PES University, RR Campus Bangalore 560085

Automata Formal Language and Logic

UE23CS243A

Mini Project

3rd Semester, Academic Year 2024

Date: 06/11/2023

To describe and write Grammar for the constraints of a programming language

Language : Java

Constructs : Looping Statements, Conditional Statements And Method Definition

Name	SRN	Section :
CHAUDHARI DIPAK RAJENDRA	PES1UG24CS804	K

The Lexar Program :(lexer.py)

```
import ply.lex as lex

tokens = (
    'IF', 'ELSE', 'FOR', 'WHILE', 'DO',
    'RETURN', 'VOID', 'IDENTIFIER', 'NUMBER',
    'LPAREN', 'RPAREN', 'LBRACE', 'RBRACE',
    'SEMICOLON', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE',
    'ASSIGN', 'LT', 'GT', 'NOT', 'COMMA', 'STRING_LITERAL'
)
t_IF = r'if'
t_ELSE = r'else'
t_FOR = r'for'
t_WHILE = r'while'
t_DO = r'do'
t_RETURN = r'return'
t_VOID = r'void'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACE = r'\{'
```

```
t_RBRACE = r'\}'
t_SEMICOLON = r';'
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_ASSIGN = r'='
t_LT = r'<'
t_GT = r'>'
t_NOT = r'!'
t_COMMA = r','
t_STRING_LITERAL = r'"([^"\\]*(\\.[^"\\]*)*)"'

def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    if t.value in ('if', 'else', 'for', 'while', 'do', 'return', 'void'):
        t.type = t.value.upper()
    return t

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

t_ignore = ' \t'
def t_NEWLINE(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
def t_error(t):
    print(f"Illegal character '{t.value[0]}'")
    t.lexer.skip(1)

lexer = lex.lex()

def validate_syntax(tokens):
    stack = []
    valid_constructs = {
        'IF': 'condition',
        'WHILE': 'loop',
        'FOR': 'loop',
        'DO': 'do-while',
        'RETURN': 'return',
        'VOID': 'method'
    }
    last_token = None
    expecting_condition = False
    expecting_semi = False
    in_for_loop = False
    for_parts = 0
    in_if_condition = False
    last_operator = None
```

```
    for token in tokens:
        if token.type in ('LBRACE', 'LPAREN'):
            stack.append(token)

        elif token.type in ('RBRACE', 'RPAREN'):
            if not stack:
                print(f"Unmatched closing token: {token.value}")
                return False
            if (token.type == 'RBRACE' and stack[-1].type != 'LBRACE') or \
               (token.type == 'RPAREN' and stack[-1].type != 'LPAREN'):
                print(f"Unmatched closing token: {token.value}")
                return False
            stack.pop()

        if last_token and last_token.type in valid_constructs:
            if token.type == 'LPAREN':
                expecting_condition = True
                if last_token.type == 'IF':
                    in_if_condition = True
            elif expecting_condition and token.type == 'RPAREN':
                expecting_condition = False
                if last_token and last_token.type in ('GT', 'LT', 'GTE', 'LTE',
'EQ', 'NEQ'):
                    print("Invalid if condition: cannot end with a comparison
operator.")
                    return False
                if last_operator is None:
                    print("Invalid if condition: incomplete condition before
closing parenthesis.")
                    return False
                in_if_condition = False
                last_operator = None
            elif token.type in ('GT', 'LT', 'GTE', 'LTE', 'EQ', 'NEQ'):
                last_operator = token.type
            elif expecting_condition and token.type not in ('LBRACE',
'SEMICOLON'):
                print(f"Invalid token after {last_token.type}: {token.value}")
                return False

        if last_token and last_token.type == 'FOR':
            in_for_loop = True

        if in_for_loop:
            if token.type == 'SEMICOLON':
                for_parts += 1
                if for_parts > 3:
                    print("Too many parts in for loop.")
                    return False
                if for_parts == 1:
                    expecting_condition = True
                elif for_parts == 2:
                    expecting_condition = False
```

```
            elif token.type == 'RPAREN':
                if for_parts != 2:
                    print("Invalid for loop structure: not enough parts.")
                    return False
                in_for_loop = False
                for_parts = 0

        last_token = token

    if stack:
        print("Unmatched opening token(s):", ', '.join([t.value for t in
stack]))
        return False

    return True
```

The Main Program :(main.py)

```
from lexer import lexer, validate_syntax

def main():
    print("Enter Java code (press Ctrl+D or Ctrl+Z on a new line to finish):")
    code = []
    while True:
        try:
            line = input()
            code.append(line)
        except EOFError:
            break
    code = "\n".join(code)
    lexer.input(code)
    tokens = []
    for token in lexer:
        tokens.append(token)

    print("\nTokens:")
    for token in tokens:
        print(f"Type: {token.type}, Value: '{token.value}', Line:
{token.lineno}, Position: {token.lexpos}")

    # Validate syntax
    if validate_syntax(tokens):
        print("Syntax is valid.")
    else:
        print("Syntax is invalid.")

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

The Output

Conditional Statements -

```
PS C:\Users\Dipak\Desktop\LexicalAnalysis in Java\python> python main.py
Enter Java code (press Ctrl+D or Ctrl+Z on a new line to finish):
if(a>b){
^Z

