PES UNIVERSITY

Department of Computer Science and Engineering

Electronic City Campus, Bangalore

Automata Formal Language and Logic

UE23CS242A

Mini Project

3rd Semester, Academic Year 2024

Date: 12/11/2024

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

Language : Python

Constructs :

Name of team members	SRN of team members	Section
M Niranjan	PES2UG23CS308	E
Keerthan P.V	PES2UG23CS272	E

Grammar:

1. **Function Declaration:**

S -> 'def' ID '(' parameters ')' ':'

parameters -> parameter (',' parameter)*
parameter -> ID | ID ':' T
T -> int | str | bool | float | None
ID -> letter (letter | digit) *
letter -> [a-zA-Z_]
digit -> [0-9]

2. <u>Looping Constructs – While Loop</u>

S -> 'while' CONDITION ':' NEWLINE INDENT BLOCK DEDENT
CONDITION -> expression
BODY -> STATEMENT (NEWLINE STATEMENT)*
STATEMENT -> assignment | function_call | loop_control
assignment -> ID '=' VALUE
loop_control -> 'continue' | 'break'
VALUE -> expression | literal
ID -> letter (letter | digit) *
letter -> [a-zA-Z_]
digit -> [0-9]

3. <u>Datatype Declaration</u>

S -> ID ':' T '=' VALUE
T -> int | float | str | bool | None
VALUE -> expression | literal
ID -> letter (letter | digit) *
letter -> [a-zA-Z_]
digit -> [0-9]

4. <u>if_statement</u>

if_statement -> 'if' CONDITION ':' NEWLINE INDENT BLOCK DEDENT | 'if' CONDITION ':' NEWLINE INDENT BLOCK DEDENT elif_clauses else_clause? elif_clauses -> ('elif' CONDITION ':' NEWLINE INDENT BLOCK DEDENT)* else_clause -> 'else' ':' NEWLINE INDENT BLOCK DEDENT
CONDITION -> expression
BLOCK -> STATEMENT (NEWLINE STATEMENT)*
STATEMENT -> assignment | function_call | loop_control | if_statement assignment -> ID '=' VALUE
loop_control -> 'continue' | 'break'
VALUE -> expression | literal
ID -> letter (letter | digit)*
letter -> [a-zA-Z_]
digit -> [0-9]

5. Array Decleration

S -> ID ':' 'list' '[' T ']' '=' ARRAY_LITERAL
| ID ':' 'list' '[' T ']' # for type annotation without assignment
T -> int | float | str | bool | None
ARRAY_LITERAL -> '[' ELEMENT (',' ELEMENT)* ']' ELEMENT -> VALUE

VALUE -> expression | literal

ID -> letter (letter | digit)*
letter -> [a-zA-Z_]
digit -> [0-9]

