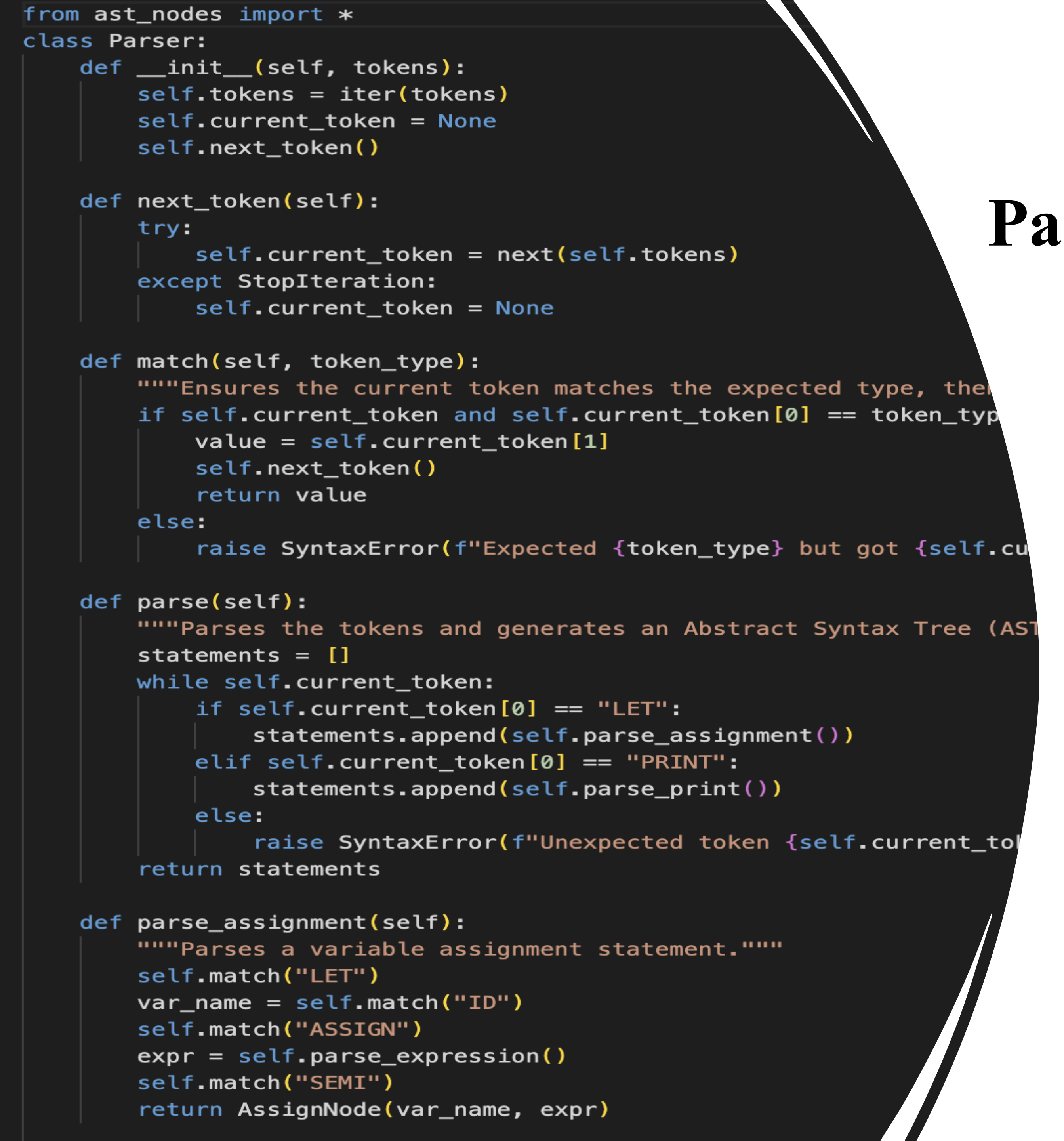


```
ort re

oken specification
EN_SPEC = [
 ("LET", r"let"),       # Keyword 'let'
 ("PRINT", r"print"),   # Keyword 'print'
 ("ID", r"[a-zA-Z_][a-zA-Z_0-9]*"),  # General identifi
 ("NUMBER", r"\d+"),
 ("ASSIGN", r"="),
 ("PLUS", r"\+"),
 ("MINUS", r"-"),
 ("MUL", r"\*"),
 ("DIV", r"/"),
 ("LPAREN", r"\("),
 ("RPAREN", r"\)"),
 ("SEMI", r";"),
 ("SKIP", r"[ \t]+"),   # Skip spaces and tabs
 ("MISMATCH", r"."),    # Any other character


okenizer function
 tokenize(code):
 token_regex = "|".join(f"(?P<{pair[0]}>{pair[1]})" for pair i
 line_num = 1
 line_start = 0
 for match in re.finditer(token_regex, code):
     kind = match.lastgroup
     value = match.group()
     column = match.start() - line_start
     if kind == "NUMBER":
         value = int(value)
     elif kind == "SKIP":
         continue
     elif kind == "MISMATCH":
         raise SyntaxError(f"Unexpected character {value
     yield (kind, value)
```

