Automata Formal Languages and Logic
UE23CS242A

Project Title:Implementation of lexer and parser for javascript

Team Members:

Manoj R - PES2UG23CS328

SECTION:E

Lexer and parser code(function):

```
import ply.lex as lex
import ply.yacc as yacc

# Lexer definition
tokens = (
    'FUNCTION',
    'RETURN',
    'IDENTIFIER',
    'LPAREN',
    'RPAREN',
    'LCURLY',
    'RCURLY',
    'NUMBER',
    'STRING',
    'SEMICOLON',
    'COMMA',
    'ASSIGN',
    'PLUS',
    'STAR',
    'DOT',
)

# Token specifications
t_FUNCTION = r'function'
t_RETURN = r'return'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LCURLY = r'\{'
t_RCURLY = r'\}'
t_SEMICOLON = r';'
t_COMMA = r','
t_ASSIGN = r'='
t_PLUS = r'\+'
t_STAR = r'\*'
t_DOT = r'\.'
t_NUMBER = r'\d+'
t_STRING = r'\".*?\"|\'[^\']*\''
```

```
t_STRING = r'\".*?\"|\'[^\']*\''
t_ignore = ' \t'

def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    if t.value == 'function':
        t.type = 'FUNCTION'
    elif t.value == 'return':
        t.type = 'RETURN'
    return t

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

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

# Build the lexer
lexer = lex.lex()

# Parser definition
precedence = (
    ('left', 'PLUS'),
    ('left', 'STAR'),
)

def p_program(p):
    '''program : function_declaration
               | program function_declaration'''
    if len(p) == 2:
        p[0] = {'type': 'program', 'functions': [p[1]]}
    else:
        p[0] = {'type': 'program', 'functions': p[1]['functions'] + [p[2]]}
```

```
def p_function_declaration(p):
    '''function_declaration : FUNCTION IDENTIFIER LPAREN params RPAREN LCURLY statements RCURLY'''
    p[0] = {
        'type': 'function',
        'name': p[2],
        'params': p[4],
        'body': p[7]
    }

def p_params(p):
    '''params : IDENTIFIER
              | params COMMA IDENTIFIER
              | '''
    if len(p) == 2:
        p[0] = [p[1]]
    elif len(p) == 4:
        p[0] = p[1] + [p[3]]
    else:
        p[0] = []

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

def p_statement(p):
    '''statement : assignment_statement SEMICOLON
                 | function_call_statement SEMICOLON
                 | return_statement SEMICOLON'''
    p[0] = p[1]
```

```
def p_assignment_statement(p):
    '''assignment_statement : IDENTIFIER ASSIGN expression'''
    p[0] = {
        'type': 'assignment',
        'variable': p[1],
        'value': p[3]
    }

def p_return_statement(p):
    '''return_statement : RETURN expression'''
    p[0] = {
        'type': 'return',
        'value': p[2]
    }

def p_function_call_statement(p):
    '''function_call_statement : IDENTIFIER LPAREN args RPAREN
                               | IDENTIFIER DOT IDENTIFIER LPAREN args RPAREN'''
    if len(p) == 5:
        p[0] = {
            'type': 'function_call',
            'function': p[1],
            'args': p[4]
        }
    else:
        p[0] = {
            'type': 'method_call',
            'object': p[1],
            'method': p[3],
            'args': p[5]
        }
```

```
def p_expression(p):
    '''expression : NUMBER
                  | STRING
                  | IDENTIFIER
                  | expression PLUS expression
                  | expression STAR expression'''
    if len(p) == 4:
        p[0] = {
            'type': 'binary_operation',
            'operator': p[2],
            'left': p[1],
            'right': p[3]
        }
    else:
        p[0] = {
            'type': 'expression',
            'value': p[1]
        }

def p_args(p):
    '''args : expression
            | args COMMA expression
            | '''
    if len(p) == 2:
        p[0] = [p[1]]
    elif len(p) == 4:
        p[0] = p[1] + [p[3]]
    else:
        p[0] = []
```

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

# Build the parser
parser = yacc.yacc()

# Example input for testing
input_data = """
function add(x, y) {
    return x + y;
}

function greet(name) {
    console.log("Hello, " + name);
}

function multiply(a, b) {
    return a * b;
}
"""

# Parse the input and determine validity
result = parser.parse(input_data)

if result:
    print("Valid syntax!")
else:
    print("Invalid syntax.")
```

Output Screenshot :

Valid syntax:

```
input_data = """
function add(x, y) {
    return x + y;
}

function greet(name) {
    console.log("Hello, " + name);
}

function multiply(a, b) {
    return a * b;
}
"""
```

```
Valid syntax!
```

Invalid syntax:

```
input_data = """
function add(x, y) {
    return x  y;
}

function greet(name) {
    console.log("Hello, " + name);


function multiply(a, b) {
    return a * b;
}
"""
```

```
Syntax error at token 'y' on line 3.
Syntax error at token 'function' on line 10.
Invalid syntax.
```

Lexer and parser code(do while):

