Automata Formal Language and

Logic

CONSTRUCT 1: FOR LOOP

GRAMMER:

forLoop -> for (initialization ; condition ; increment)
statement

initialization -> expression

condition -> expression

increment -> expression

LEXAR CODE:

```
from ply import lex
from ply import yacc

# Tokens
tokens = (
    'FOR', 'IN', 'IDENTIFIER', 'NUMBER', 'LPAREN', 'RPAREN', 'COLON', 'COMMA',
'RANGE'
)

# Token definitions
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_COLON = r':'
t_COMMA = r','

t_ignore = ' \t\n'

def t_FOR(t): r'for'; return t
def t_IN(t): r'in'; return t
def t_RANGE(t): r'range'; return t

def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    return t

def t_NUMBER(t):
    r'\d+'
```

```
    t.value = int(t.value)
    return t

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

# Create lexer
lexer = lex.lex()

# Parser rules

def p_for_loop(p):
    '''for_loop : FOR IDENTIFIER IN range_expr COLON'''
    p[0] = 'Valid for loop'

def p_range_expr(p):
    '''range_expr : RANGE LPAREN NUMBER RPAREN
                  | RANGE LPAREN NUMBER COMMA NUMBER RPAREN
                  | RANGE LPAREN NUMBER COMMA NUMBER COMMA NUMBER RPAREN'''

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

# Create parser
parser = yacc.yacc()

# Main loop for user input
def main():
    while True:
        try:
            check = input("Press Y/N to Validate Syntax: ")
            if check.upper() == 'N':
                break
            s = input('Enter for loop code: ')
            if not s:
                continue
            result = parser.parse(s)
            if result == "Valid for loop":
                print("Valid for loop syntax")
            else:
                print("Invalid for loop syntax")
        except EOFError:
            break
        except Exception as e:
```

```
            print(f"Error: {e}")

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

OUTPUT:

```
Generating LALR tables
Press Y/N to Validate Syntax: y
Valid for loop syntax
Press Y/N to Validate Syntax: y
Enter for loop code: for i range(5):
Syntax error at token: range
Invalid for loop syntax
Press Y/N to Validate Syntax: n
```

CONSTRUCT 2: ARRAY DECLARATION

GRAMMER:

array -> [elements]

elements -> element|

element , elementselement -> expression

LEXAR CODE:

```
from ply import lex
from ply import yacc

# List of token names
tokens = (
    'TYPE',
    'IDENTIFIER',
    'ASSIGN',
    'LBRACKET',
    'RBRACKET',
    'COMMA',
    'NUMBER',
    'STRING',
    'BOOLEAN',
    'NONE', # Using 'NONE' for Python's 'None'
    'SEMICOLON',
)

# Regular expression rules for simple tokens
t_IDENTIFIER = r'[a-zA-Z_][a-zA-Z0-9_]*'
t_ASSIGN = r'='
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_COMMA = r','
t_SEMICOLON = r';'

def t_TYPE(t):
    r'const|let|var'
    return t

def t_NUMBER(t):
    r'\d+\.\d+|\d+'  # Matches both float and integer numbers
    t.value = float(t.value) if '.' in t.value else int(t.value) # Convert to
appropriate numeric type
    return t
```

```
def t_STRING(t):
    r'\"([^\\\n]|(\\.))*?\"'
    t.value = t.value[1:-1]  # Remove the surrounding quotes
    return t

def t_BOOLEAN(t):
    r'true|false'
    t.value = t.value == 'true'  # Convert to boolean
    return t

def t_NONE(t):
    r'None'  # Python's None
    t.value = None
    return t

t_ignore = ' \t'

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

lexer = lex.lex()

# Define precedence and associativity
precedence = (
    ('left', 'COMMA'),
)

# Parsing rules
def p_statement(p):
    '''statement : declaration SEMICOLON
                 | declaration'''  # Allow the semicolon to be optional
    p[0] = "Valid"

def p_declaration(p):
    '''declaration : TYPE IDENTIFIER ASSIGN array'''
    # You can add more actions here if needed

def p_array(p):
    '''array : LBRACKET elements_opt RBRACKET'''
    # You can add more actions here if needed

def p_elements_opt(p):
    '''elements_opt : elements
                    | elements COMMA
                    | empty'''  # Allow an optional ending comma