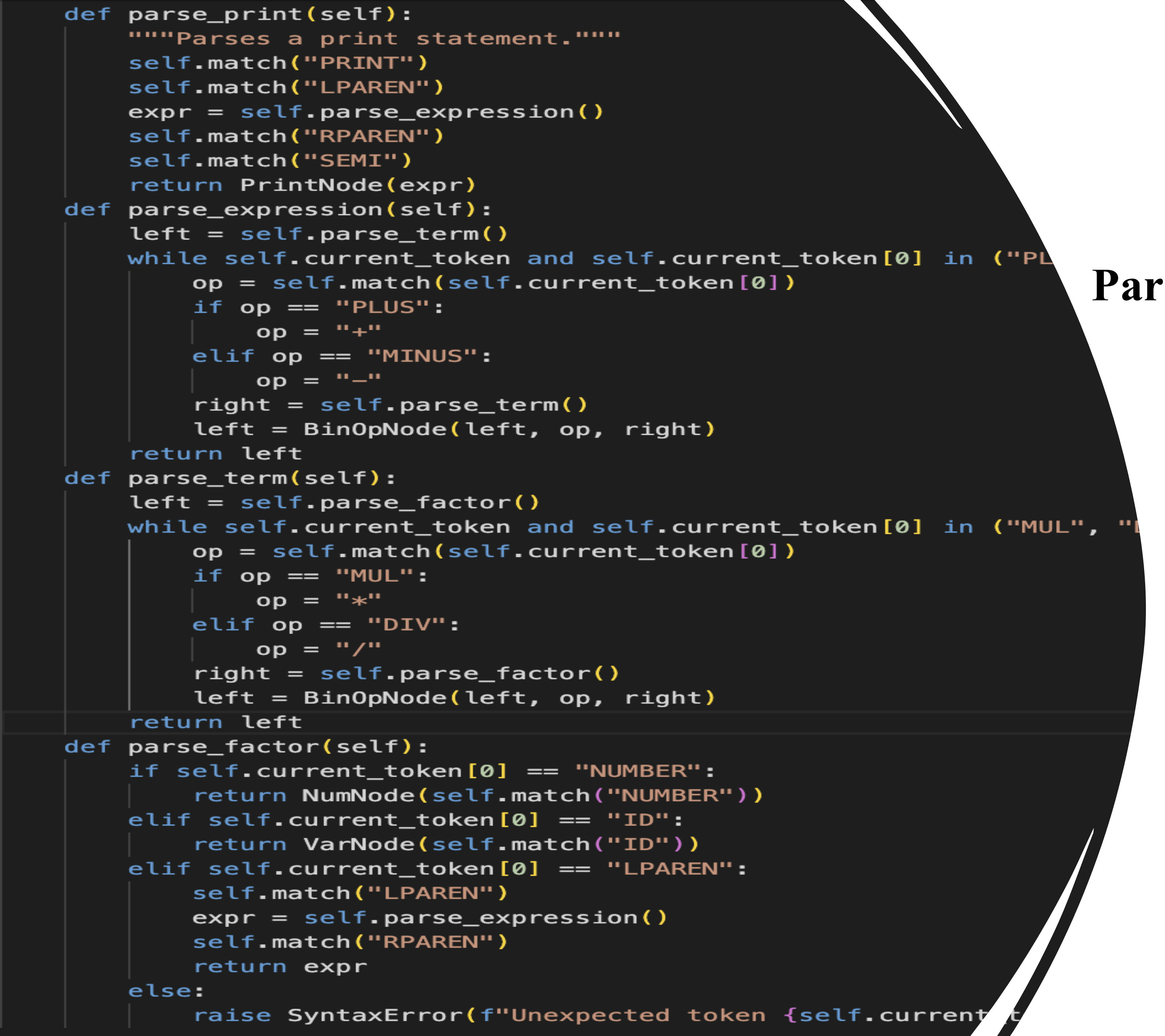
- **Example Output:** Input: let x = 10 + 5; Tokens: [('LET', 'let'), ('ID', 'x'), ('ASSIGN', '='), ('NUMBER', 10), ('PLUS', '+'), ...]

Parser (AST Generation)Part 1

- **File:** parser.py
- **Purpose:** Convert tokens into an Abstract Syntax Tree (AST).
- **Key Features:**
    - Supports variable assignments and print statements.
    - Handles arithmetic expressions with operator precedence.
- **Code Example:** Parsing an assignment:



```
from ast_nodes import *
class Parser:
    def __init__(self, tokens):
        self.tokens = iter(tokens)
        self.current_token = None
        self.next_token()

    def next_token(self):
        try:
            self.current_token = next(self.tokens)
        except StopIteration:
            self.current_token = None

    def match(self, token_type):
        """Ensures the current token matches the expected type, the
        if self.current_token and self.current_token[0] == token_typ
            value = self.current_token[1]
            self.next_token()
            return value
        else:
            raise SyntaxError(f"Expected {token_type} but got {self.cu

    def parse(self):
        """Parses the tokens and generates an Abstract Syntax Tree (AST
        statements = []
        while self.current_token:
            if self.current_token[0] == "LET":
                statements.append(self.parse_assignment())
            elif self.current_token[0] == "PRINT":
                statements.append(self.parse_print())
            else:
                raise SyntaxError(f"Unexpected token {self.current_to
        return statements

    def parse_assignment(self):
        """Parses a variable assignment statement."""
        self.match("LET")
        var_name = self.match("ID")
        self.match("ASSIGN")
        expr = self.parse_expression()
        self.match("SEMI")
        return AssignNode(var_name, expr)
```



