BolBachchan Language

Group: 32

Presenters:

Charu Sneha Laguduva Ravi,

Aditya Soude,

Savankumar Pethani,

Vidhisha Amle

Arizona State University

Content

- Introduction to our Language
- Features
- Flow Chart
- Grammar
- Tokenizer
- Parser
- Interpreter
- Main.py
- Instruction to Execute
- Sample Cases

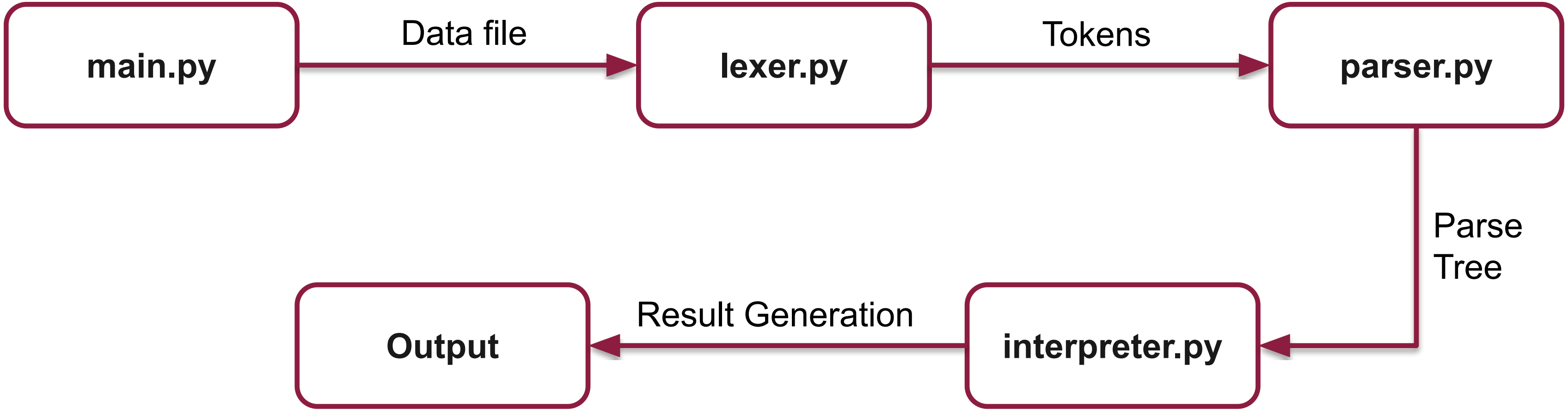
Introduction to BolBachchan

BolBachchan is a creative, humorous programming language that blends Hindi-English keywords with traditional programming constructs. Designed to be fun and expressive, it is backed by a custom lexer and parser written in Python using PLY (Python Lex-Yacc).

Features of BolBachchan

- **Primitive types:** int, bool, string
- **Arithmetic operators:** jodo (+), ghatao (-), guna (*), bhaag (/)
- **Relational operators:** badaHai (>), chhotaHai (<), barabarHai (==)
- **Loops:** baarBaar (for), jabTak (while)
- **Conditionals:** agar-toh-nahiToh (if-then-else), ternary (? :)
- **Print Statements:** bolBhai() prints any data type
- **Assignments:** rakho keyword assigns values to variables
- **Logical operators:** & (and), | (or)
- **Define custom function:** function func_name and wapis to return.

Flow between the Files



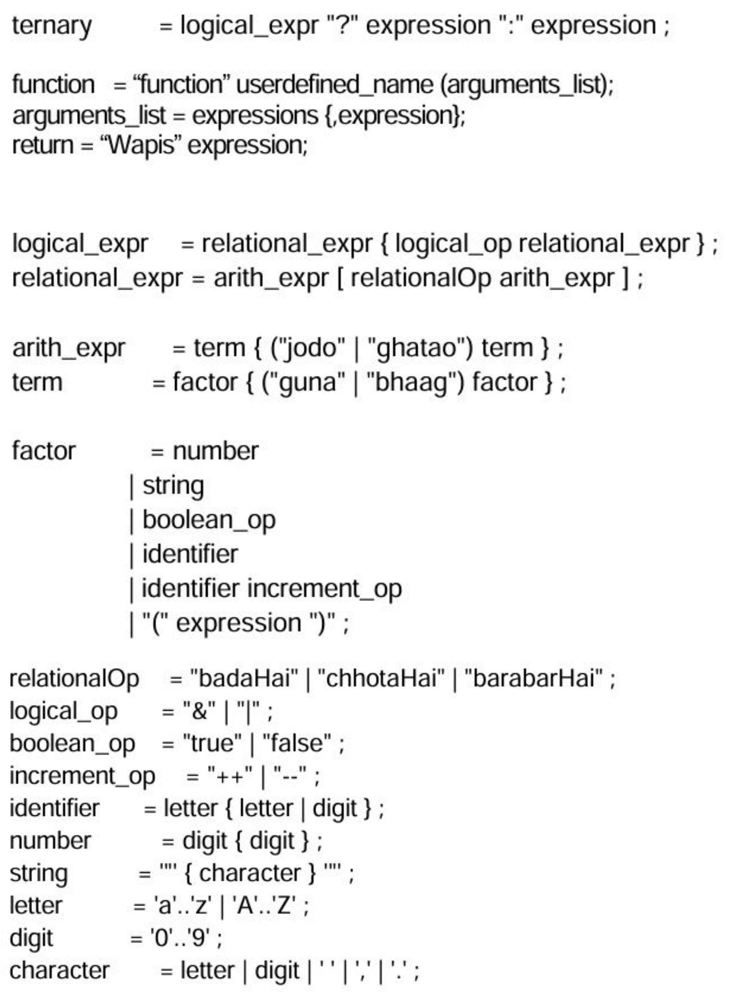
Grammar

We have used **EBNF** (Extended Backus-Naur Form). We thought of using this because:

- Standard format for language specification.
- Easier to read and write for documentation.
- Language-agnostic and widely understood.
- Supported by many parser generators.

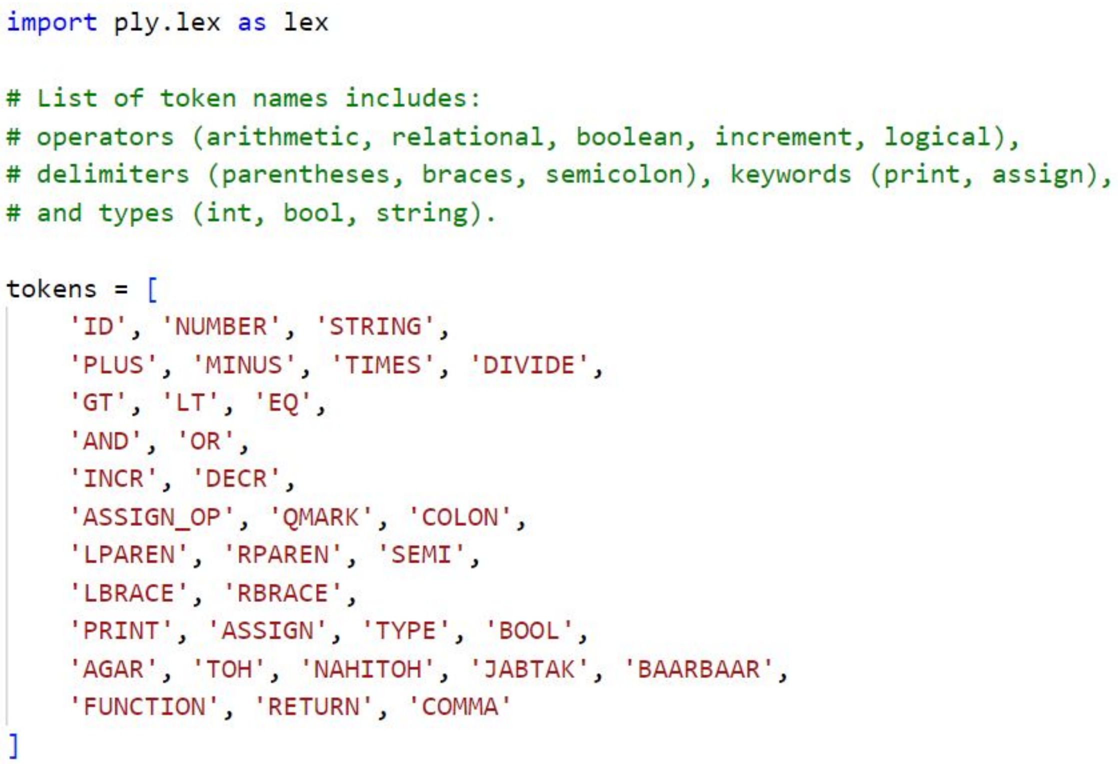
Grammar Code

```
program        = { statement } ;

statement      = declaration
               | assignment
               | print
               | ifStatement
               | whileLoop
               | forLoop
               | expression ";" ;

declaration    = datatype identifier ";" ;
assignment     = "rakho" identifier "=" expression ";" ;
datatype       = "int" | "bool" | "string" ;

print          = "bolBhai" "(" expression ")" ";" ;

ifStatement    = "agar" "(" expression ")" "toh" "{" { statement } "}"
               [ "nahiToh" "{" { statement } "}" ] ;

whileLoop      = "jabTak" "(" expression ")" "{" { statement } "}" ;

forLoop        = "baarBaar" "(" assignment expression ";" assignment ")"
               "{" { statement } "}" ;

expression     = ternary | logical_expr ;
```



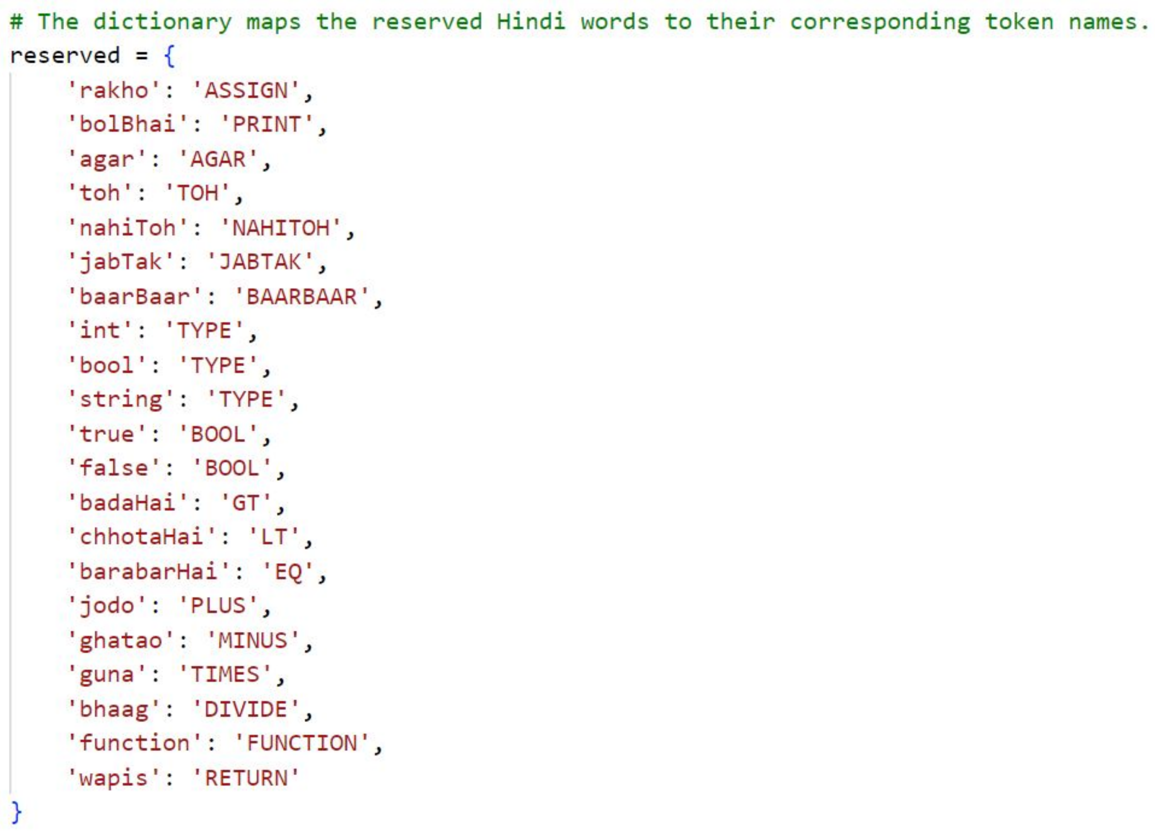
```
ternary        = logical_expr "?" expression ":" expression ;

function  = “function” userdefined_name (arguments_list);
arguments_list = expressions {,expression};
return = “Wapis” expression;


logical_expr    = relational_expr { logical_op relational_expr } ;
relational_expr = arith_expr [ relationalOp arith_expr ] ;

arith_expr     = term { ("jodo" | "ghatao") term } ;
term           = factor { ("guna" | "bhaag") factor } ;

factor         = number
               | string
               | boolean_op
               | identifier
               | identifier increment_op
               | "(" expression ")" ;

relationalOp   = "badaHai" | "chhotaHai" | "barabarHai" ;
logical_op     = "&" | "|" ;
boolean_op     = "true" | "false" ;
increment_op   = "++" | "--" ;
identifier     = letter { letter | digit } ;
number         = digit { digit } ;
string         = '"' { character } '"' ;
letter         = 'a'..'z' | 'A'..'Z' ;
digit          = '0'..'9' ;
character      = letter | digit | ' ' | ',' | '.' ;
```