def p_elements(p):
```

```
    '''elements : value
                | elements COMMA value'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]

def p_value(p):
    '''value : NUMBER
             | STRING
             | BOOLEAN
             | NONE
             | array'''
    p[0] = p[1]

def p_empty(p):
    'empty :'
    pass

def p_error(p):
    if p:
        print("Syntax error at token:", p.value)
    else:
        print("Syntax error at EOF")

parser = yacc.yacc()

if __name__ == '__main__':
    while True:
        try:
            check = input("Press Y/N to Validate Syntax: ")
            if check == 'N':
                exit(0)
            else:
                s = input('Enter Python code: ')
        except EOFError:
            break
        if not s:
            continue
        result = parser.parse(s)
        if result == "Valid":
            print("Valid syntax\n")
```

OUTPUT:

```
Generating LALR tables
Press Y/N to Validate Syntax: y
Enter Python code: let myArray = [1, 2, 3];
Valid syntax

Press Y/N to Validate Syntax: y
Enter Python code: var mixedArray = [42, "Hello", false, None];
Valid syntax

Press Y/N to Validate Syntax: y
Enter Python code: let my[1,2]
Syntax error at token: [
Press Y/N to Validate Syntax: n
```

CONSTRUCT 3: FUNCTION DECLARATION

GRAMMER:

<function_declaration> -> <return_type> <function_name> (<parameter_list>) {

<statements> }

<parameter_list> -> <parameter> | <parameter>, <parameter_list> | ε

<parameter> -> <data_type> <identifier>

<return_type> -> <data_type> | void

<data_type> -> int | float | char | <identifier>

LEXAR CODE:

```
from ply import lex

from ply import yacc


# List of token names
tokens = (
    'DEF', 'IDENTIFIER', 'NUMBER', 'STRING', 'ASSIGN', 'WHILE', 'LPAREN',
'RPAREN',
    'LBRACE', 'RBRACE', 'PRINT', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'NULL',
    'BOOLEAN', 'COMMA', 'SEMICOLON'
)

# Regular expressions for simple tokens
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_ASSIGN = r'='
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_COMMA = r','
t_SEMICOLON = r';'

# More complex tokens
def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    # Check for reserved words
    if t.value == 'def':
        t.type = 'DEF'
    elif t.value == 'while':
        t.type = 'WHILE'
    elif t.value == 'print':
        t.type = 'PRINT'
    elif t.value in ('true', 'false'):
        t.type = 'BOOLEAN'
    elif t.value == 'null':
        t.type = 'NULL'
    return t
```

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

def t_STRING(t):
    r'\"([^\\\n]|(\\.))*?\"'
    return t

# Ignored characters (spaces, tabs, newlines)
t_ignore = ' \t\n'

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

# Build the lexer
lexer = lex.lex()

# Parsing rules
def p_program(p):
    '''program : statements'''
    p[0] = p[1]

def p_statements(p):
    '''statements : statement
                  | statements statement'''
    if len(p) == 2:
        p[0] = [p[1]] if p[1] is not None else []
    else:
        p[0] = p[1] + [p[2]] if p[2] is not None else p[1]

def p_statement(p):
    '''statement : function_decl
                 | while_stmt
                 | assign_stmt
                 | print_stmt'''
    p[0] = p[1]

def p_function_decl(p):
    '''function_decl : DEF IDENTIFIER LPAREN param_list RPAREN LBRACE
statements RBRACE'''
    p[0] = ('function_decl', p[2], p[4], p[7])

def p_param_list(p):
    '''param_list : IDENTIFIER
                  | param_list COMMA IDENTIFIER
```

```
                | empty'''
    if len(p) == 2:
        p[0] = [p[1]] if p[1] is not None else []
    elif len(p) == 4:
        p[0] = p[1] + [p[3]]

def p_while_stmt(p):
    '''while_stmt : WHILE LPAREN expression RPAREN LBRACE statements RBRACE'''
    p[0] = ('while_stmt', p[3], p[6])

def p_assign_stmt(p):
    '''assign_stmt : IDENTIFIER ASSIGN expression SEMICOLON'''
    p[0] = ('assign_stmt', p[1], p[3])

def p_print_stmt(p):
    '''print_stmt : PRINT LPAREN expression_list RPAREN SEMICOLON'''
    p[0] = ('print_stmt', p[3])

def p_expression_list(p):
    '''expression_list : expression
                       | expression_list COMMA expression'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]

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

def p_term(p):
    '''term : IDENTIFIER
            | NUMBER
            | STRING
            | BOOLEAN
            | NULL
            | LPAREN expression RPAREN'''
    if len(p) == 2:
        p[0] = ('term', p[1])
    else:
        p[0] = p[2]
```

```
def p_empty(p):
    'empty :'
    pass

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

# Build the parser with error recovery
parser = yacc.yacc()

# Interactive loop for input
if __name__== '__main__':
    while True:
        try:
            check = input("Press Y/N to Validate Syntax: ")
            if check.lower() == 'n':
                break
            elif check.lower() == 'y':
                s = input('Enter Python-like code: ')
                if not s:
                    continue
                try:
                    result = parser.parse(s)
                    if result is not None:
                        print("Valid syntax")
                except:
                    print("Invalid syntax")
        except EOFError:
            break
```