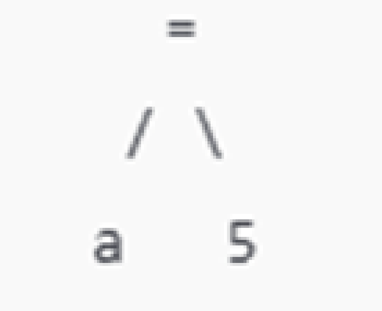
```
    def parse_print(self):
        """Parses a print statement."""
        self.match("PRINT")
        self.match("LPAREN")
        expr = self.parse_expression()
        self.match("RPAREN")
        self.match("SEMI")
        return PrintNode(expr)
    def parse_expression(self):
        left = self.parse_term()
        while self.current_token and self.current_token[0] in ("PL
            op = self.match(self.current_token[0])
            if op == "PLUS":
                op = "+"
            elif op == "MINUS":
                op = "-"
            right = self.parse_term()
            left = BinOpNode(left, op, right)
        return left
    def parse_term(self):
        left = self.parse_factor()
        while self.current_token and self.current_token[0] in ("MUL", "
            op = self.match(self.current_token[0])
            if op == "MUL":
                op = "*"
            elif op == "DIV":
                op = "/"
            right = self.parse_factor()
            left = BinOpNode(left, op, right)
        return left
    def parse_factor(self):
        if self.current_token[0] == "NUMBER":
            return NumNode(self.match("NUMBER"))
        elif self.current_token[0] == "ID":
            return VarNode(self.match("ID"))
        elif self.current_token[0] == "LPAREN":
            self.match("LPAREN")
            expr = self.parse_expression()
            self.match("RPAREN")
            return expr
        else:
            raise SyntaxError(f"Unexpected token {self.current_t
```

Parser (AST Generation)Part 2

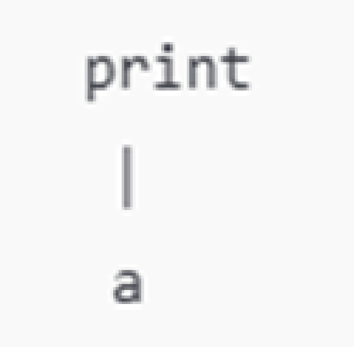
- **Output:**
- AssignNode(var=x, expr=BinOpNode(left=10, op=+, right=5))

Abstract Syntax Tree (AST) Overview

- Tree-like data structure used in programming language interpreters and compilers.
- it provides an organized and simplified way to analyze, transform, and execute programs.
- Sample tree structures



```
    =
   / \
  a   5
```



```
print
  |
  a
```

Key Characteristics of AST

- **Node Representation**

  categorized as expressions, statements, or declarations.

- **Hierarchy**

  The **root node** represents the entire program.

  Child nodes represent components or sub-expressions.