Tokenizer (lexer.py)



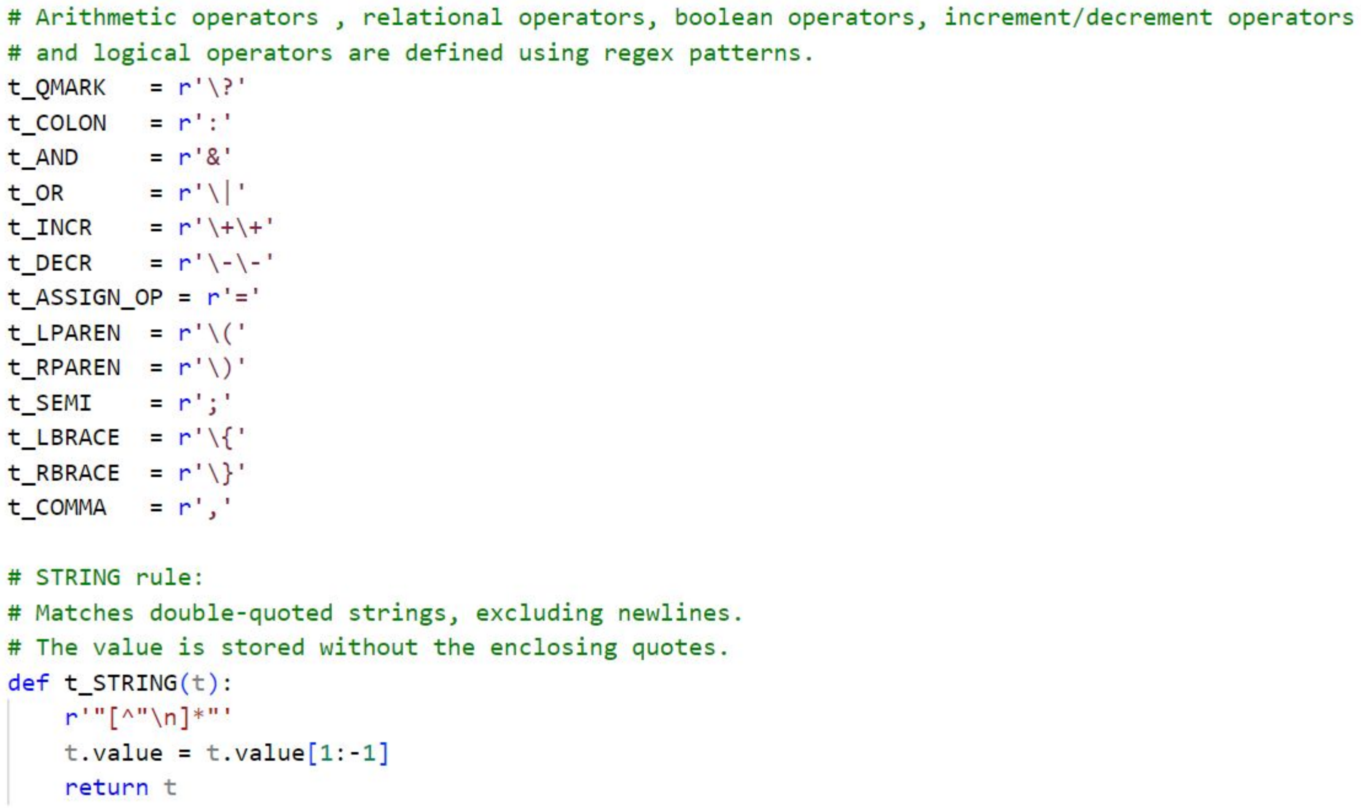
```python
import ply.lex as lex

# List of token names includes:
# operators (arithmetic, relational, boolean, increment, logical),
# delimiters (parentheses, braces, semicolon), keywords (print, assign),
# and types (int, bool, string).

tokens = [
    'ID', 'NUMBER', 'STRING',
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE',
    'GT', 'LT', 'EQ',
    'AND', 'OR',
    'INCR', 'DECR',
    'ASSIGN_OP', 'QMARK', 'COLON',
    'LPAREN', 'RPAREN', 'SEMI',
    'LBRACE', 'RBRACE',
    'PRINT', 'ASSIGN', 'TYPE', 'BOOL',
    'AGAR', 'TOH', 'NAHITOH', 'JABTAK', 'BAARBAAR',
    'FUNCTION', 'RETURN', 'COMMA'
]
```

Tokenizer (lexer.py)



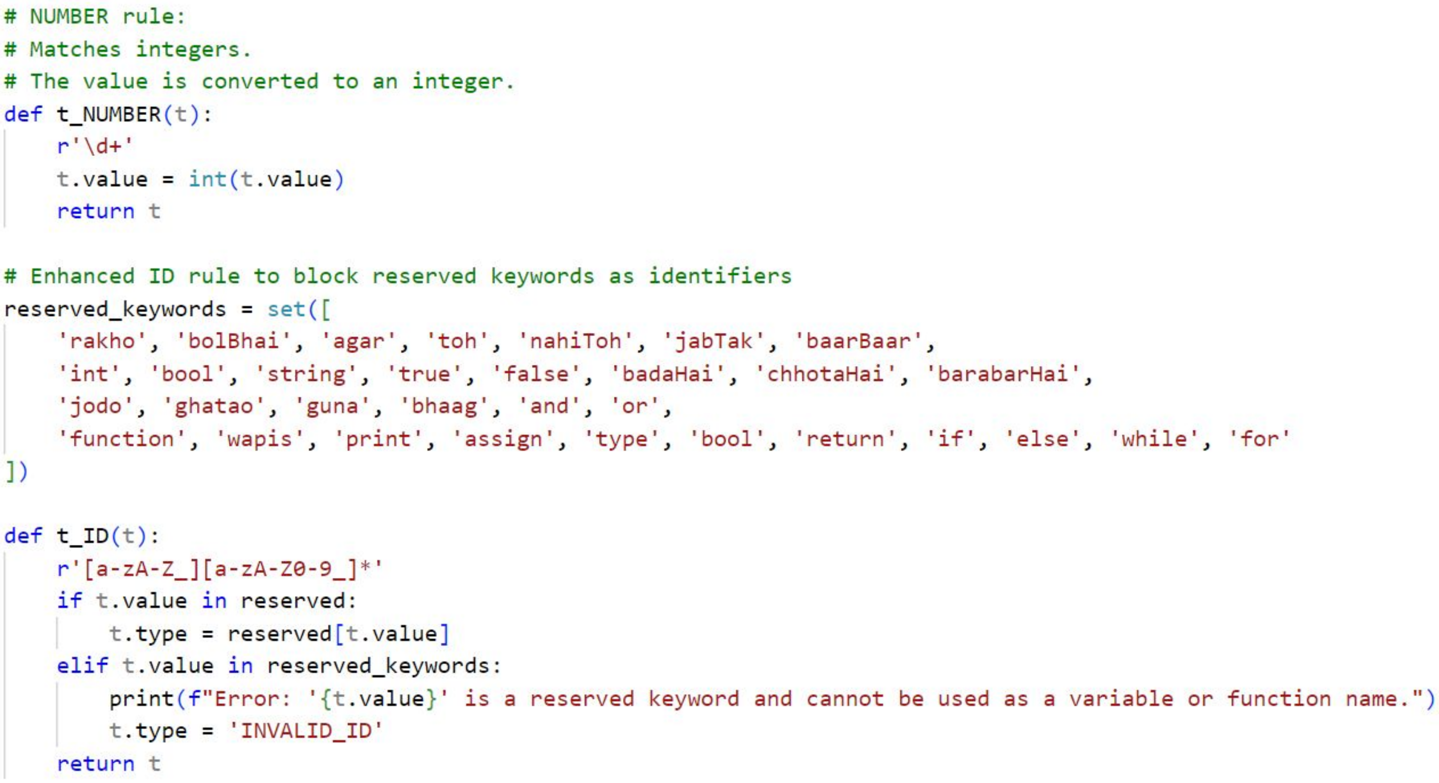
```
# The dictionary maps the reserved Hindi words to their corresponding token names.
reserved = {
    'rakho': 'ASSIGN',
    'bolBhai': 'PRINT',
    'agar': 'AGAR',
    'toh': 'TOH',
    'nahiToh': 'NAHITOH',
    'jabTak': 'JABTAK',
    'baarBaar': 'BAARBAAR',
    'int': 'TYPE',
    'bool': 'TYPE',
    'string': 'TYPE',
    'true': 'BOOL',
    'false': 'BOOL',
    'badaHai': 'GT',
    'chhotaHai': 'LT',
    'barabarHai': 'EQ',
    'jodo': 'PLUS',
    'ghatao': 'MINUS',
    'guna': 'TIMES',
    'bhaag': 'DIVIDE',
    'function': 'FUNCTION',
    'wapis': 'RETURN'
}
```

Tokenizer (lexer.py)



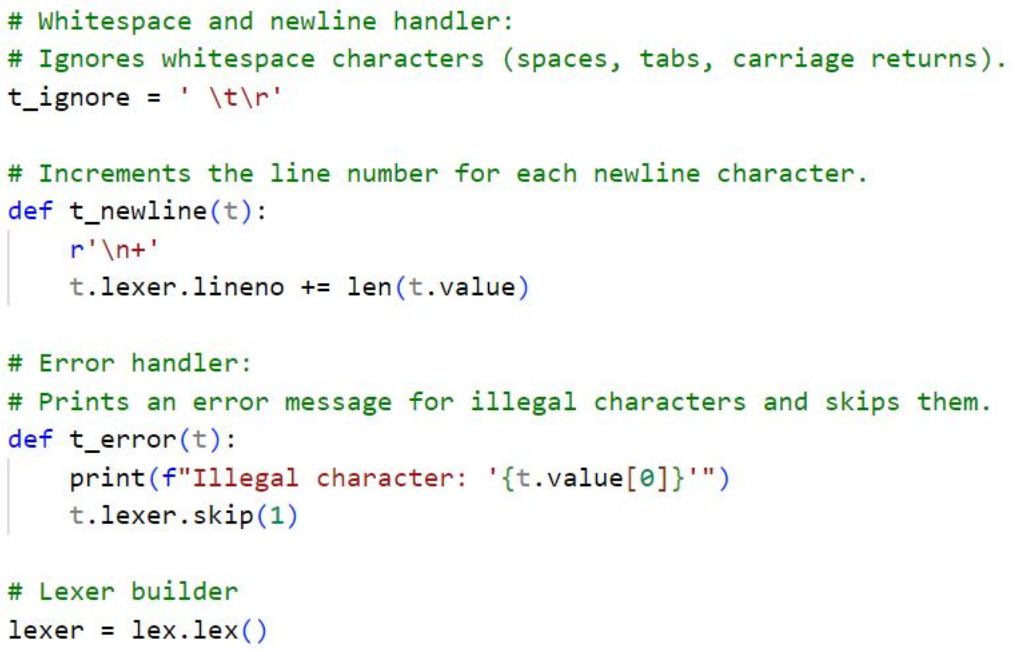
```
# Arithmetic operators , relational operators, boolean operators, increment/decrement operators
# and logical operators are defined using regex patterns.
t_QMARK    = r'\?'
t_COLON    = r':'
t_AND      = r'&'
t_OR       = r'\|'
t_INCR     = r'\+\+'
t_DECR     = r'\-\-'
t_ASSIGN_OP = r'='
t_LPAREN   = r'\('
t_RPAREN   = r'\)'
t_SEMI     = r';'
t_LBRACE   = r'\{'
t_RBRACE   = r'\}'
t_COMMA    = r','

# STRING rule:
# Matches double-quoted strings, excluding newlines.
# The value is stored without the enclosing quotes.
def t_STRING(t):
    r'"[^"\n]*"'
    t.value = t.value[1:-1]
    return t
```

Tokenizer (lexer.py)