The Lexar Program:

```
import ply.lex as lex
import ply.yacc as yacc
tokens = (
   'IF', 'ELSE', 'FOR', 'IN', 'RANGE',
   'SWITCH', 'CASE', 'BREAK', 'DEF', 'GT',
   'LT', 'EQ', 'COMMA', 'RETURN', 'LPAREN',
   'RPAREN', 'LBRACE', 'RBRACE', 'COLON',
   'SEMICOLON', 'EQUALS', 'NUMBER', 'STRING',
   'BOOLEAN', 'INCREMENT', 'IDENTIFIER'
)

reserved = {
   'if': 'IF', 'else': 'ELSE', 'for': 'FOR',
   'in': 'IN', 'range': 'RANGE', 'switch': 'SWITCH', 'case':
   'CASE', 'return': 'RETURN', 'true': 'BOOLEAN', 'false':
   'BOOLEAN', 'break': 'BREAK', 'def': 'DEF',
}

def t_IDENTIFIER(t):
   r'[a-zA-Z_][a-zA-Z0-9_]*'
   t.type = reserved.get(t.value, 'IDENTIFIER')
```

```
    return t
t_LPAREN = r'\('

t_RPAREN = r'\)'

t_LBRACE = r'\{'

t_RBRACE = r'\}'

t_COLON = r':'

t_GT = r'>'

t_COMMA = r','

t_EQUALS = r'='

t_STRING = r'\"[^"]*\"'

t_INCREMENT = r'\+\+'

t_LT = r'<'

t_EQ = r'=='

t_SEMICOLON = r';'


def t_NUMBER(t):

    r'\d+'

    t.value = int(t.value)

    return t


def t_newline(t):

    r'\n+'

    t.lexer.lineno += len(t.value)


t_ignore = ' \t'


def t_error(t):

    print(f"Illegal character: '{t.value[0]}' at line

    {t.lexer.lineno}") t.lexer.skip(1)


precedence = (
    ('left', 'GT', 'LT', 'EQ'),

)


def p_program(p):
```

```
    '''program : statement_list'''
    p[0] = p[1]

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

def p_statement(p):
    '''statement : if_statement
                 | for_statement
                 | switch_statement
                 | function_def
                 | assignment_statement
                 | return_statement
                 | function_call
                 | block'''
    p[0] = p[1]

def p_if_statement(p):
    '''if_statement : IF LPAREN condition RPAREN block |
    IF LPAREN condition RPAREN block ELSE block''' if
    len(p) == 6:
        p[0] = ('if', p[3], p[5], None)
    else:
        p[0] = ('if', p[3], p[5], p[7])

def p_for_statement(p):
    '''for_statement : FOR IDENTIFIER IN RANGE LPAREN NUMBER RPAREN
    block''' p[0] = ('for', p[2], ('range', p[6]), p[8])
```

```
    def p_switch_statement(p):
'''switch_statement : SWITCH LPAREN expression RPAREN LBRACE case_list RBRACE'''

    def p_case_list(p):
        '''case_list : case
                     | case_list case'''
        if len(p) == 2:
            p[0] = [p[1]]
        else:
            p[0] = p[1] + [p[2]]

    def p_case(p):
        '''case : CASE expression COLON statement_list BREAK
        SEMICOLON''' p[0] = ('case', p[2], p[4])

    def p_function_def(p):
        '''function_def : DEF IDENTIFIER LPAREN RPAREN COLON
        statement''' p[0] = ('function_def', p[2], [], p[6])

    def p_return_statement(p):
        '''return_statement : RETURN expression SEMICOLON'''
        p[0] = ('return', p[2])

    def p_block(p):
        '''block : LBRACE statement_list RBRACE
                 | LBRACE RBRACE'''
        if len(p) == 4:
            p[0] = ('block', p[2])
        else:
            p[0] = ('block', [])

    def p_function_call(p):
        '''function_call : IDENTIFIER LPAREN argument_list RPAREN
```

```
    SEMICOLON"' p[0] = ('function_call', p[1], p[3])


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


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


def p_assignment_statement(p):
    "'assignment_statement : IDENTIFIER EQUALS expression SEMICOLON"'
    p[0] = ('assignment', p[1], p[3])


def p_expression(p):
    "'expression : IDENTIFIER
                | NUMBER
                | STRING
                | BOOLEAN"'
    p[0] = ('expression', p[1])


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

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

lexer = lex.lex()
parser = yacc.yacc()

test_cases = [
    """for i in range(10){
        break;
    }
    """,
    """if (x < y) [
        something();
    ] else {
        something_else();
    }""",
    """switch (x) {
        case 1:
            something();
            break;
        case 2:
            something_else();
            break;
    }""",
    """def function():
        return 0;"""
]

def test_parser():
    for test in test_cases:
```

```
    print(f"\nParsing:\n{test}"
    ) try:
        result =
    parser.parse(test)
    print("Parse successful!")
    except Exception as e:
        print(f'Parse failed: {str(e)}")


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

Output:

```
Parsing:
import math()
Syntax error at '(' (line 13)
Parse successful!

Parsing:
from os import path
Syntax error at EOF
Parse successful!

Parsing:
class MyClass:
        def __init__(self):
            pass;

Syntax error at 'def' (line 14)
Syntax error at ':' (line 14)
Parse successful!
```

```
Parsing:
for i in range(10){
            pass;
        }

Syntax error at '{' (line 1)
Parse successful!

Parsing:
def function():
        return 0;
Parse successful!

Parsing:
switch (x):
        case 1[
            something();
            break;
            ]
        case 2:
            something_else();
            break;

Illegal character: '[' at line 6
Syntax error at 'something' (line 7)
Illegal character: ']' at line 9
Syntax error at 'break' (line 12)
Parse successful!
```