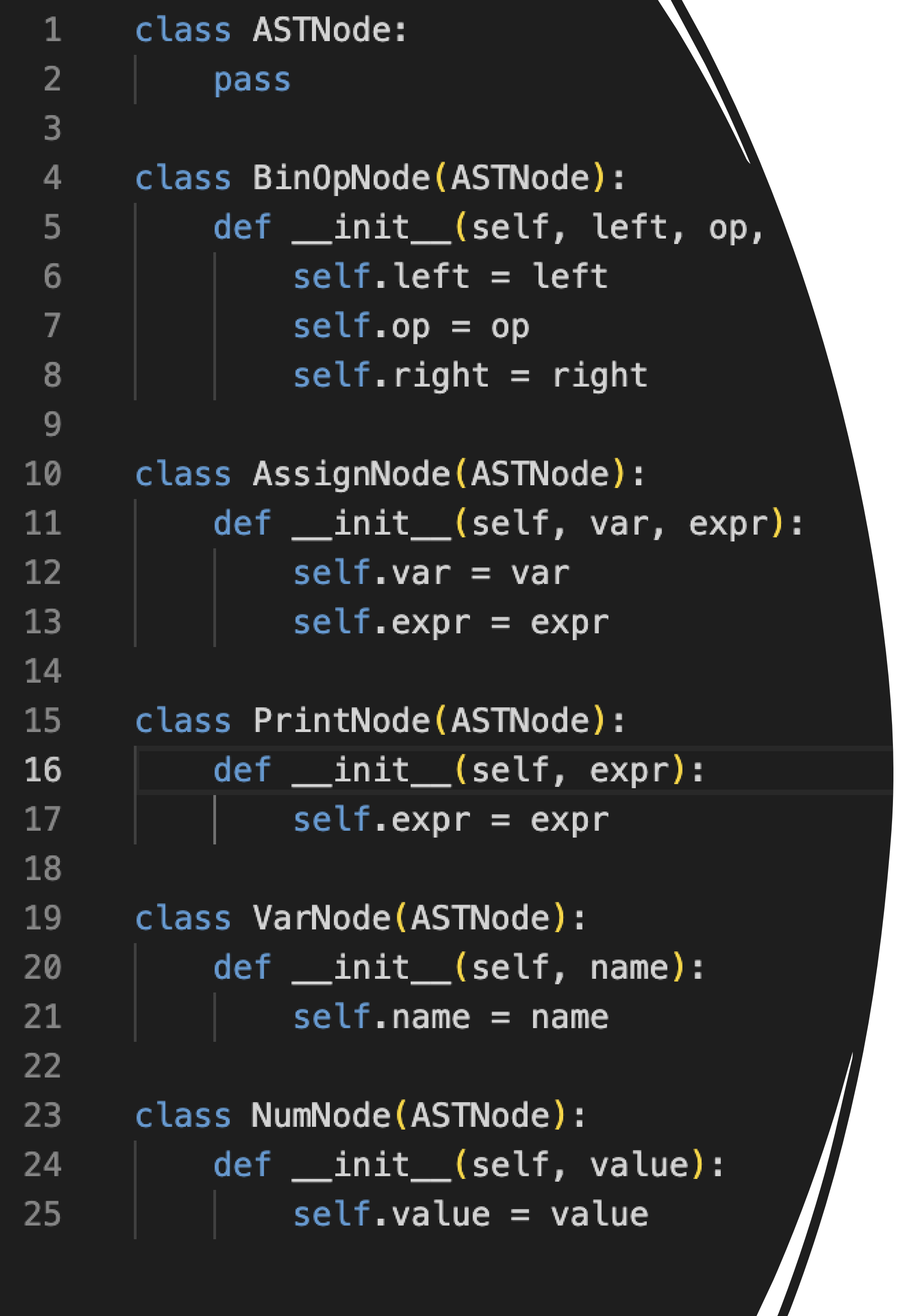
- **Abstract Representation**

  Focuses on the meaningful components of the code, ignoring irrelevant syntax.

- **Semantic Information**

  Store additional details about the program, such as variable types, scopes, and function definitions.