OUTPUT:

```
Generating LALR tables
Press Y/N to Validate Syntax: y
Enter Python-like code: def my_function(x, y) { print("Sum:", x + y); }
Valid syntax
Press Y/N to Validate Syntax: y
Enter Python-like code: def example(x, y) ( print(x + y); }
Syntax error at token: (
Syntax error at token: }
Press Y/N to Validate Syntax: y
Enter Python-like code: def calculate(a, b) { result = (a + b) * (a - b); print(result); }
Valid syntax
Press Y/N to Validate Syntax: n
```

4) CONSTRUCT:DICTIONARY

GRAMMAR:

```
grammar = {
    "<function_declaration>": [
        ["<return_type>", "<function_name>", "(", "<parameter_list>", ")", "{",
"<statements>", "}"]
    ],
    "<parameter_list>": [
        ["<parameter>"],
        ["<parameter>", ",", "<parameter_list>"],
        ["ε"]
    ],
```

```
    "<parameter>": [
        ["<data_type>", "<identifier>"]
    ],
    "<return_type>": [
        ["<data_type>"],
        ["void"]
    ],
    "<data_type>": [
        ["int"],
        ["float"],
        ["char"],
        ["<identifier>"]
    ]
}
```

LEXER CODE:

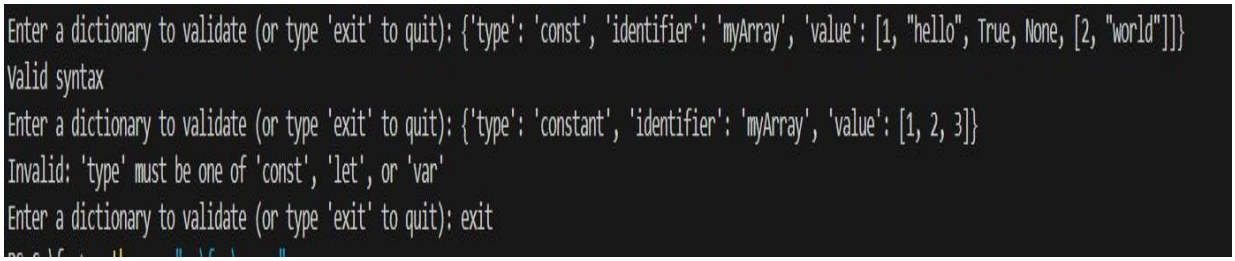
```
import re

from ply import lex
from ply import yacc
import ast  # To safely evaluate input dictionary from user

# List of token names
tokens = (
    'TYPE',
    'IDENTIFIER',
    'ASSIGN',
    'LBRACKET',
    'RBRACKET',
    'COMMA',
    'NUMBER',
    'STRING',
    'BOOLEAN',
    'NONE',  # Using 'NONE' for Python's 'None'
    'SEMICOLON',
)
```

```
# Regular expression rules for simple tokens
t_IDENTIFIER = r'[a-zA-Z_][a-zA-Z0-9_]*'
t_ASSIGN = r'='
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_COMMA = r','
t_SEMICOLON = r';'

def t_TYPE(t):
    r'const|let|var'
    return t

def t_NUMBER(t):
    r'\d+\.\d+|\d+'  # Matches both float and integer numbers
    t.value = float(t.value) if '.' in t.value else int(t.value)
    return t

def t_STRING(t):
    r'\"([^\\\n]|(\\.))*?\"'
    t.value = t.value[1:-1]
    return t

def t_BOOLEAN(t):
    r'true|false'
    t.value = t.value == 'true'
    return t

def t_NONE(t):
    r'None'
    t.value = None
    return t

t_ignore = ' \t'

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

lexer = lex.lex()

# Define precedence and associativity
precedence = (
    ('left', 'COMMA'),
)

# Parsing rules
def p_statement(p):
    '''statement : declaration SEMICOLON
```

```
                | declaration'''  # Allow the semicolon to be optional
    p[0] = "Valid"

def p_declaration(p):
    '''declaration : TYPE IDENTIFIER ASSIGN array'''

def p_array(p):
    '''array : LBRACKET elements_opt RBRACKET'''

def p_elements_opt(p):
    '''elements_opt : elements
                    | elements COMMA
                    | empty'''

def p_elements(p):
    '''elements : value