```
import ply.yacc as yacc
import ply.lex as lex

# Lexer definition
tokens = (
    'DO',
    'WHILE',
    'LPAREN',
    'RPAREN',
    'LCURLY',
    'RCURLY',
    'ID',
    'ASSIGN',
    'NUMBER',
    'LT',
    'SEMICOLON'
)

# Token specifications
t_WHILE = r'while'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LCURLY = r'\{'
t_RCURLY = r'\}'
t_ASSIGN = r'='
t_LT = r'<'
t_SEMICOLON = r';'
```

```python
# Identifier token
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    # Check if it's a keyword
    if t.value == 'do':
        t.type = 'DO'
    elif t.value == 'while':
        t.type = 'WHILE'
    return t

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

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

# Ignore spaces and tabs
t_ignore = ' \t'

# Error handling rule
def t_error(t):
    print(f"Illegal character '{t.value[0]}'")
    t.lexer.skip(1)

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

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

# Parsing rules
def p_do_while_loop(p):
    '''do_while_loop : DO LCURLY statements RCURLY WHILE LPAREN condition RPAREN SEMICOLON'''
    # print("Parsed do-while loop")
    p[0] = ('do_while_loop', p[3], p[6])

def p_condition(p):
    '''condition : ID LT NUMBER'''
    p[0] = ('condition', p[1], p[2], p[3])

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

def p_statement(p):
    '''statement : ID ASSIGN NUMBER SEMICOLON'''
    p[0] = ('assignment', p[1], p[3])

# Error rule for syntax errors
def p_error(p):
    if p:
    |   print(f"Invalid expression at '{p.value}'")
    else:
    |   print("Invalid expression at EOF")
```

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

# Test input for do-while (JavaScript style)
data = '''
do {
    i = 5;
} while (i < 10);
'''

# Tokenize
# lexer.input(data)

# Check lexer output
# print("Lexer output:")
# for tok in lexer:
#     print(tok)

# Parse the input
result = parser.parse(data)
if result:
    print("Expression is valid")
else:
    print("Expression is invalid")
```

Output Screenshot :

Valid syntax:

```
data = '''
do {
    i = 5;
} while (i < 10);
'''
```

```
Expression is valid
```

Invalid syntax:

```
data = '''
do {
    i  5;
} while (i > 10);
'''
```

```
Invalid expression at '5'
Illegal character '>'
Expression is invalid
```

Lexer and parser code(error handling):

```
import ply.lex as lex
import ply.yacc as yacc

# Lexer definition
tokens = (
    'TRY',
    'CATCH',
    'FINALLY',
    'LCURLY',
    'RCURLY',
    'IDENTIFIER',
    'STRING',
    'SEMICOLON',
    'LPAREN',
    'RPAREN',
    'DOT',
    'COMMA',
)

# Token specifications
t_ignore = ' \t'  # Ignore spaces and tabs
t_TRY = r'try'
t_CATCH = r'catch'
t_FINALLY = r'finally'
t_LCURLY = r'\{'
t_RCURLY = r'\}'
t_SEMICOLON = r';'
t_STRING = r'\".*?\"|\'[^\']*\''
t_DOT = r'\.'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_COMMA = r','
```

```
def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = {
        'try': 'TRY',
        'catch': 'CATCH',
        'finally': 'FINALLY'
    }.get(t.value, 'IDENTIFIER')
    return t

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

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

# lexer building
lexer = lex.lex()

# Parser definition
def p_program(p):
    '''program : try_block catch_block finally_block
               | try_block catch_block
               | try_block finally_block
               | try_block'''
    p[0] = ('program', p[1:])
    # print("Parsed program")

def p_try_block(p):
    '''try_block : TRY LCURLY statements RCURLY'''
    p[0] = ('try_block', p[3])
    # print("Parsed try block")
```

```
def p_catch_block(p):
    '''catch_block : CATCH LPAREN IDENTIFIER RPAREN LCURLY statements RCURLY'
    p[0] = ('catch_block', p[3], p[6])
    # print("Parsed catch block")

def p_finally_block(p):
    '''finally_block : FINALLY LCURLY statements RCURLY'''
    p[0] = ('finally_block', p[3])
    # print("Parsed finally block")

def p_statements(p):
    '''statements : statement
    |   |   |   | | statements statement'''
    p[0] = p[1:] if len(p) > 2 else p[1]
    # print("Parsed statements")

def p_statement(p):
    '''statement : method_call SEMICOLON
    |   |   |   || STRING SEMICOLON
    |   |   |   || empty'''
    p[0] = p[1]
    # print("Parsed statement")

def p_method_call(p):
    '''method_call : IDENTIFIER DOT IDENTIFIER LPAREN args RPAREN'''
    p[0] = ('method_call', p[1], p[3], p[5])
    # print("Parsed method call")

def p_args(p):
    '''args : STRING
    |   |   | args COMMA STRING
    |   |   | empty'''
    p[0] = p[1:] if len(p) > 2 else p[1]
```