AST Nodes

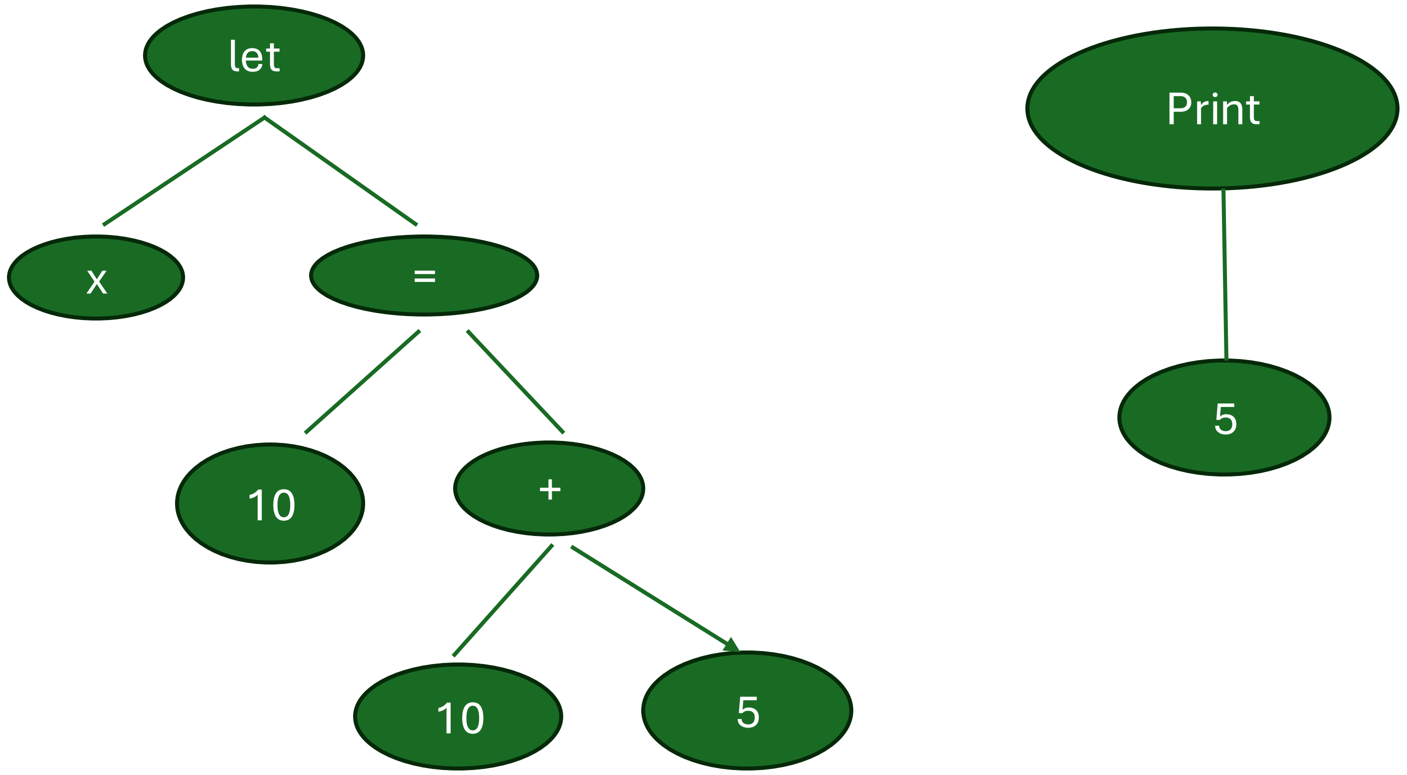


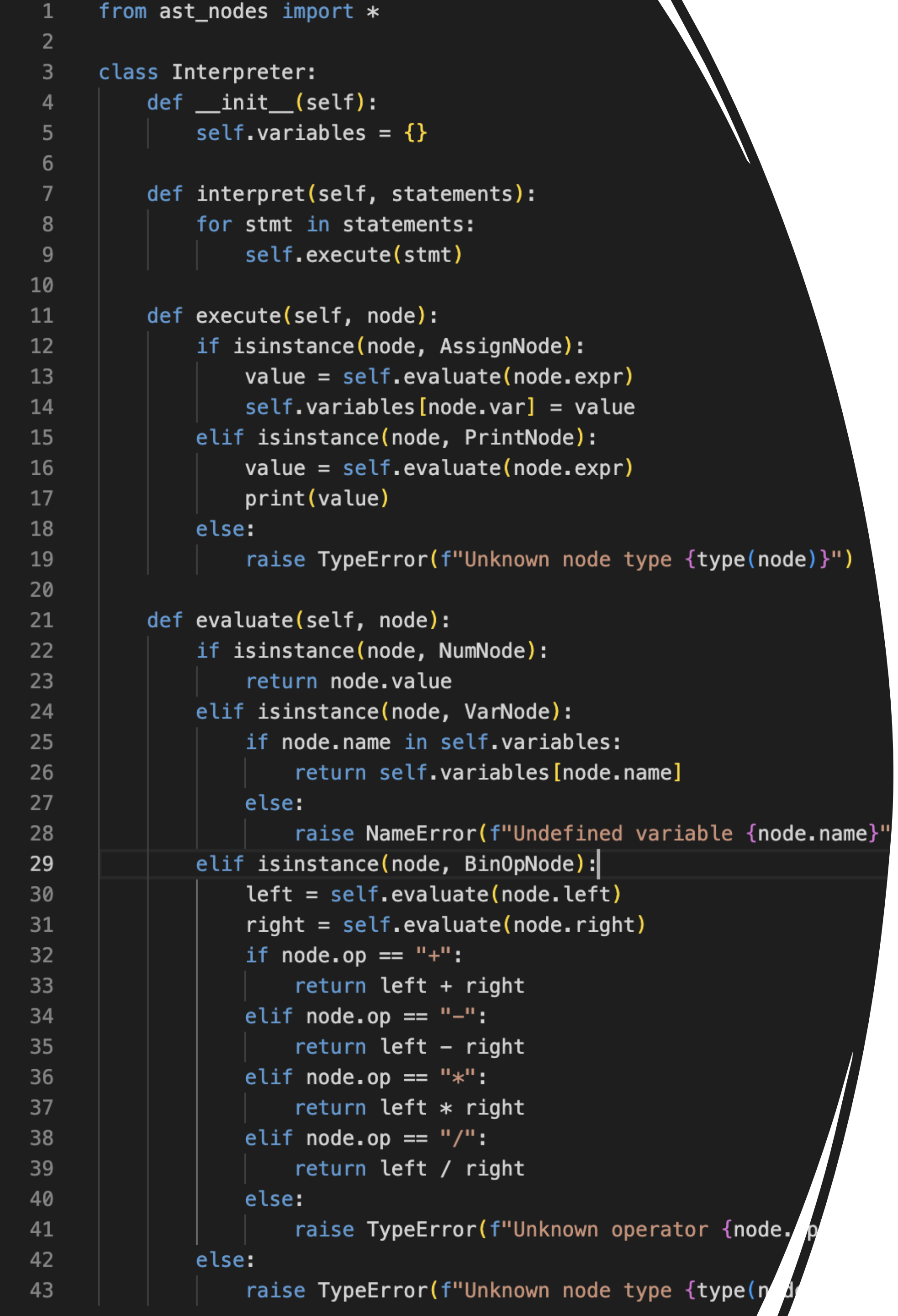
```python
class ASTNode:
    pass

class BinOpNode(ASTNode):
    def __init__(self, left, op,
        self.left = left
        self.op = op
        self.right = right

class AssignNode(ASTNode):
    def __init__(self, var, expr):
        self.var = var
        self.expr = expr

class PrintNode(ASTNode):
    def __init__(self, expr):
        self.expr = expr

class VarNode(ASTNode):
    def __init__(self, name):
        self.name = name

class NumNode(ASTNode):
    def __init__(self, value):
        self.value = value
```

- **File:** ast_nodes.py
- **Purpose:** Represent the logical structure of the program.
- **Node Types:**
  - BinOpNode: Represents arithmetic operations.
  - AssignNode: Represents variable assignments.
  - PrintNode: Represents print statements.