                | elements COMMA value'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]

def p_value(p):
    '''value : NUMBER
             | STRING
             | BOOLEAN
             | NONE
             | array'''
    p[0] = p[1]

def p_empty(p):
    'empty :'
    pass

def p_error(p):
    if p:
        print("Syntax error at token:", p.value)
    else:
        print("Syntax error at EOF")

parser = yacc.yacc()

# Validation function for dictionary input
def validate_dict(data):
    # Check if data is a dictionary
    if not isinstance(data, dict):
        return "Invalid: Input is not a dictionary"
```

```
    # Check for required keys
    required_keys = {'type', 'identifier', 'value'}
    if not required_keys.issubset(data.keys()):
        return "Invalid: Missing required keys"

    # Validate 'type'
    if data['type'] not in {'const', 'let', 'var'}:
        return "Invalid: 'type' must be one of 'const', 'let', or 'var'"

    # Validate 'identifier' (must match identifier rules)
    if not isinstance(data['identifier'], str) or not re.match(r'^[a-zA-Z_][a-
zA-Z0-9_]*$', data['identifier']):
        return "Invalid: 'identifier' must be a valid identifier string"

    # Validate 'value' - recursive check for allowed data types
    if not validate_value(data['value']):
        return "Invalid: 'value' contains unsupported elements"

    return "Valid syntax"

def validate_value(value):
    # Check if value is a basic allowed type
    if isinstance(value, (int, float, str, bool)) or value is None:
        return True
    # Check if value is a list (array) and recursively validate each element
    elif isinstance(value, list):
        return all(validate_value(element) for element in value)
    else:
        return False

# Main function to prompt user input and validate it
if __name__== '__main__':
    while True:
        user_input = input("Enter a dictionary to validate (or type 'exit' to
quit): ")
        if user_input.lower() == 'exit':
            break
        try:
            # Safely parse user input as a dictionary
            test_data = ast.literal_eval(user_input)
            if isinstance(test_data, dict):
                print(validate_dict(test_data))
            else:
                print("Invalid input: Please enter a valid dictionary.")
        except (SyntaxError, ValueError):
            print("Invalid input: Please enter a valid dictionary format.")
```

OUTPUT:



4) CONSTRUCT: CLASS DECLARATION

GRAMMAR:

<class_declaration> -> class <class_name> { <class_body> }

<class_body> -> <member_declaration> <class_body> | ε

<member_declaration> -> <field_declaration> | <method_declaration>

<field_declaration> -> <data_type> <identifier> ;

<method_declaration> -> <return_type> <method_name> (<parameter_list>) { <statements> }