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

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

# Build the parser
parser = yacc.yacc(debug=False)

# Example input for testing
input_data = """
try {
    console.log("Attempting...");
}
catch (error) {
    console.error("An error occurred.");
}
finally {
    console.log("Cleanup.");
}
"""

# Parse the input and determine validity
result = parser.parse(input_data)
```

```
if result:
    print(f"Valid syntax!")
else:
    print("Invalid syntax.")
```

Output Screenshot :

Valid syntax:

```
input_data = """
try {
    console.log("Attempting...");
}
catch (error) {
    console.error("An error occurred.");
}
finally {
    console.log("Cleanup.");
}
"""
```

```
Valid syntax!
```

Invalid syntax:

```
input_data = """
try {
    console.log("Attempting...");
}
catch error {
    console.log("An error occurred.");
}
finally {
    console.log("Cleanup.");
}
"""
```

```
Syntax error at token 'error' on line 5.
Invalid syntax.
```

Lexer and Parser code for (if,if-else,nested if):

```
class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

def lexer(code):
    tokens = []
    words = code.replace("(", " ( ").replace(")", " ) ").replace("{", " { ").replace("}", " } ").split()

    for word in words:
        if word == "if":
            tokens.append(Token("IF", word))
        elif word == "else":
            tokens.append(Token("ELSE", word))
        elif word == "(":
            tokens.append(Token("OPEN_PAREN", word))
        elif word == ")":
            tokens.append(Token("CLOSE_PAREN", word))
        elif word == "{":
            tokens.append(Token("OPEN_BRACE", word))
        elif word == "}":
            tokens.append(Token("CLOSE_BRACE", word))
        elif word.isidentifier():
            tokens.append(Token("IDENTIFIER", word))
        else:
```

```
            raise ValueError(f"Unexpected token: {word}")
    return tokens

def parse_if_else(tokens):
    i = 0

    def expect(expected_type):
        nonlocal i
        if tokens[i].type != expected_type:
            raise ValueError(f"Unexpected token: {tokens[i].value}")
        i += 1

    def parse_expression():
        expect("IDENTIFIER")

    def parse_block():
        expect("OPEN_BRACE")
        parse_expression()
        expect("CLOSE_BRACE")

    def parse_if_statement():
        expect("IF")
        expect("OPEN_PAREN")
```

```
        parse_expression()
        expect("CLOSE_PAREN")
        parse_block()

        if i < len(tokens) and tokens[i].type == "ELSE":
            expect("ELSE")
            parse_block()

    parse_if_statement()

def main():
    code = input("Enter code to parse: ")
    try:
        tokens = lexer(code)
        parse_if_else(tokens)
        print("Valid syntax")
    except ValueError as e:
        print("Invalid syntax:", e)

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

Output Screenshot :

Valid syntax:

```
Enter code to parse: if (condition) { statement }
Valid syntax
```

Invalid syntax:

```
Enter code to parse: if () { statement } else { other_statement }
Invalid syntax: Unexpected token: )
```

Lexer and Parser code for (variable declaration):

```
class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

def lexer(code):
    tokens = []
    words = code.replace("=", " = ").replace(";", " ; ").split()

    i = 0
    while i < len(words):
        word = words[i]
        if word in ["int", "float", "string"]:
            tokens.append(Token("TYPE", word))
        elif word == "=":
            tokens.append(Token("EQUALS", word))
        elif word == ";":
            tokens.append(Token("SEMICOLON", word))
        elif word.isidentifier():
            tokens.append(Token("IDENTIFIER", word))
        elif word.isdigit():
            tokens.append(Token("INTEGER", word))
        elif word.replace('.', '', 1).isdigit() and word.count('.') == 1:
            tokens.append(Token("FLOAT", word))
        elif word.startswith('"') and word.endswith('"'):
            tokens.append(Token("STRING", word))
        else:
            raise ValueError(f"Unexpected token: {word}")
        i += 1
    return tokens

def parse_variable_declaration(tokens):
    i = 0

    def expect(expected_type):
        nonlocal i
        if i >= len(tokens) or tokens[i].type != expected_type:
            raise ValueError(f"Unexpected token: {tokens[i].value if i < len(tokens) else 'EOF'}")
        i += 1

    expect("TYPE")
    expect("IDENTIFIER")
    expect("EQUALS")
    if tokens[i-3].value == "int":
        expect("INTEGER")
    elif tokens[i-3].value == "float":
        expect("FLOAT")
    elif tokens[i-3].value == "string":
        expect("STRING")
    else:
        raise ValueError("Invalid type for variable declaration")

    expect("SEMICOLON")

def main():
    code = input("Enter code to parse: ")
    try:
        tokens = lexer(code)
        parse_variable_declaration(tokens)
        print("Valid syntax")
    except ValueError as e:
        print("Invalid syntax:", e)

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

Output Screenshot :

Valid syntax:

```
Enter code to parse: int x = 10;
Valid syntax
```

Invalid syntax:

```
Enter code to parse: double z = 5.5;
Invalid syntax: Unexpected token: double
```