Example

- Tree for **let x = 10 + 5 and print 5**

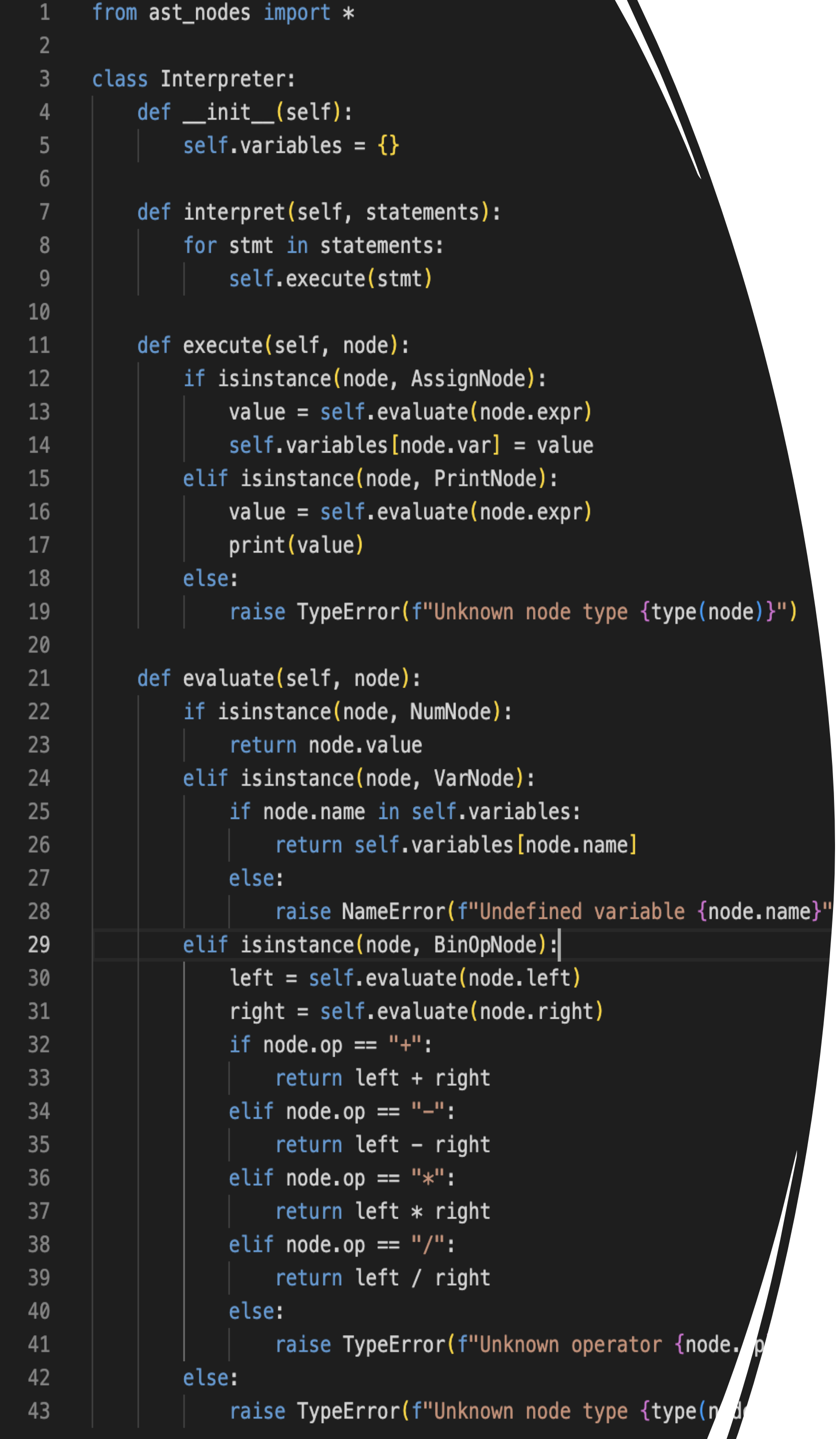




```python
from ast_nodes import *

class Interpreter:
    def __init__(self):
        self.variables = {}

    def interpret(self, statements):
        for stmt in statements:
            self.execute(stmt)

    def execute(self, node):
        if isinstance(node, AssignNode):
            value = self.evaluate(node.expr)
            self.variables[node.var] = value
        elif isinstance(node, PrintNode):
            value = self.evaluate(node.expr)
            print(value)
        else:
            raise TypeError(f"Unknown node type {type(node)}")

    def evaluate(self, node):
        if isinstance(node, NumNode):
            return node.value
        elif isinstance(node, VarNode):
            if node.name in self.variables:
                return self.variables[node.name]
            else:
                raise NameError(f"Undefined variable {node.name}"
        elif isinstance(node, BinOpNode):
            left = self.evaluate(node.left)
            right = self.evaluate(node.right)
            if node.op == "+":
                return left + right
            elif node.op == "-":
                return left - right
            elif node.op == "*":
                return left * right
            elif node.op == "/":
                return left / right
            else:
                raise TypeError(f"Unknown operator {node.
        else:
            raise TypeError(f"Unknown node type {type(n
```

