Prateek P
PES1UG23AM211

Automata Formal Language and Logic

```
# lexer.py
import ply.lex as lex

# token list
tokens = [
    'NUMBER',
    'ID',
    'EQUALS',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
    'LBRACE',
    'RBRACE',
    'SEMICOLON',
    'COMMA',
    'GT',
    'LT',
    'GE',
    'LE',
    'EQ',
    'NE'
]

# reserved words
reserved = {
    'if': 'IF',
    'else': 'ELSE',
    'while': 'WHILE',
    'for': 'FOR',
    'int': 'INT',
    'float': 'FLOAT',
    'print': 'PRINT'
}

tokens = tokens + list(reserved.values())

# regex rules
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_LPAREN = r'\('
t_RPAREN = r'\)'
```

Prateek P
PES1UG23AM211

```
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_SEMICOLON = r';'
t_COMMA = r','
t_GT = r'>'
t_LT = r'<'
t_GE = r'>='
t_LE = r'<='
t_EQ = r'=='
t_NE = r'!='

# complex tokens
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'ID')
    return t

# ignored characters
t_ignore = ' \t'    # this is supposed to be tab - check later

# new line
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# for some error
def t_error(t):
    print(f"Illegal character '{t.value[0]}' at line {t.lexer.lineno}")
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

```
# parser_rules.py
import ply.yacc as yacc
from lexer import tokens, lexer

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

Prateek P
PES1UG23AM211

```
    def evaluate_expression(self, expr):
        if isinstance(expr, tuple):
            if expr[0] == 'binop':
                left = self.evaluate_expression(expr[1])
                right = self.evaluate_expression(expr[3])
                op = expr[2]
                if op == '+': return left + right
                if op == '-': return left - right
                if op == '*': return left * right
                if op == '/':
                    if right == 0:
                        print("Error: Division by zero")
                        self.had_error = True
                        return 0
                    return left / right
            elif expr[0] == 'var':
                if expr[1] not in self.variables:
                    print(f"Error: Variable '{expr[1]}' not declared")
                    self.had_error = True
                    return 0
                return self.variables.get(expr[1], 0)
            elif expr[0] == 'number':
                return expr[1]
        return expr

    def evaluate_condition(self, condition):
        if not condition or condition[0] != 'condition':
            return False

        left = self.evaluate_expression(condition[2])
        right = self.evaluate_expression(condition[3])
        op = condition[1]

        if op == '>': return left > right
        if op == '<': return left < right
        if op == '>=': return left >= right
        if op == '<=': return left <= right
        if op == '==': return left == right
        if op == '!=': return left != right
        return False

    def execute(self, node):
        if not node:
            return

        if isinstance(node, list):
            for statement in node:
                self.execute(statement)
```

Prateek P
PES1UG23AM211

```
            return

        if node[0] == 'declaration':
            for var in node[2]:
                self.variables[var] = None

        elif node[0] == 'assignment':
            var_name = node[1]
            if var_name not in self.variables:
                print(f"Error: Variable '{var_name}' not declared")
                self.had_error = True
            else:
                self.variables[var_name] = self.evaluate_expression(node[2])

        elif node[0] == 'if':
            condition_result = self.evaluate_condition(node[1])
            if condition_result:
                self.execute(node[2])
            elif len(node) > 3:  # has else branch
                self.execute(node[3])

        elif node[0] == 'while':
            while self.evaluate_condition(node[1]):
                self.execute(node[2])

        elif node[0] == 'print':
            values = []
            for expr in node[1]:
                values.append(self.evaluate_expression(expr))
            print(*values)

# Dictionary to store variables
interpreter = Interpreter()

def p_program(p):
    '''program : statement
               | program statement'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[2]]

def p_statement(p):
    '''statement : declaration
                 | assignment
                 | if_statement
                 | while_statement
                 | for_statement
                 | print_statement'''
```

Prateek P
PES1UG23AM211

```
    p[0] = p[1]

def p_declaration(p):
    '''declaration : type declaration_list SEMICOLON'''
    p[0] = ('declaration', p[1], p[2])

def p_declaration_list(p):
    '''declaration_list : ID
                        | ID COMMA declaration_list'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = [p[1]] + p[3]

def p_type(p):
    '''type : INT
            | FLOAT'''
    p[0] = p[1]

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

def p_if_statement(p):
    '''if_statement : IF LPAREN condition RPAREN LBRACE program RBRACE
                    | IF LPAREN condition RPAREN LBRACE program RBRACE ELSE
LBRACE program RBRACE'''
    if len(p) == 8:
        p[0] = ('if', p[3], p[6])
    else:
        p[0] = ('if', p[3], p[6], p[10])

def p_while_statement(p):
    '''while_statement : WHILE LPAREN condition RPAREN LBRACE program
RBRACE'''
    p[0] = ('while', p[3], p[6])

def p_for_statement(p):
    '''for_statement : FOR LPAREN assignment condition SEMICOLON ID EQUALS
expression RPAREN LBRACE program RBRACE'''
    p[0] = ('for', p[3], p[4], (p[6], p[8]), p[11])

def p_print_statement(p):
    '''print_statement : PRINT LPAREN print_list RPAREN SEMICOLON'''
    p[0] = ('print', p[3])

def p_print_list(p):
    '''print_list : expression
                  | expression COMMA print_list'''
```