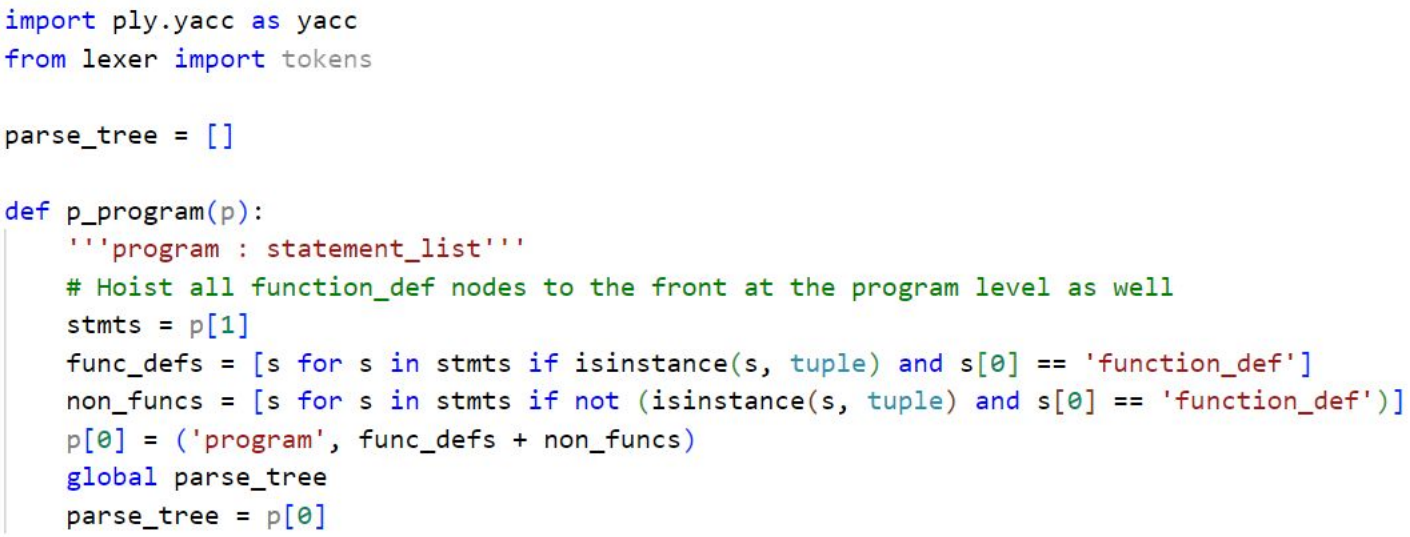
```python
# NUMBER rule:
# Matches integers.
# The value is converted to an integer.
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Enhanced ID rule to block reserved keywords as identifiers
reserved_keywords = set([
    'rakho', 'bolBhai', 'agar', 'toh', 'nahiToh', 'jabTak', 'baarBaar',
    'int', 'bool', 'string', 'true', 'false', 'badaHai', 'chhotaHai', 'barabarHai',
    'jodo', 'ghatao', 'guna', 'bhaag', 'and', 'or',
    'function', 'wapis', 'print', 'assign', 'type', 'bool', 'return', 'if', 'else', 'while', 'for'
])

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    if t.value in reserved:
        t.type = reserved[t.value]
    elif t.value in reserved_keywords:
        print(f"Error: '{t.value}' is a reserved keyword and cannot be used as a variable or function name.")
        t.type = 'INVALID_ID'
    return t
```

Tokenizer (lexer.py)



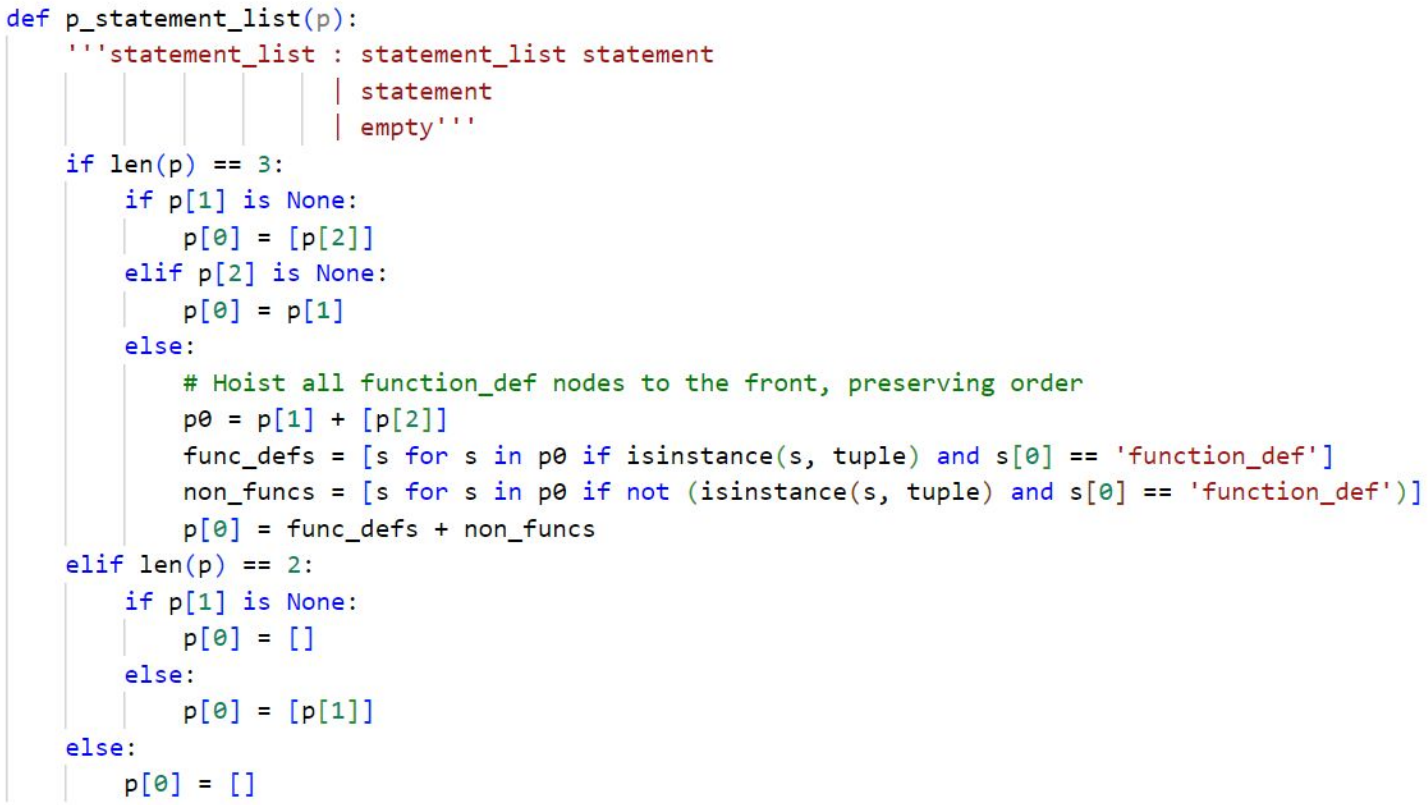
```python
# Whitespace and newline handler:
# Ignores whitespace characters (spaces, tabs, carriage returns).
t_ignore = ' \t\r'

# Increments the line number for each newline character.
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# Error handler:
# Prints an error message for illegal characters and skips them.
def t_error(t):
    print(f"Illegal character: '{t.value[0]}'")
    t.lexer.skip(1)

# Lexer builder
lexer = lex.lex()
```

Generating Parse Tree (parser.py)



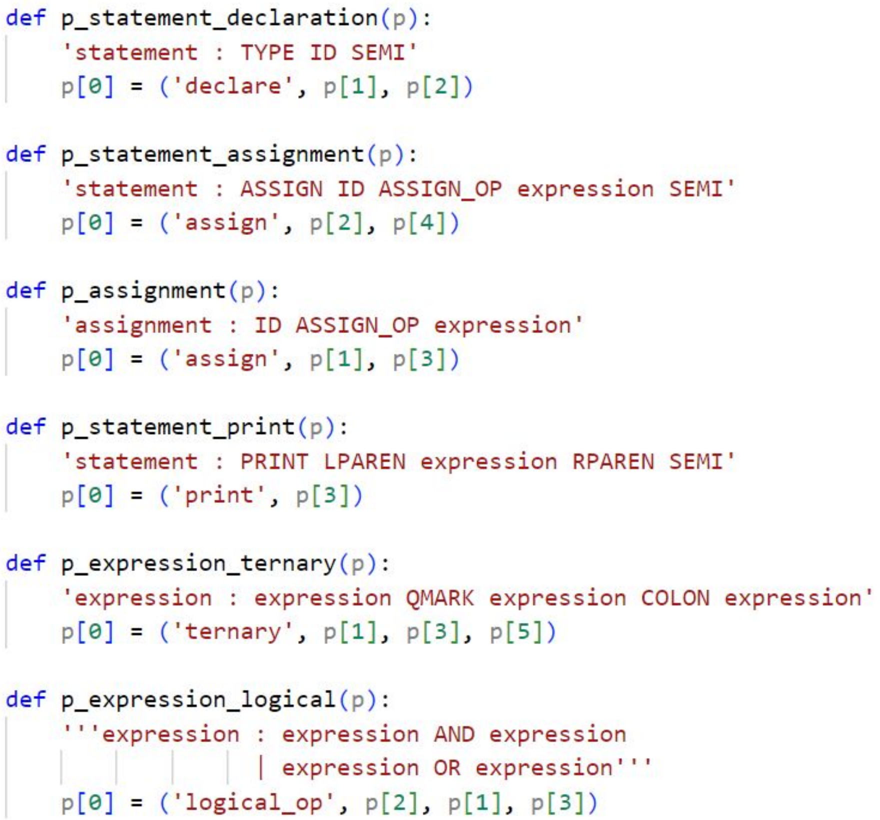
```python
import ply.yacc as yacc
from lexer import tokens

parse_tree = []

def p_program(p):
    '''program : statement_list'''
    # Hoist all function_def nodes to the front at the program level as well
    stmts = p[1]
    func_defs = [s for s in stmts if isinstance(s, tuple) and s[0] == 'function_def']
    non_funcs = [s for s in stmts if not (isinstance(s, tuple) and s[0] == 'function_def')]
    p[0] = ('program', func_defs + non_funcs)
    global parse_tree
    parse_tree = p[0]
```

Generating Parse Tree (parser.py)



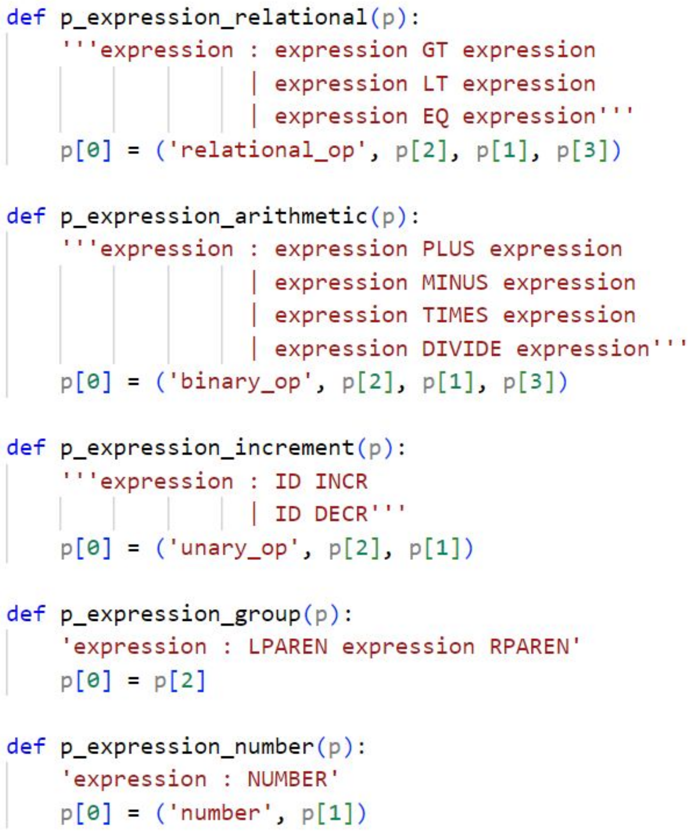
```python
def p_statement_list(p):
    '''statement_list : statement_list statement
                      | statement
                      | empty'''
    if len(p) == 3:
        if p[1] is None:
            p[0] = [p[2]]
        elif p[2] is None:
            p[0] = p[1]
        else:
            # Hoist all function_def nodes to the front, preserving order
            p0 = p[1] + [p[2]]
            func_defs = [s for s in p0 if isinstance(s, tuple) and s[0] == 'function_def']
            non_funcs = [s for s in p0 if not (isinstance(s, tuple) and s[0] == 'function_def')]
            p[0] = func_defs + non_funcs
    elif len(p) == 2:
        if p[1] is None:
            p[0] = []
        else:
            p[0] = [p[1]]
    else:
        p[0] = []
```

Generating Parse Tree (parser.py)