Milestone 3: Interpreter

- **Title**: "Understanding the interpreter.py"
- Input Code → Lexer → Parser → AST → **Interpreter** → Output
- **Input Code:**
- let x = 10 + 5;
- let y = x * 2;
- print(x);
- print(y);
- **Output:**
- 15
- 30
- The interpreter **takes the AST** created by the parser.
- It **evaluates variable** assignments, **arithmetic operations**, and print statements.
- Executes statements **sequentially.**



```
from ast_nodes import *

class Interpreter:
    def __init__(self):
        self.variables = {}

    def interpret(self, statements):
        for stmt in statements:
            self.execute(stmt)

    def execute(self, node):
        if isinstance(node, AssignNode):
            value = self.evaluate(node.expr)
            self.variables[node.var] = value
        elif isinstance(node, PrintNode):
            value = self.evaluate(node.expr)
            print(value)
        else:
            raise TypeError(f"Unknown node type {type(node)}")

    def evaluate(self, node):
        if isinstance(node, NumNode):
            return node.value
        elif isinstance(node, VarNode):
            if node.name in self.variables:
                return self.variables[node.name]
            else:
                raise NameError(f"Undefined variable {node.name}"
        elif isinstance(node, BinOpNode):
            left = self.evaluate(node.left)
            right = self.evaluate(node.right)
            if node.op == "+":
                return left + right
            elif node.op == "-":
                return left - right
            elif node.op == "*":
                return left * right
            elif node.op == "/":
                return left / right
            else:
                raise TypeError(f"Unknown operator {node.p
        else:
            raise TypeError(f"Unknown node type {type(n
```

"Execution Steps in interpreter.py"

A **simplified AST structure** for the example:

AST:

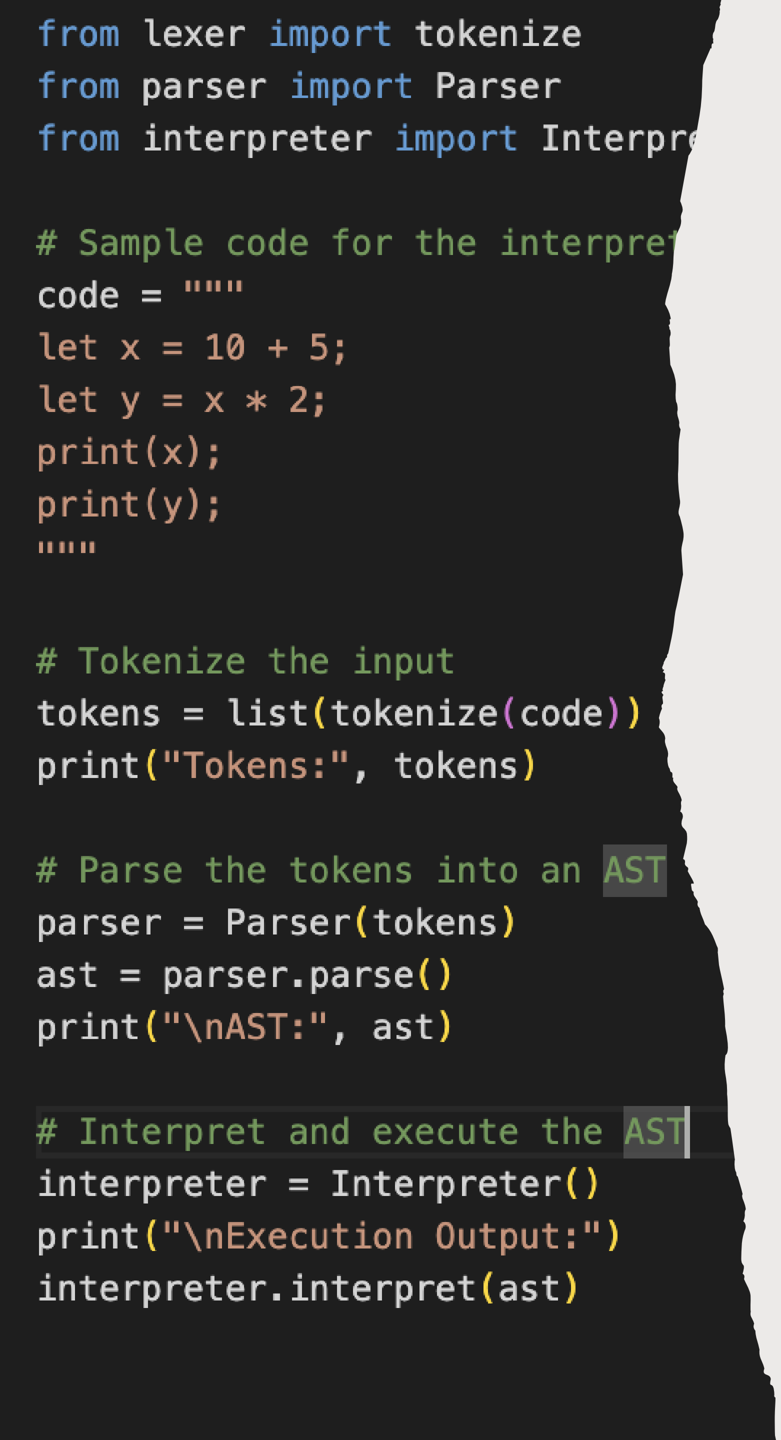
```
[       AssignNode(var='x', expr=BinOpNode(left=NumNode(10), op='+', right=NumNode(5))),
    AssignNode(var='y', expr=BinOpNode(left=VarNode('x'), op='*', right=NumNode(2))),
    PrintNode(expr=VarNode('x')),
    PrintNode(expr=VarNode('y'))
]
```

Highlight the flow:

- Assign x = 15.
- Assign y = 30.
- Print x → 15.
- Print y → 30
- .