Prateek P
PES1UG23AM211

```
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = [p[1]] + p[3]

def p_condition(p):
    '''condition : expression GT expression
                 | expression LT expression
                 | expression GE expression
                 | expression LE expression
                 | expression EQ expression
                 | expression NE expression'''
    p[0] = ('condition', p[2], p[1], p[3])

def p_expression(p):
    '''expression : term
                  | expression PLUS term
                  | expression MINUS term'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = ('binop', p[1], p[2], p[3])

def p_term(p):
    '''term : factor
            | term TIMES factor
            | term DIVIDE factor'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = ('binop', p[1], p[2], p[3])

def p_factor(p):
    '''factor : NUMBER
              | ID
              | LPAREN expression RPAREN'''
    if len(p) == 2:
        if isinstance(p[1], (int, float)):
            p[0] = ('number', p[1])
        else:
            p[0] = ('var', p[1])
    else:
        p[0] = p[2]

def p_error(p):
    if p:
        print(f"Syntax error at line {p.lineno}, position {p.lexpos}:
Unexpected token '{p.value}'")
    else:
```

Prateek P
PES1UG23AM211

```
        print("Syntax error: Unexpected end of input")

# Build the parser
parser = yacc.yacc()
```

```
# main_prog.py
from parser_rules import parser, lexer, interpreter

def main():
    print("Enter your code (type 'end' on a new line to finish):")
    code_lines = []
    while True:
        try:
            line = input('> ')
            if line.strip() == 'end':
                break
            code_lines.append(line)
        except EOFError:
            break

    code = '\n'.join(code_lines)
    if code.strip():
        result = parser.parse(code, lexer=lexer)
        if result is not None:
            interpreter.execute(result)
            if not interpreter.had_error:
                print("Parsing successful!")
            else:
                print("Execution completed with errors.")
        else:
            print("Parsing failed.")

if __name__ == "__main__":
    main()
```

Prateek P
PES1UG23AM211

```
Microsoft Windows [Version 10.0.22631.4317]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Prateek\vsCode Projects\Sem3>cd AFLL\try2

C:\Users\Prateek\vsCode Projects\Sem3\AFLL\try2>python main_prog.py
Enter your code (type 'end' on a new line to finish):
> int a, b, result;
> a = 10;
> b = 5;
> result = a + b;
> print(result);
> end
15
Parsing successful!

C:\Users\Prateek\vsCode Projects\Sem3\AFLL\try2>python main_prog.py
Enter your code (type 'end' on a new line to finish):
> int x, y, z;
> x = 20;
> y = 4;
> z = (x + y) * 2 - y / 2;
> print(z);
> end
46.0
Parsing successful!

C:\Users\Prateek\vsCode Projects\Sem3\AFLL\try2>python main_prog.py
Enter your code (type 'end' on a new line to finish):
> int age;
> age = 18;
> if (age >= 18) {
>     print(1);
> } else {
>     print(0);
> }
> end
1
Parsing successful!
```

Prateek P
PES1UG23AM211

```
C:\Users\Prateek\vsCode Projects\Sem3\AFLL\try2>python main_prog.py
Enter your code (type 'end' on a new line to finish):
> int x;
> x = 5;
> y = 10;
> print(x, y);
> end
Error: Variable 'y' not declared
Error: Variable 'y' not declared
5 0
Execution completed with errors.

C:\Users\Prateek\vsCode Projects\Sem3\AFLL\try2>python main_prog.py
Enter your code (type 'end' on a new line to finish):
> int a, b;
> a = 10;
> b = 0;
> print(a / b);
> end
Error: Division by zero
0
Execution completed with errors.

C:\Users\Prateek\vsCode Projects\Sem3\AFLL\try2>python main_prog.py
Enter your code (type 'end' on a new line to finish):
> int prev, current, next, counter;
> prev = 0;
> current = 1;
> counter = 0;
> while (counter < 5) {
>     print(current);
>     next = prev + current;
>     prev = current;
>     current = next;
>     counter = counter + 1;
> }
> end
1
1
2
3
5
Parsing successful!
```