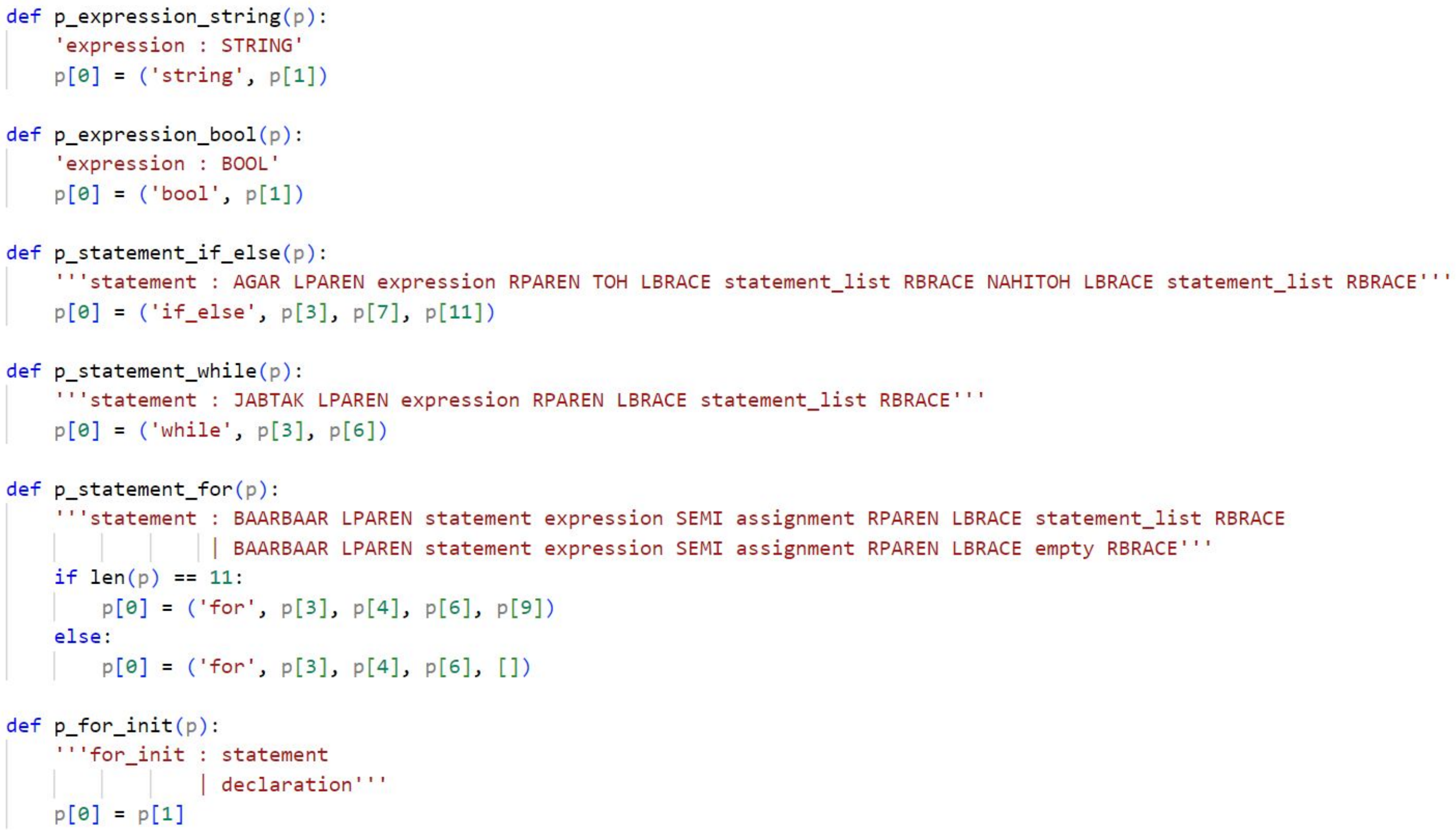
```python
def p_statement_declaration(p):
    'statement : TYPE ID SEMI'
    p[0] = ('declare', p[1], p[2])

def p_statement_assignment(p):
    'statement : ASSIGN ID ASSIGN_OP expression SEMI'
    p[0] = ('assign', p[2], p[4])

def p_assignment(p):
    'assignment : ID ASSIGN_OP expression'
    p[0] = ('assign', p[1], p[3])

def p_statement_print(p):
    'statement : PRINT LPAREN expression RPAREN SEMI'
    p[0] = ('print', p[3])

def p_expression_ternary(p):
    'expression : expression QMARK expression COLON expression'
    p[0] = ('ternary', p[1], p[3], p[5])

def p_expression_logical(p):
    '''expression : expression AND expression
                  | expression OR expression'''
    p[0] = ('logical_op', p[2], p[1], p[3])
```

Generating Parse Tree (parser.py)



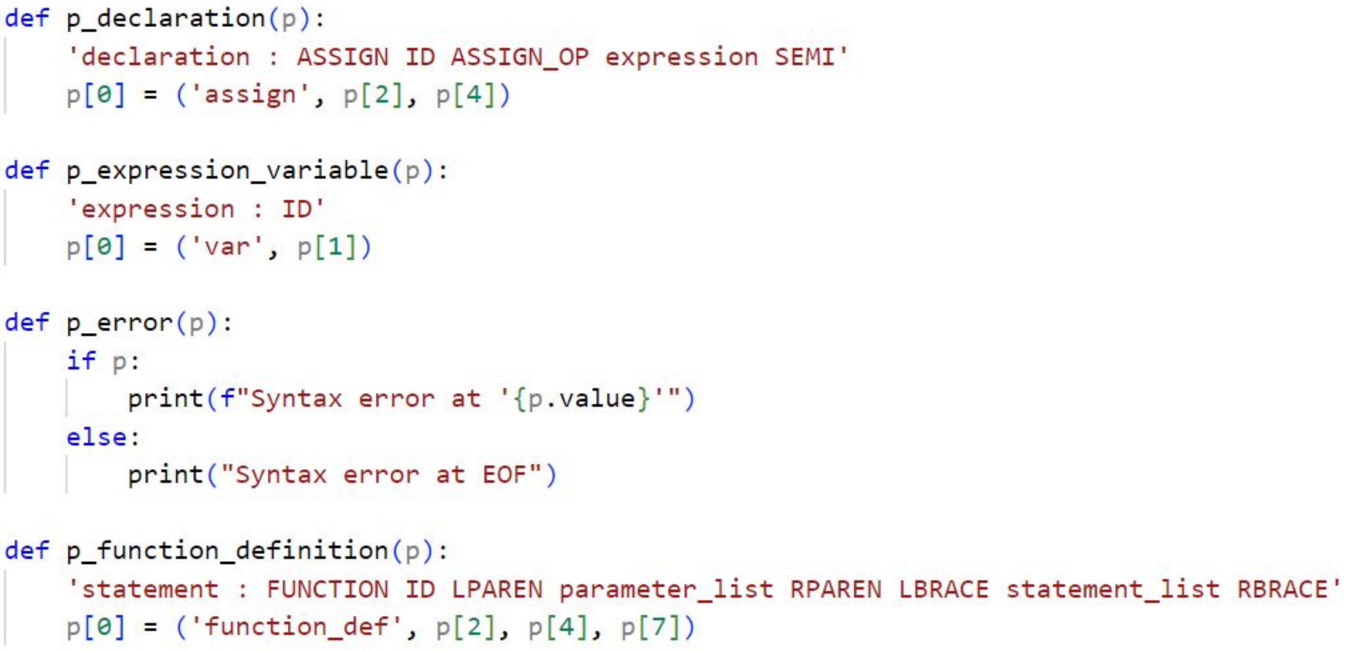
```
def p_expression_relational(p):
    '''expression : expression GT expression
                  | expression LT expression
                  | expression EQ expression'''
    p[0] = ('relational_op', p[2], p[1], p[3])

def p_expression_arithmetic(p):
    '''expression : expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression'''
    p[0] = ('binary_op', p[2], p[1], p[3])

def p_expression_increment(p):
    '''expression : ID INCR
                  | ID DECR'''
    p[0] = ('unary_op', p[2], p[1])

def p_expression_group(p):
    'expression : LPAREN expression RPAREN'
    p[0] = p[2]

def p_expression_number(p):
    'expression : NUMBER'
    p[0] = ('number', p[1])
```

Generating Parse Tree (parser.py)



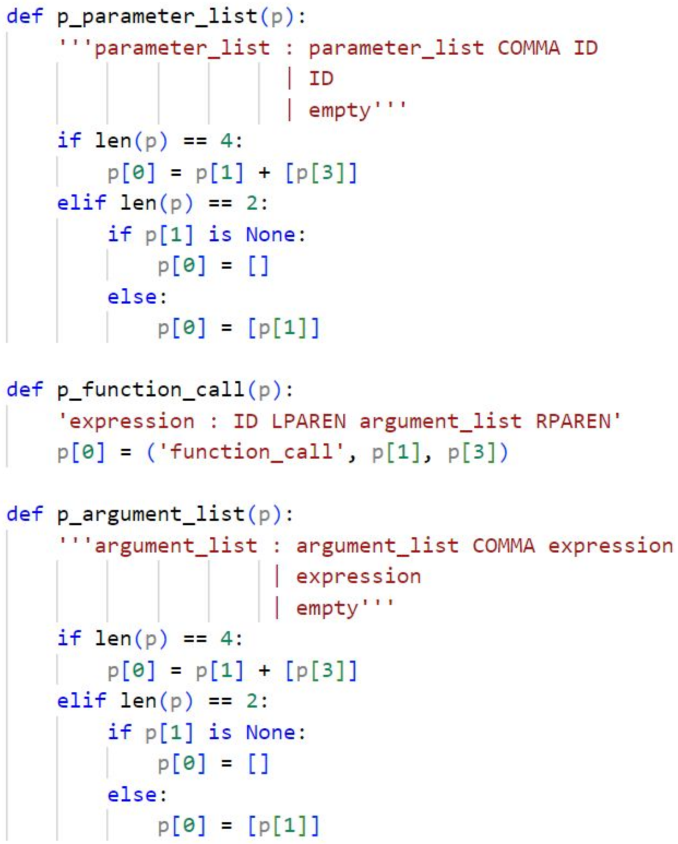
```python
def p_expression_string(p):
    'expression : STRING'
    p[0] = ('string', p[1])

def p_expression_bool(p):
    'expression : BOOL'
    p[0] = ('bool', p[1])

def p_statement_if_else(p):
    '''statement : AGAR LPAREN expression RPAREN TOH LBRACE statement_list RBRACE NAHITOH LBRACE statement_list RBRACE'''
    p[0] = ('if_else', p[3], p[7], p[11])

def p_statement_while(p):
    '''statement : JABTAK LPAREN expression RPAREN LBRACE statement_list RBRACE'''
    p[0] = ('while', p[3], p[6])

def p_statement_for(p):
    '''statement : BAARBAAR LPAREN statement expression SEMI assignment RPAREN LBRACE statement_list RBRACE
                 | BAARBAAR LPAREN statement expression SEMI assignment RPAREN LBRACE empty RBRACE'''
    if len(p) == 11:
        p[0] = ('for', p[3], p[4], p[6], p[9])
    else:
        p[0] = ('for', p[3], p[4], p[6], [])

def p_for_init(p):
    '''for_init : statement
                | declaration'''
    p[0] = p[1]
```

Syntax level error handling



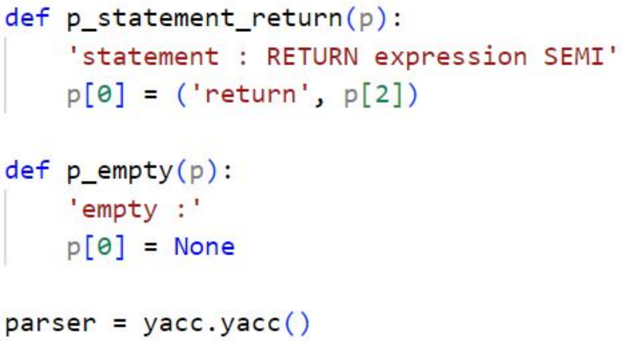
```python
def p_declaration(p):
    'declaration : ASSIGN ID ASSIGN_OP expression SEMI'
    p[0] = ('assign', p[2], p[4])

def p_expression_variable(p):
    'expression : ID'
    p[0] = ('var', p[1])

def p_error(p):
    if p:
        print(f"Syntax error at '{p.value}'")
    else:
        print("Syntax error at EOF")

def p_function_definition(p):
    'statement : FUNCTION ID LPAREN parameter_list RPAREN LBRACE statement_list RBRACE'
    p[0] = ('function_def', p[2], p[4], p[7])
```

Generating Parse Tree (parser.py)