Tokens:
Type: IF, Value: 'if', Line: 1, Position: 0
Type: LPAREN, Value: '(', Line: 1, Position: 2
Type: IDENTIFIER, Value: 'a', Line: 1, Position: 3
Type: GT, Value: '>', Line: 1, Position: 4
Type: IDENTIFIER, Value: 'b', Line: 1, Position: 5
Type: RPAREN, Value: ')', Line: 1, Position: 6
Type: LBRACE, Value: '{', Line: 1, Position: 7
Unmatched opening token(s): {
Syntax is invalid.
```

```
PS C:\Users\Dipak\Desktop\LexicalAnalysis in Java\python> python main.py
Enter Java code (press Ctrl+D or Ctrl+Z on a new line to finish):
if(a>b){
}
^Z

Tokens:
Type: IF, Value: 'if', Line: 1, Position: 0
Type: LPAREN, Value: '(', Line: 1, Position: 2
Type: IDENTIFIER, Value: 'a', Line: 1, Position: 3
Type: GT, Value: '>', Line: 1, Position: 4
Type: IDENTIFIER, Value: 'b', Line: 1, Position: 5
Type: RPAREN, Value: ')', Line: 1, Position: 6
Type: LBRACE, Value: '{', Line: 1, Position: 7
Type: RBRACE, Value: '}', Line: 2, Position: 9
Syntax is valid.
```

Method Declaration:-

```
PS C:\Users\Dipak\Desktop\LexicalAnalysis in Java\python> python main.py
Enter Java code (press Ctrl+D or Ctrl+Z on a new line to finish):
void hello({}
^Z

Tokens:
Type: VOID, Value: 'void', Line: 1, Position: 0
Type: IDENTIFIER, Value: 'hello', Line: 1, Position: 5
Type: LPAREN, Value: '(', Line: 1, Position: 10
Type: LBRACE, Value: '{', Line: 1, Position: 11
Type: RBRACE, Value: '}', Line: 1, Position: 12
Unmatched opening token(s): (
Syntax is invalid.
```

```
PS C:\Users\Dipak\Desktop\LexicalAnalysis in Java\python> python main.py
Enter Java code (press Ctrl+D or Ctrl+Z on a new line to finish):
void hello(){}
^Z

Tokens:
Type: VOID, Value: 'void', Line: 1, Position: 0
Type: IDENTIFIER, Value: 'hello', Line: 1, Position: 5
Type: LPAREN, Value: '(', Line: 1, Position: 10
Type: RPAREN, Value: ')', Line: 1, Position: 11
Type: LBRACE, Value: '{', Line: 1, Position: 12
Type: RBRACE, Value: '}', Line: 1, Position: 13
Syntax is valid.
```

Looping Statements:-

```
PS C:\Users\Dipak\Desktop\LexicalAnalysis in Java\python> python main.py
Enter Java code (press Ctrl+D or Ctrl+Z on a new line to finish):
for(int i=0 i<1 i++)

^Z

Tokens:
Type: FOR, Value: 'for', Line: 1, Position: 0
Type: LPAREN, Value: '(', Line: 1, Position: 3
Type: IDENTIFIER, Value: 'int', Line: 1, Position: 4
Type: IDENTIFIER, Value: 'i', Line: 1, Position: 8
Type: ASSIGN, Value: '=', Line: 1, Position: 9
Type: NUMBER, Value: '0', Line: 1, Position: 10
Type: IDENTIFIER, Value: 'i', Line: 1, Position: 12
Type: LT, Value: '<', Line: 1, Position: 13
Type: NUMBER, Value: '1', Line: 1, Position: 14
Type: IDENTIFIER, Value: 'i', Line: 1, Position: 16
Type: PLUS, Value: '+', Line: 1, Position: 17
Type: PLUS, Value: '+', Line: 1, Position: 18
Type: RPAREN, Value: ')', Line: 1, Position: 19
Invalid for loop structure: not enough parts.
Syntax is invalid.
```

```
PS C:\Users\Dipak\Desktop\LexicalAnalysis in Java\python> python main.py
Enter Java code (press Ctrl+D or Ctrl+Z on a new line to finish):
for(int i=0; i<1; i++){}
^Z

Tokens:
Type: FOR, Value: 'for', Line: 1, Position: 0
Type: LPAREN, Value: '(', Line: 1, Position: 3
Type: IDENTIFIER, Value: 'int', Line: 1, Position: 4
Type: IDENTIFIER, Value: 'i', Line: 1, Position: 8
Type: ASSIGN, Value: '=', Line: 1, Position: 9
Type: NUMBER, Value: '0', Line: 1, Position: 10
Type: SEMICOLON, Value: ';', Line: 1, Position: 11
Type: IDENTIFIER, Value: 'i', Line: 1, Position: 13
Type: LT, Value: '<', Line: 1, Position: 14
Type: NUMBER, Value: '1', Line: 1, Position: 15
Type: SEMICOLON, Value: ';', Line: 1, Position: 16
Type: IDENTIFIER, Value: 'i', Line: 1, Position: 18
Type: PLUS, Value: '+', Line: 1, Position: 19
Type: PLUS, Value: '+', Line: 1, Position: 20
Type: RPAREN, Value: ')', Line: 1, Position: 21
Type: LBRACE, Value: '{', Line: 1, Position: 22
Type: RBRACE, Value: '}', Line: 1, Position: 23
Syntax is valid.
```