<parameter_list> -> <parameter> | <parameter>, <parameter_list> | ε

<parameter> -> <data_type> <identifier>

<return_type> -> <data_type> | void

<data_type> -> int | float | char | <identifier>

LEXER CODE:

```
import re

from ply import lex
from ply import yacc
import ast  # To safely evaluate input dictionary from user

# List of token names
tokens = (
    'TYPE',
    'IDENTIFIER',
    'ASSIGN',
    'LBRACKET',
    'RBRACKET',
    'COMMA',
    'NUMBER',
    'STRING',
    'BOOLEAN',
    'NONE',  # Using 'NONE' for Python's 'None'
    'SEMICOLON',
)

# Regular expression rules for simple tokens
t_IDENTIFIER = r'[a-zA-Z_][a-zA-Z0-9_]*'
t_ASSIGN = r'='
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_COMMA = r','
t_SEMICOLON = r';'

def t_TYPE(t):
    r'const|let|var'
    return t

def t_NUMBER(t):
    r'\d+\.\d+|\d+'  # Matches both float and integer numbers
    t.value = float(t.value) if '.' in t.value else int(t.value)
    return t

def t_STRING(t):
    r'\"([^\\\n]|(\\.))*?\"'
    t.value = t.value[1:-1]
    return t

def t_BOOLEAN(t):
    r'true|false'
    t.value = t.value == 'true'
```

```
    return t

def t_NONE(t):
    r'None'
    t.value = None
    return t

t_ignore = ' \t'

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

lexer = lex.lex()

# Define precedence and associativity
precedence = (
    ('left', 'COMMA'),
)

# Parsing rules
def p_statement(p):
    '''statement : declaration SEMICOLON
                 | declaration'''  # Allow the semicolon to be optional
    p[0] = "Valid"

def p_declaration(p):
    '''declaration : TYPE IDENTIFIER ASSIGN array'''

def p_array(p):
    '''array : LBRACKET elements_opt RBRACKET'''

def p_elements_opt(p):
    '''elements_opt : elements
                    | elements COMMA
                    | empty'''

def p_elements(p):
    '''elements : value
                | elements COMMA value'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]

def p_value(p):
    '''value : NUMBER
             | STRING
```

```
              | BOOLEAN
              | NONE
              | array'''
    p[0] = p[1]

def p_empty(p):
    'empty :'
    pass

def p_error(p):
    if p:
        print("Syntax error at token:", p.value)
    else:
        print("Syntax error at EOF")

parser = yacc.yacc()

# Validation function for class creation input
class ClassValidator:
    def __init__(self, data):
        self.data = data

    def validate(self):
        # Check if data is a dictionary
        if not isinstance(self.data, dict):
            return "Invalid: Input is not a dictionary"

        # Check for required keys
        required_keys = {'class_name', 'attributes'}
        if not required_keys.issubset(self.data.keys()):
            return "Invalid: Missing required keys"

        # Validate 'class_name'
        if not isinstance(self.data['class_name'], str) or not re.match(r'^[A-
Za-z_][A-Za-z0-9_]*$', self.data['class_name']):
            return "Invalid: 'class_name' must be a valid identifier string"

        # Validate 'attributes' (should be a dictionary of attribute names and
their values)
        if not isinstance(self.data['attributes'], dict):
            return "Invalid: 'attributes' must be a dictionary"

        # Validate each attribute name and value
        for attribute, value in self.data['attributes'].items():
            if not re.match(r'^[a-zA-Z_][a-zA-Z0-9_]*$', attribute):
                return f"Invalid: Attribute name '{attribute}' is not valid"
            if not self._validate_value(value):
```

```
                return f"Invalid: Attribute '{attribute}' contains unsupported
value type"

        return "Valid class creation"

    def _validate_value(self, value):
        # Check if value is a basic allowed type
        if isinstance(value, (int, float, str, bool)) or value is None:
            return True
        # Check if value is a list (array) and recursively validate each
element
        elif isinstance(value, list):
            return all(self._validate_value(element) for element in value)
        return False

# Main function to prompt user input and validate it
if __name__ == '__main__':
    while True:
        user_input = input("Enter a class creation to validate (or type 'exit'
to quit): ")
        if user_input.lower() == 'exit':
            break
        try:
            # Safely parse user input as a dictionary
            test_data = ast.literal_eval(user_input)
            if isinstance(test_data, dict):
                validator = ClassValidator(test_data)
                print(validator.validate())
            else:
                print("Invalid input.")
        except (SyntaxError, ValueError):
            print("Invalid input: Please enter a valid  format.")
```

OUTPUT:

```
Enter a class creation dictionary to validate (or type 'exit' to quit): {'class_name': 'Person', 'attributes': {'name': 'John', 'age': 30, 'is_student': True}}
Valid class creation
Enter a class creation dictionary to validate (or type 'exit' to quit): {'class_name': 'Person', 'attributes': {'name': 'John', 'age': '30', 'is-student': True}}

Invalid: Attribute name 'is-student' is not valid
Enter a class creation dictionary to validate (or type 'exit' to quit): exit
```