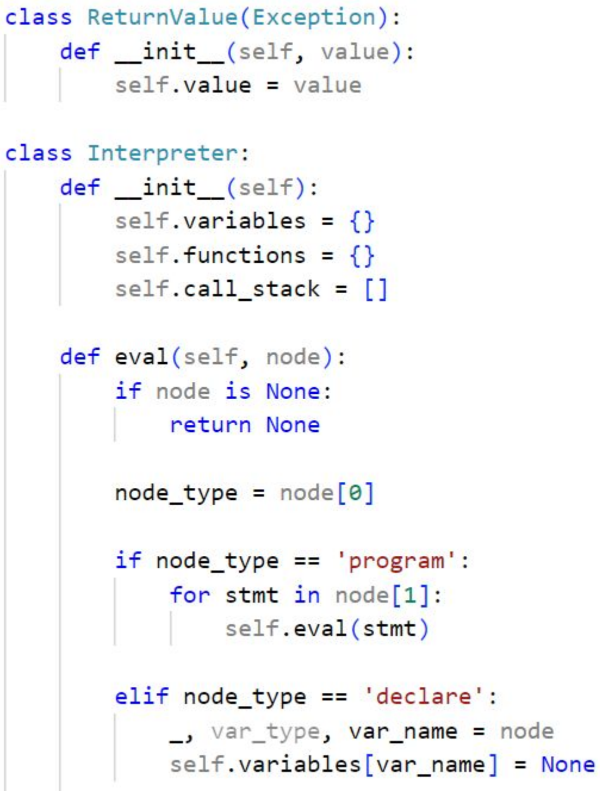
```python
def p_parameter_list(p):
    '''parameter_list : parameter_list COMMA ID
                      | ID
                      | empty'''
    if len(p) == 4:
        p[0] = p[1] + [p[3]]
    elif len(p) == 2:
        if p[1] is None:
            p[0] = []
        else:
            p[0] = [p[1]]

def p_function_call(p):
    'expression : ID LPAREN argument_list RPAREN'
    p[0] = ('function_call', p[1], p[3])

def p_argument_list(p):
    '''argument_list : argument_list COMMA expression
                     | expression
                     | empty'''
    if len(p) == 4:
        p[0] = p[1] + [p[3]]
    elif len(p) == 2:
        if p[1] is None:
            p[0] = []
        else:
            p[0] = [p[1]]
```

Generating Parse Tree (parser.py)



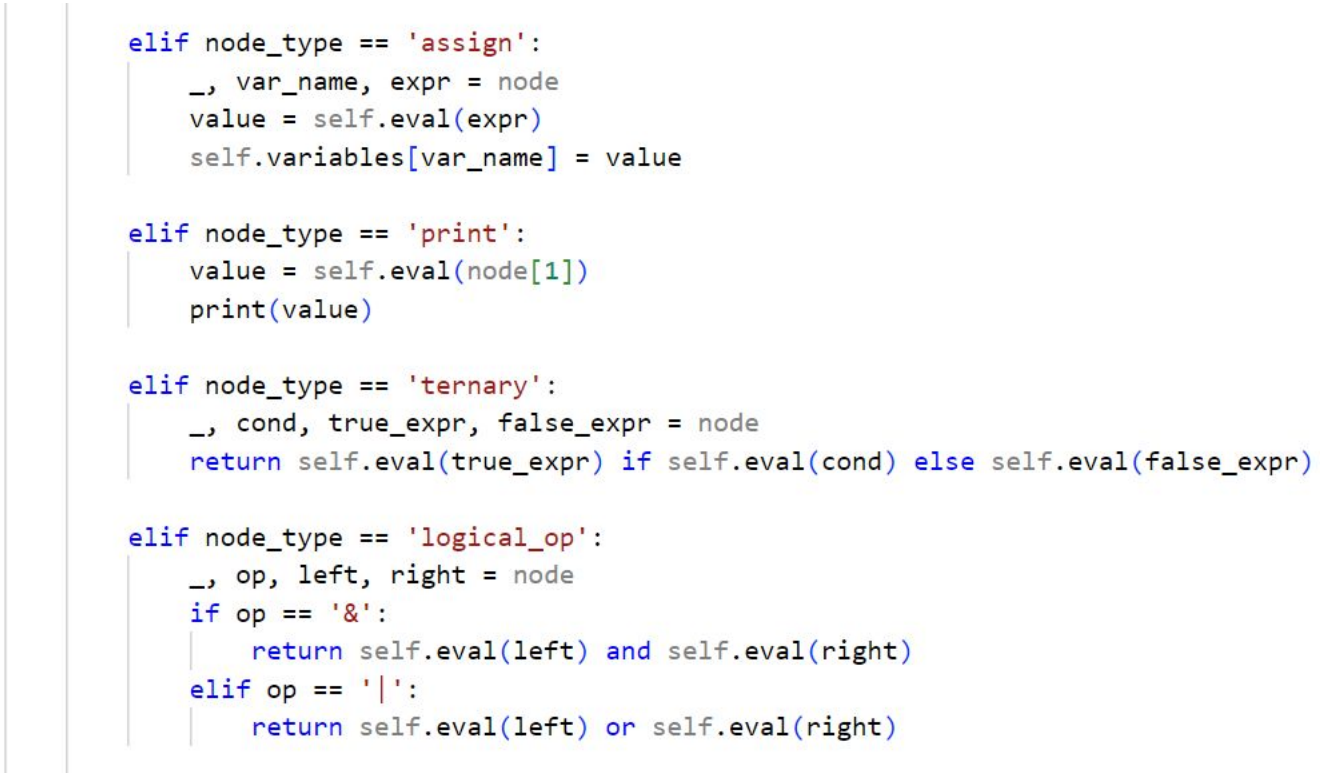
```python
def p_statement_return(p):
    'statement : RETURN expression SEMI'
    p[0] = ('return', p[2])

def p_empty(p):
    'empty :'
    p[0] = None

parser = yacc.yacc()
```

Interpreter (interpreter.py)



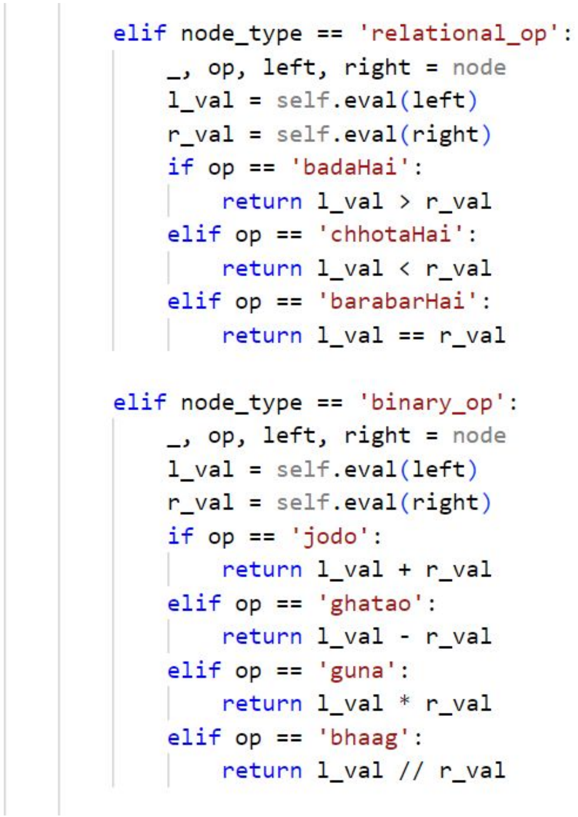
```python
class ReturnValue(Exception):
    def __init__(self, value):
        self.value = value

class Interpreter:
    def __init__(self):
        self.variables = {}
        self.functions = {}
        self.call_stack = []

    def eval(self, node):
        if node is None:
            return None

        node_type = node[0]

        if node_type == 'program':
            for stmt in node[1]:
                self.eval(stmt)

        elif node_type == 'declare':
            _, var_type, var_name = node
            self.variables[var_name] = None
```

Interpreter (interpreter.py)



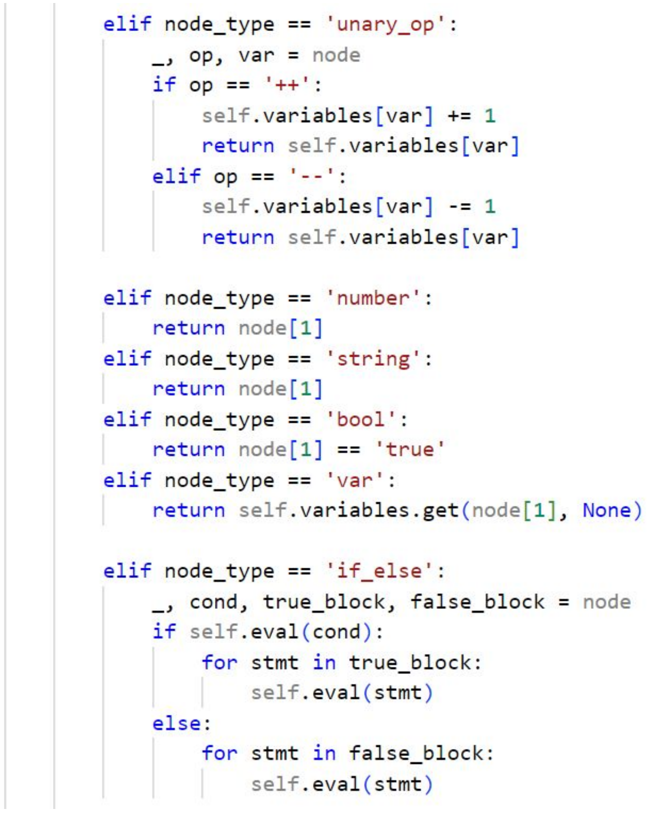
```python
        elif node_type == 'assign':
            _, var_name, expr = node
            value = self.eval(expr)
            self.variables[var_name] = value

        elif node_type == 'print':
            value = self.eval(node[1])
            print(value)

        elif node_type == 'ternary':
            _, cond, true_expr, false_expr = node
            return self.eval(true_expr) if self.eval(cond) else self.eval(false_expr)

        elif node_type == 'logical_op':
            _, op, left, right = node
            if op == '&':
                return self.eval(left) and self.eval(right)
            elif op == '|':
                return self.eval(left) or self.eval(right)
```

Interpreter (interpreter.py)



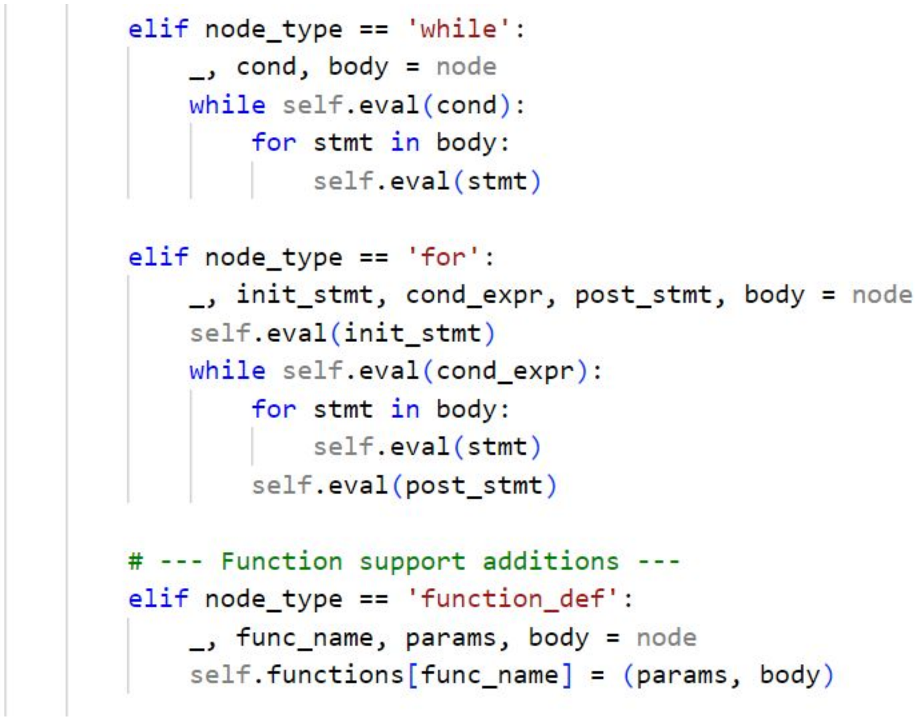
```
        elif node_type == 'relational_op':
            _, op, left, right = node
            l_val = self.eval(left)
            r_val = self.eval(right)
            if op == 'badaHai':
                return l_val > r_val
            elif op == 'chhotaHai':
                return l_val < r_val
            elif op == 'barabarHai':
                return l_val == r_val

        elif node_type == 'binary_op':
            _, op, left, right = node
            l_val = self.eval(left)
            r_val = self.eval(right)
            if op == 'jodo':
                return l_val + r_val
            elif op == 'ghatao':
                return l_val - r_val
            elif op == 'guna':
                return l_val * r_val
            elif op == 'bhaag':
                return l_val // r_val
```