Bullet Points:

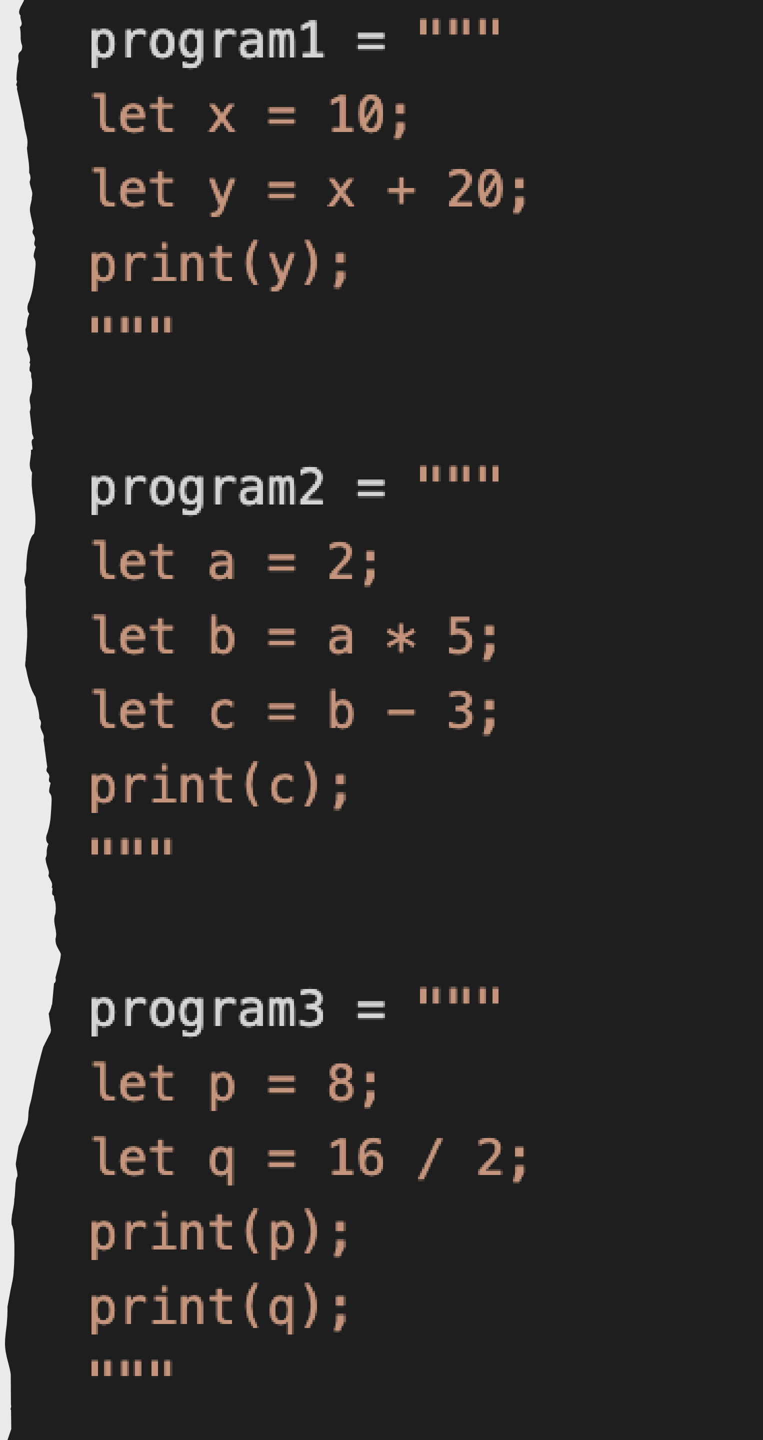
- AssignNode: Evaluates and stores variable values.
- BinOpNode: Performs operations like +, -, *, /.
- PrintNode: Outputs the value of variables or expressions.



```
from lexer import tokenize
from parser import Parser
from interpreter import Interpre

# Sample code for the interpret
code = """
let x = 10 + 5;
let y = x * 2;
print(x);
print(y);
"""


# Tokenize the input
tokens = list(tokenize(code))
print("Tokens:", tokens)

# Parse the tokens into an AST
parser = Parser(tokens)
ast = parser.parse()
print("\nAST:", ast)

# Interpret and execute the AST
interpreter = Interpreter()
print("\nExecution Output:")
interpreter.interpret(ast)
```

Test Program

- **File:** test_program.py
  test2.py
- **Purpose:** Demonstrate the full pipeline from input to execution.
- Example Code:
  - let x = 10 + 5;
  - let y = x * 2;
  - print(x);
  - print(y);
- Process:Tokenization → Parsing → AST Generation → Execution
  - Output:
    - Tokens: [('LET', 'let'), ('ID', 'x'), ...]
    - AST: [AssignNode(...), PrintNode(...)]
    - Execution Output: 15, 30



```
program1 = """
let x = 10;
let y = x + 20;
print(y);
"""


program2 = """
let a = 2;
let b = a * 5;
let c = b - 3;
print(c);
"""


program3 = """
let p = 8;
let q = 16 / 2;
print(p);
print(q);
"""
```

Thank you for your attention and engagement during this presentation. We would also like to extend my heartfelt gratitude to Dr. Indranil Roy for his invaluable guidance and support throughout the semester. His encouragement and insights greatly contributed to the success of this project and our deeper understanding of the subject matter.