Interpreter (interpreter.py)



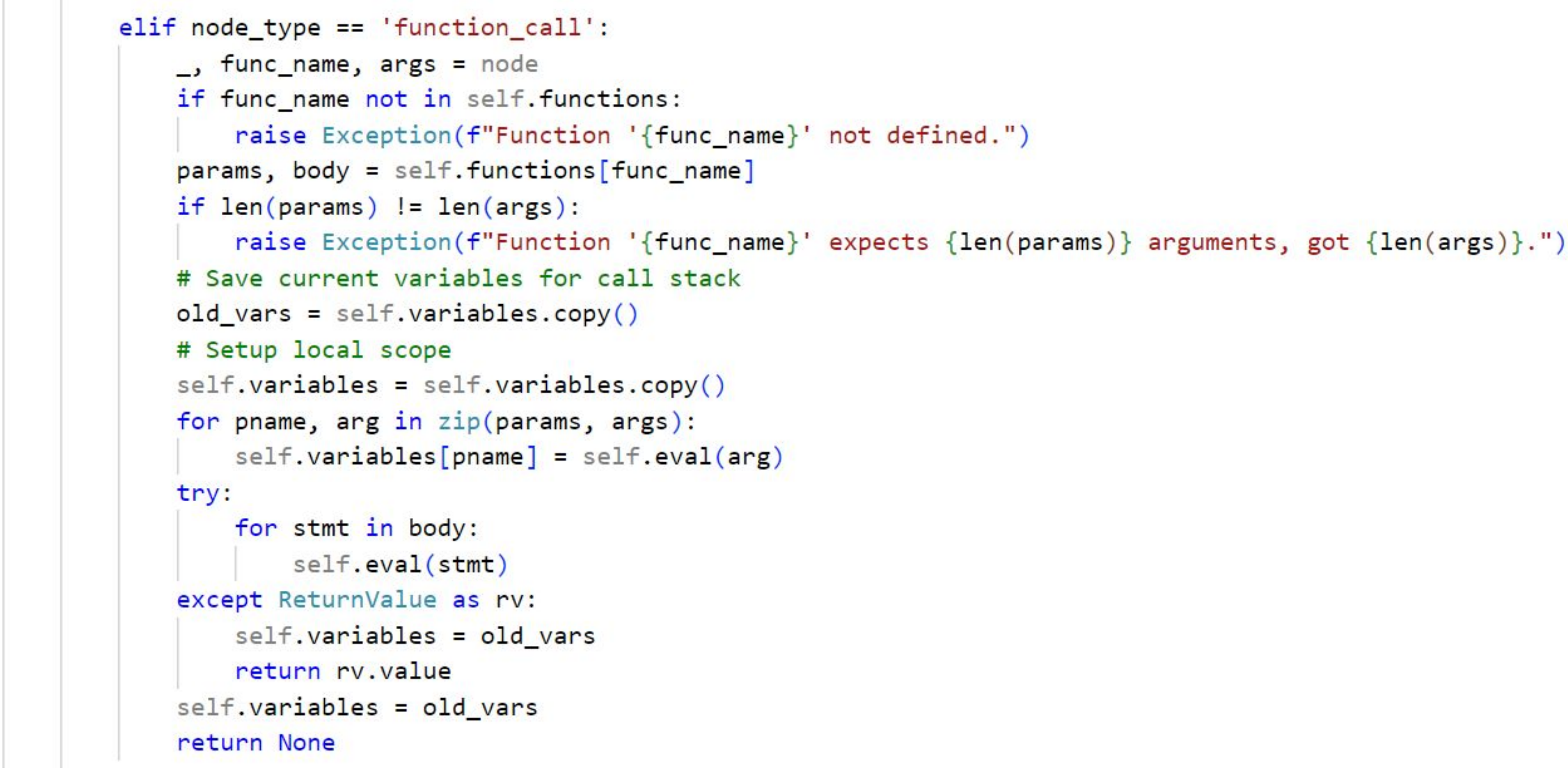
```python
        elif node_type == 'unary_op':
            _, op, var = node
            if op == '++':
                self.variables[var] += 1
                return self.variables[var]
            elif op == '--':
                self.variables[var] -= 1
                return self.variables[var]

        elif node_type == 'number':
            return node[1]
        elif node_type == 'string':
            return node[1]
        elif node_type == 'bool':
            return node[1] == 'true'
        elif node_type == 'var':
            return self.variables.get(node[1], None)

        elif node_type == 'if_else':
            _, cond, true_block, false_block = node
            if self.eval(cond):
                for stmt in true_block:
                    self.eval(stmt)
            else:
                for stmt in false_block:
                    self.eval(stmt)
```

Interpreter (interpreter.py)



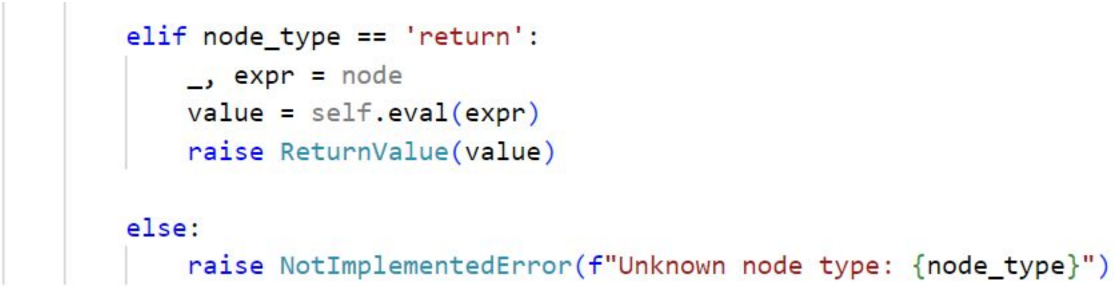
```python
        elif node_type == 'while':
            _, cond, body = node
            while self.eval(cond):
                for stmt in body:
                    self.eval(stmt)

        elif node_type == 'for':
            _, init_stmt, cond_expr, post_stmt, body = node
            self.eval(init_stmt)
            while self.eval(cond_expr):
                for stmt in body:
                    self.eval(stmt)
                self.eval(post_stmt)

        # --- Function support additions ---
        elif node_type == 'function_def':
            _, func_name, params, body = node
            self.functions[func_name] = (params, body)
```

Interpreter (interpreter.py)



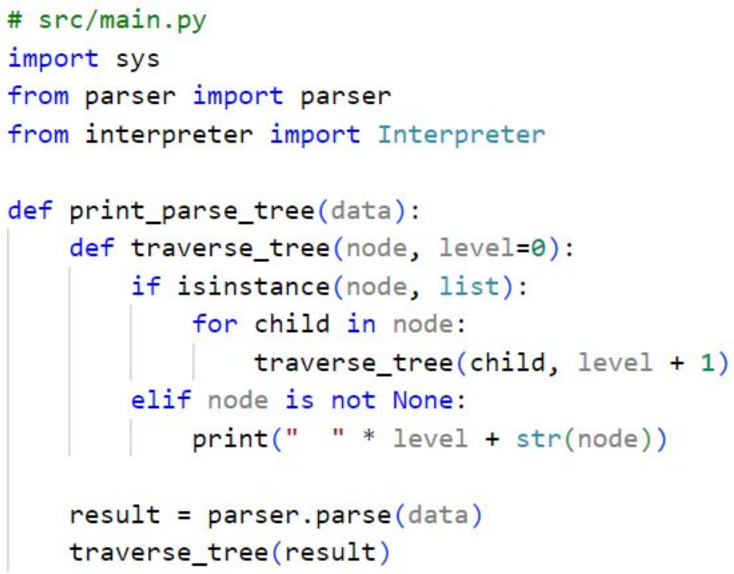
```python
        elif node_type == 'function_call':
            _, func_name, args = node
            if func_name not in self.functions:
                raise Exception(f"Function '{func_name}' not defined.")
            params, body = self.functions[func_name]
            if len(params) != len(args):
                raise Exception(f"Function '{func_name}' expects {len(params)} arguments, got {len(args)}.")
            # Save current variables for call stack
            old_vars = self.variables.copy()
            # Setup local scope
            self.variables = self.variables.copy()
            for pname, arg in zip(params, args):
                self.variables[pname] = self.eval(arg)
            try:
                for stmt in body:
                    self.eval(stmt)
            except ReturnValue as rv:
                self.variables = old_vars
                return rv.value
            self.variables = old_vars
            return None
```

Interpreter (interpreter.py)

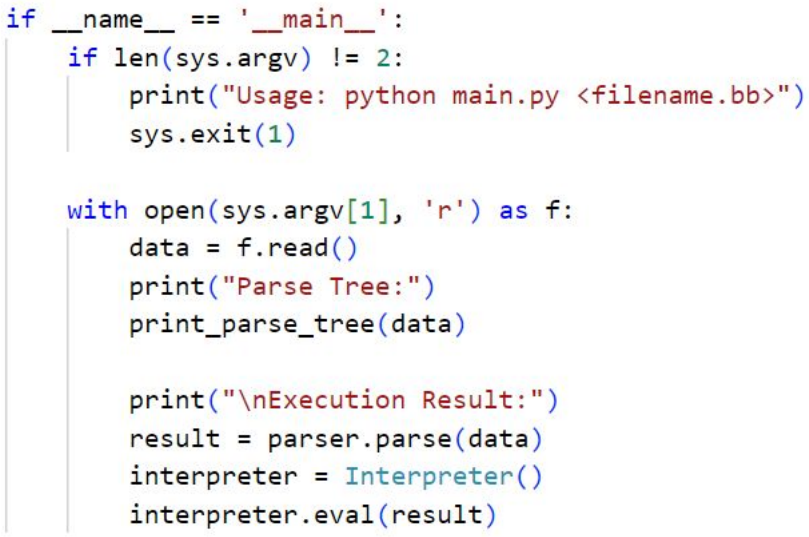


```
        elif node_type == 'return':
            _, expr = node
            value = self.eval(expr)
            raise ReturnValue(value)

        else:
            raise NotImplementedError(f"Unknown node type: {node_type}")
```

Main.py



```python
# src/main.py
import sys
from parser import parser
from interpreter import Interpreter

def print_parse_tree(data):
    def traverse_tree(node, level=0):
        if isinstance(node, list):
            for child in node:
                traverse_tree(child, level + 1)
        elif node is not None:
            print("  " * level + str(node))

    result = parser.parse(data)
    traverse_tree(result)
```



```python
if __name__ == '__main__':
    if len(sys.argv) != 2:
        print("Usage: python main.py <filename.bb>")
        sys.exit(1)

    with open(sys.argv[1], 'r') as f:
        data = f.read()
        print("Parse Tree:")
        print_parse_tree(data)

        print("\nExecution Result:")
        result = parser.parse(data)
        interpreter = Interpreter()
        interpreter.eval(result)
```

Instructions to Execute

- **Building** the required Python packages

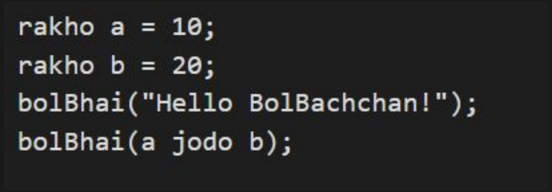
```
pip install -r requirements.txt
```

- **Run Program**

```
python src/main.py data/Sample1.bb
```

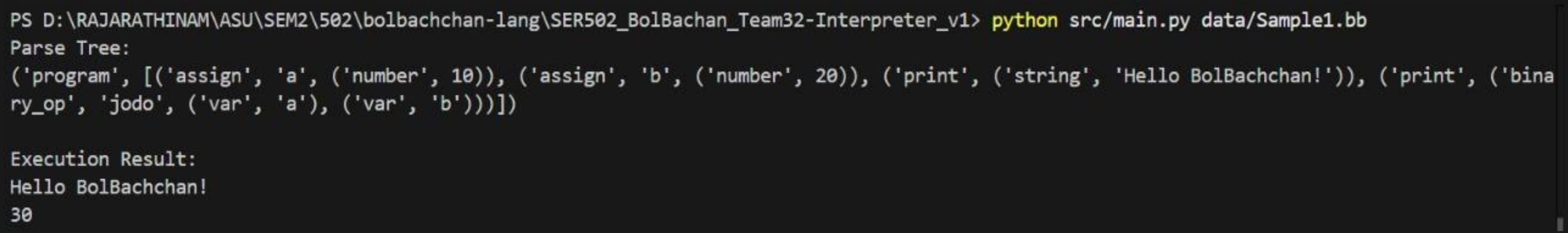
Sample Data 1 Snapshot

Input



```
rakho a = 10;
rakho b = 20;
bolBhai("Hello BolBachchan!");
bolBhai(a jodo b);
```

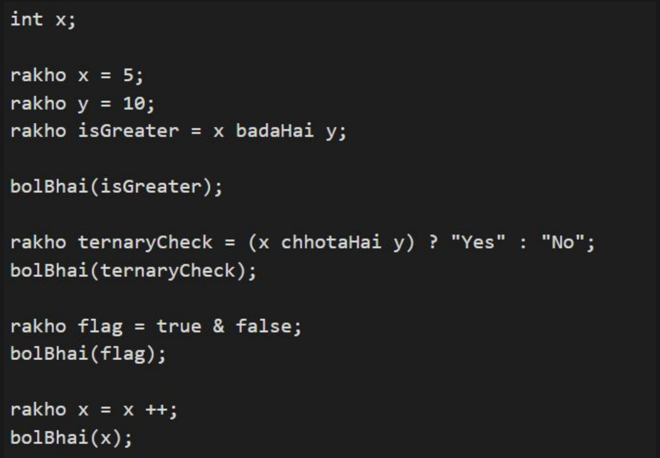
Output



```
PS D:\RAJARATHINAM\ASU\SEM2\502\bolbachchan-lang\SER502_BolBachan_Team32-Interpreter_v1> python src/main.py data/Sample1.bb
Parse Tree:
('program', [('assign', 'a', ('number', 10)), ('assign', 'b', ('number', 20)), ('print', ('string', 'Hello BolBachchan!')), ('print', ('bina
ry_op', 'jodo', ('var', 'a'), ('var', 'b')))])

Execution Result:
Hello BolBachchan!
30
```

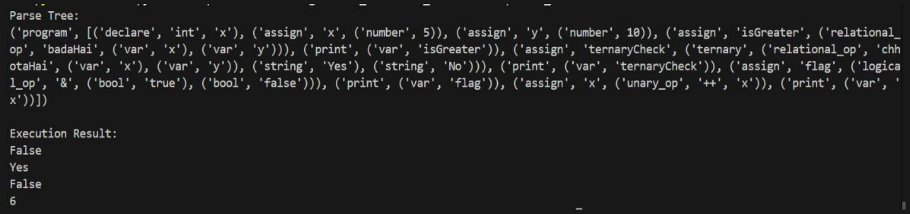
Sample Data 2 Snapshot

Input



```
int x;

rakho x = 5;
rakho y = 10;
rakho isGreater = x badaHai y;

bolBhai(isGreater);

rakho ternaryCheck = (x chhotaHai y) ? "Yes" : "No";
bolBhai(ternaryCheck);

rakho flag = true & false;
bolBhai(flag);

rakho x = x ++;
bolBhai(x);
```

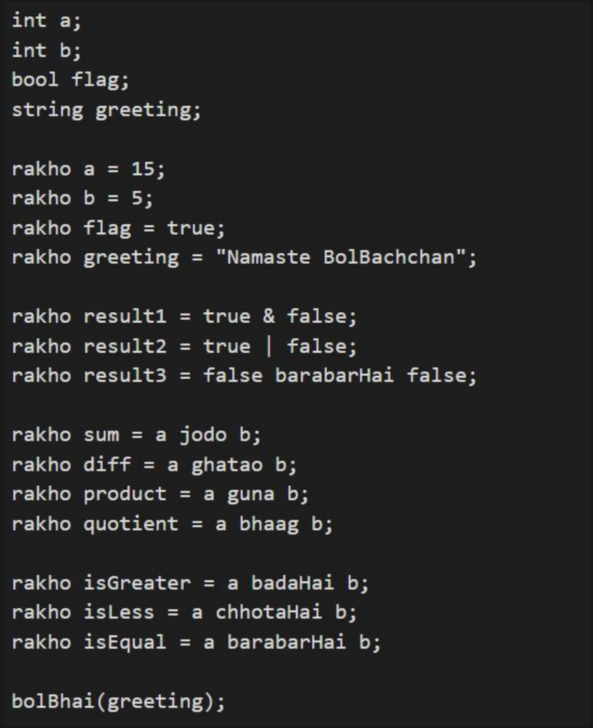
Output



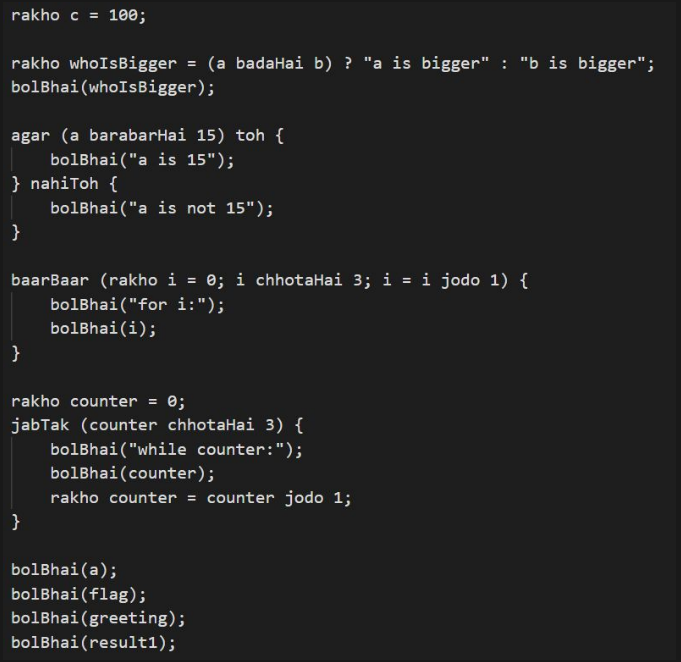
```
Parse Tree:
('program', [('declare', 'int', 'x'), ('assign', 'x', ('number', 5)), ('assign', 'y', ('number', 10)), ('assign', 'isGreater', ('relational_
op', 'badaHai', ('var', 'x'), ('var', 'y'))), ('print', ('var', 'isGreater')), ('assign', 'ternaryCheck', ('ternary', ('relational_op', 'chh
otaHai', ('var', 'x'), ('var', 'y')), ('string', 'Yes'), ('string', 'No'))), ('print', ('var', 'ternaryCheck')), ('assign', 'flag', ('logica
l_op', '&', ('bool', 'true'), ('bool', 'false'))), ('print', ('var', 'flag')), ('assign', 'x', ('unary_op', '++', 'x')), ('print', ('var', '
x'))])

Execution Result:
False
Yes
False
6
```

Sample Data 3 Snapshot

Input



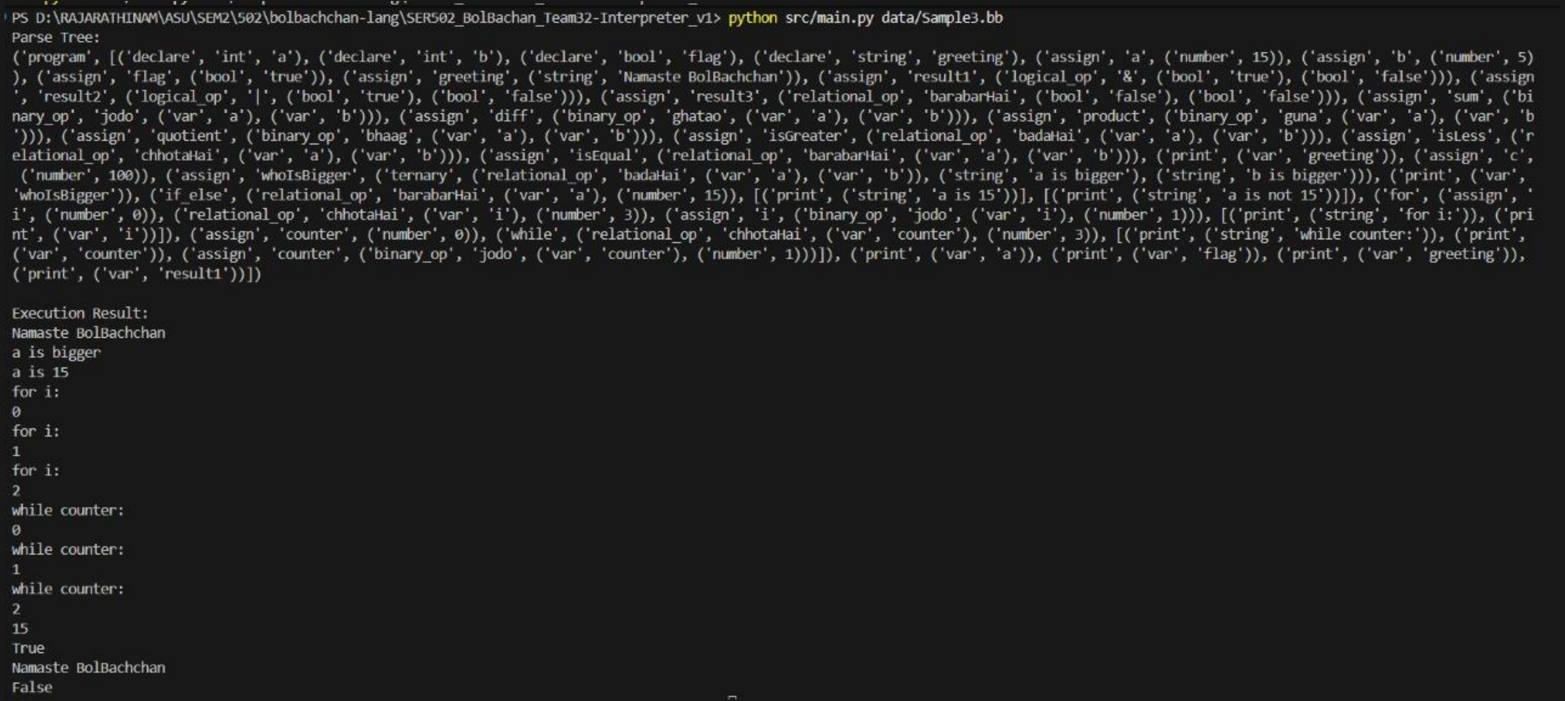
```
int a;
int b;
bool flag;
string greeting;

rakho a = 15;
rakho b = 5;
rakho flag = true;
rakho greeting = "Namaste BolBachchan";

rakho result1 = true & false;
rakho result2 = true | false;
rakho result3 = false barabarHai false;

rakho sum = a jodo b;
rakho diff = a ghatao b;
rakho product = a guna b;
rakho quotient = a bhaag b;

rakho isGreater = a badaHai b;
rakho isLess = a chhotaHai b;
rakho isEqual = a barabarHai b;

bolBhai(greeting);
```



```
rakho c = 100;

rakho whoIsBigger = (a badaHai b) ? "a is bigger" : "b is bigger";
bolBhai(whoIsBigger);

agar (a barabarHai 15) toh {
    bolBhai("a is 15");
} nahiToh {
    bolBhai("a is not 15");
}

baarBaar (rakho i = 0; i chhotaHai 3; i = i jodo 1) {
    bolBhai("for i:");
    bolBhai(i);
}

rakho counter = 0;
jabTak (counter chhotaHai 3) {
    bolBhai("while counter:");
    bolBhai(counter);
    rakho counter = counter jodo 1;
}

bolBhai(a);
bolBhai(flag);
bolBhai(greeting);
bolBhai(result1);
```

Sample Data 3 Snapshot

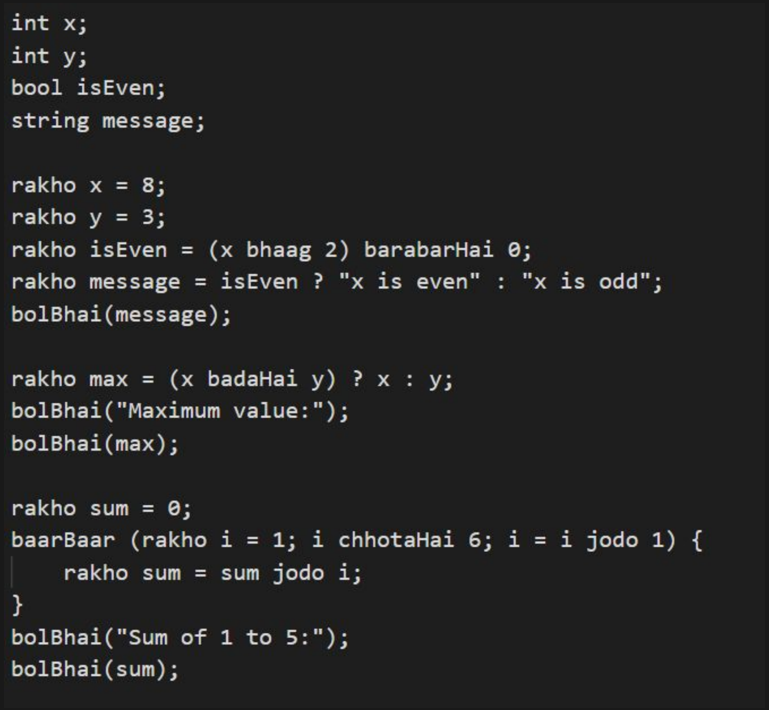
Output



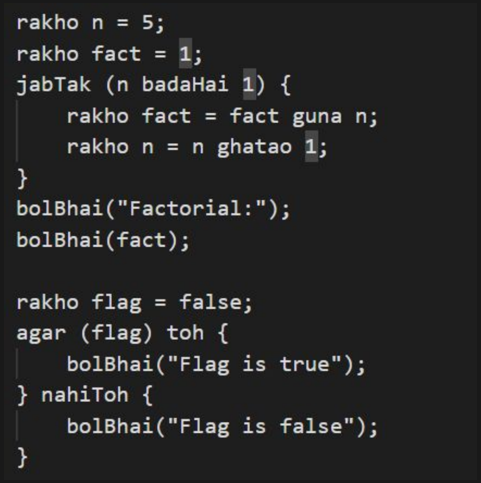
```
PS D:\RAJARATHINAM\ASU\SEM2\502\bolbachchan-lang\SER502_BolBachan_Team32-Interpreter_v1> python src/main.py data/Sample3.bb
Parse Tree:
('program', [('declare', 'int', 'a'), ('declare', 'int', 'b'), ('declare', 'bool', 'flag'), ('declare', 'string', 'greeting'), ('assign', 'a', ('number', 15)), ('assign', 'b', ('number', 5)
), ('assign', 'flag', ('bool', 'true')), ('assign', 'greeting', ('string', 'Namaste BolBachchan')), ('assign', 'result1', ('logical_op', '&', ('bool', 'true'), ('bool', 'false'))), ('assign
', 'result2', ('logical_op', '|', ('bool', 'true'), ('bool', 'false'))), ('assign', 'result3', ('relational_op', 'barabarHai', ('bool', 'false'), ('bool', 'false'))), ('assign', 'sum', ('bi
nary_op', 'jodo', ('var', 'a'), ('var', 'b'))), ('assign', 'diff', ('binary_op', 'ghatao', ('var', 'a'), ('var', 'b'))), ('assign', 'product', ('binary_op', 'guna', ('var', 'a'), ('var', 'b
'))), ('assign', 'quotient', ('binary_op', 'bhaag', ('var', 'a'), ('var', 'b'))), ('assign', 'isGreater', ('relational_op', 'badaHai', ('var', 'a'), ('var', 'b'))), ('assign', 'isLess', ('r
elational_op', 'chhotaHai', ('var', 'a'), ('var', 'b'))), ('assign', 'isEqual', ('relational_op', 'barabarHai', ('var', 'a'), ('var', 'b'))), ('print', ('var', 'greeting')), ('assign', 'c',
 ('number', 100)), ('assign', 'whoIsBigger', ('ternary', ('relational_op', 'badaHai', ('var', 'a'), ('var', 'b')), ('string', 'a is bigger'), ('string', 'b is bigger'))), ('print', ('var',
'whoIsBigger')), ('if_else', ('relational_op', 'barabarHai', ('var', 'a'), ('number', 15)), [('print', ('string', 'a is 15'))], [('print', ('string', 'a is not 15'))]), ('for', ('assign', '
i', ('number', 0)), ('relational_op', 'chhotaHai', ('var', 'i'), ('number', 3)), ('assign', 'i', ('binary_op', 'jodo', ('var', 'i'), ('number', 1))), [('print', ('string', 'for i:')), ('pri
nt', ('var', 'i'))]), ('assign', 'counter', ('number', 0)), ('while', ('relational_op', 'chhotaHai', ('var', 'counter'), ('number', 3)), [('print', ('string', 'while counter:')), ('print',
('var', 'counter')), ('assign', 'counter', ('binary_op', 'jodo', ('var', 'counter'), ('number', 1)))]), ('print', ('var', 'a')), ('print', ('var', 'flag')), ('print', ('var', 'greeting')),
('print', ('var', 'result1'))])

Execution Result:
Namaste BolBachchan
a is bigger
a is 15
for i:
0
for i:
1
for i:
2
while counter:
0
while counter:
1
while counter:
2
15
True
Namaste BolBachchan
False
```

Sample Data 4 Snapshot

Input



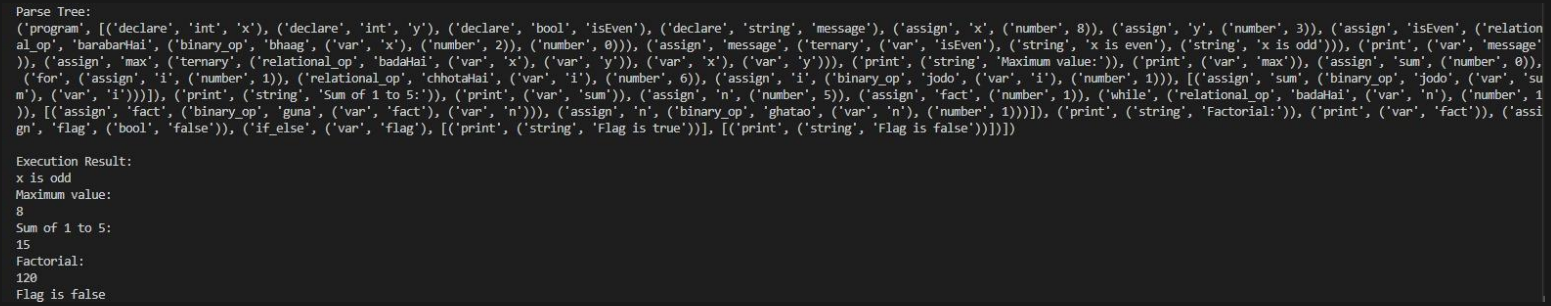
```
int x;
int y;
bool isEven;
string message;

rakho x = 8;
rakho y = 3;
rakho isEven = (x bhaag 2) barabarHai 0;
rakho message = isEven ? "x is even" : "x is odd";
bolBhai(message);

rakho max = (x badaHai y) ? x : y;
bolBhai("Maximum value:");
bolBhai(max);

rakho sum = 0;
baarBaar (rakho i = 1; i chhotaHai 6; i = i jodo 1) {
    rakho sum = sum jodo i;
}
bolBhai("Sum of 1 to 5:");
bolBhai(sum);
```



```
rakho n = 5;
rakho fact = 1;
jabTak (n badaHai 1) {
    rakho fact = fact guna n;
    rakho n = n ghatao 1;
}
bolBhai("Factorial:");
bolBhai(fact);

rakho flag = false;
agar (flag) toh {
    bolBhai("Flag is true");
} nahiToh {
    bolBhai("Flag is false");
}
```

Sample Data 4 Snapshot

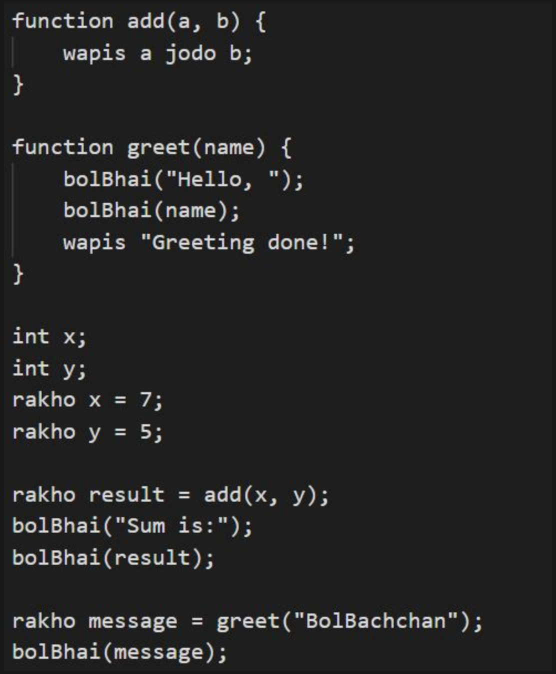
Output



```
Parse Tree:
('program', [('declare', 'int', 'x'), ('declare', 'int', 'y'), ('declare', 'bool', 'isEven'), ('declare', 'string', 'message'), ('assign', 'x', ('number', 8)), ('assign', 'y', ('number', 3)), ('assign', 'isEven', ('relation
al_op', 'barabarHai', ('binary_op', 'bhaag', ('var', 'x'), ('number', 2)), ('number', 0))), ('assign', 'message', ('ternary', ('var', 'isEven'), ('string', 'x is even'), ('string', 'x is odd'))), ('print', ('var', 'message'
)), ('assign', 'max', ('ternary', ('relational_op', 'badaHai', ('var', 'x'), ('var', 'y')), ('var', 'x'), ('var', 'y'))), ('print', ('string', 'Maximum value:')), ('print', ('var', 'max')), ('assign', 'sum', ('number', 0)),
 ('for', ('assign', 'i', ('number', 1)), ('relational_op', 'chhotaHai', ('var', 'i'), ('number', 6)), ('assign', 'i', ('binary_op', 'jodo', ('var', 'i'), ('number', 1))), [('assign', 'sum', ('binary_op', 'jodo', ('var', 'su
m'), ('var', 'i')))]), ('print', ('string', 'Sum of 1 to 5:')), ('print', ('var', 'sum')), ('assign', 'n', ('number', 5)), ('assign', 'fact', ('number', 1)), ('while', ('relational_op', 'badaHai', ('var', 'n'), ('number', 1
)), [('assign', 'fact', ('binary_op', 'guna', ('var', 'fact'), ('var', 'n'))), ('assign', 'n', ('binary_op', 'ghatao', ('var', 'n'), ('number', 1)))]), ('print', ('string', 'Factorial:')), ('print', ('var', 'fact')), ('assi
gn', 'flag', ('bool', 'false')), ('if_else', ('var', 'flag'), [('print', ('string', 'Flag is true'))], [('print', ('string', 'Flag is false'))])])

Execution Result:
x is odd
Maximum value:
8
Sum of 1 to 5:
15
Factorial:
120
Flag is false
```

Sample Data 5 Snapshot

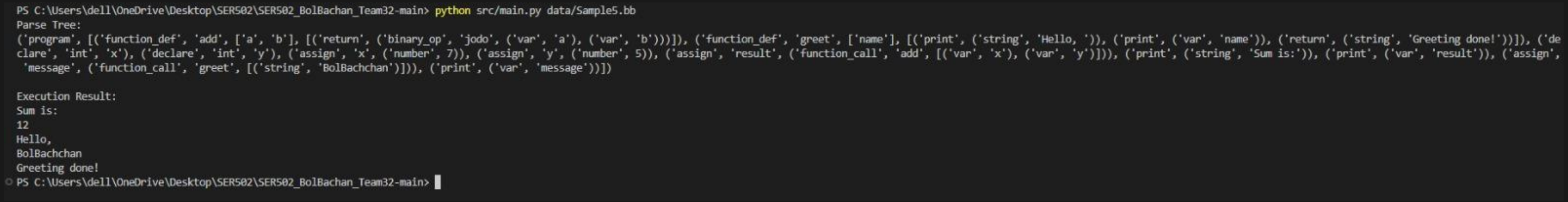
Input



```
function add(a, b) {
    wapis a jodo b;
}

function greet(name) {
    bolBhai("Hello, ");
    bolBhai(name);
    wapis "Greeting done!";
}

int x;
int y;
rakho x = 7;
rakho y = 5;

rakho result = add(x, y);
bolBhai("Sum is:");
bolBhai(result);

rakho message = greet("BolBachchan");
bolBhai(message);
```

Sample Data 5 Snapshot

Output



```
PS C:\Users\dell\OneDrive\Desktop\SER502\SER502_BolBachan_Team32-main> python src/main.py data/Sample5.bb
Parse Tree:
('program', [('function_def', 'add', ['a', 'b'], [('return', ('binary_op', 'jodo', ('var', 'a'), ('var', 'b')))]), ('function_def', 'greet', ['name'], [('print', ('string', 'Hello, ')), ('print', ('var', 'name')), ('return', ('string', 'Greeting done!'))]), ('de
clare', 'int', 'x'), ('declare', 'int', 'y'), ('assign', 'x', ('number', 7)), ('assign', 'y', ('number', 5)), ('assign', 'result', ('function_call', 'add', [('var', 'x'), ('var', 'y')])), ('print', ('string', 'Sum is:')), ('print', ('var', 'result')), ('assign',
 'message', ('function_call', 'greet', [('string', 'BolBachchan')])), ('print', ('var', 'message'))])

Execution Result:
Sum is:
12
Hello,
BolBachchan
Greeting done!
PS C:\Users\dell\OneDrive\Desktop\SER502\SER502_BolBachan_Team32-main> 
```

ASU®

Arizona State
University