AUTOMATA FORMAL LANGUAGES AND LOGIC ASSINGMENT

NAME : VAGEESH
SRN : PES2UG24CS663
SEC : K (CSE)

#Syntax Validation of a programming language by writing the Context Free Grammar, Using **Python Lex-Yaac(PLY)** tool.

Programming Language :: **Swift**

Swift is a powerful,intuitive, and modern open-source programming language developed by Apple Inc . It is designed for speed and efficiency,delivery performance comparable to C-based Languages .Use in Mobile App development, macOS Development, Server-Side Development etc.

lexer.py

```
import ply.lex as lex
lexer_error = False
# Reserved keywords in Swift subset
reserved = {
    'struct': 'STRUCT',
    'var': 'VAR',
    'let': 'LET',
    'where': 'WHERE',
    'protocol': 'PROTOCOL',
    'true': 'TRUE',
    'false': 'FALSE',
}
# List of token names
tokens = [
    'ID', 'NUMBER', 'STRING', 'ASSIGN', 'COLON', 'COMMA',
    'LBRACE', 'RBRACE', 'LANGLE', 'RANGLE', 'SEMICOLON',
] + list(reserved.values())

# Regular expressions for tokens
t_ASSIGN = r'='
t_COLON = r':'
t_COMMA = r','
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_LANGLE = r'<'
```

```
t_RANGLE = r'>'
t_SEMICOLON = r';'
# Identifier (variable, struct, etc.)
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'ID')
    return t
# Number (int/float/scientific)
def t_NUMBER(t):
    r"\d+(?:\.\d+)?(?:[eE][+-]?\d+)?"
    s = t.value
    if ('.' in s) or ('e' in s) or ('E' in s):
        try:
            t.value = float(s)
        except Exception:
            t.value = s
    else:
        try:
            t.value = int(s)
        except Exception:
            t.value = s
    return t
# String literal
def t_STRING(t):
    r'\"([^\\\n]|(\\.))*?\"'
    t.value = t.value[1:-1]
    return t
# Single-line comment
def t_COMMENT(t):
    r'//.*'
    pass
# Ignore whitespace
t_ignore = ' \t\r\n'
# Handle newlines
def t_newline(t):
    r'(\r\n|\n|\r)+'
    t.lexer.lineno += len(t.value)
# Error handling
def t_error(t):
    global lexer_error
    print(f"Illegal character '{t.value[0]}' at line {t.lexer.lineno}")
    lexer_error = True
    t.lexer.skip(1)
lexer = lex.lex()
```

parser.py

```
# Swift Parser using PLY
import ply.yacc as yacc
from lexer import tokens
precedence = (
    ('left', 'STRUCT'),
    ('left', 'LET', 'VAR'),
    ('left', 'COLON'),
    ('left', 'ASSIGN'),
    ('left', 'COMMA'),
    ('left', 'LANGLE', 'RANGLE'),
    ('left', 'WHERE'),
)
# --- Program Root ---
def p_program(p):
    '''program : declaration_list'''
    p[0] = ('program', p[1])
def p_declaration_list(p):
    '''declaration_list : declaration
                        | declaration_list declaration'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[2]]
def p_declaration(p):
    '''declaration : var_declaration
                   | struct_declaration
                   | protocol_declaration'''
    p[0] = p[1]
# --- Variable and Constant Declarations ---
def p_var_declaration(p):
    '''var_declaration : LET ID type_annotation_opt assign_opt
                       | VAR ID type_annotation_opt assign_opt'''
    p[0] = ('var_decl', p[1], p[2], p[3], p[4])
def p_type_annotation_opt(p):
    '''type_annotation_opt : COLON ID
                           | empty'''
    if len(p) == 3:
        p[0] = ('type_annotation', p[2])
    else:
        p[0] = None
def p_assign_opt(p):
    '''assign_opt : ASSIGN expression
                  | empty'''
    if len(p) == 3:
        p[0] = ('assignment', p[2])
```

```
    else:
        p[0] = None
def p_expression_number(p):
    'expression : NUMBER'
    p[0] = ('expression', 'NUMBER', p[1])
def p_expression_string(p):
    'expression : STRING'
    p[0] = ('expression', 'STRING', p[1])
def p_expression_id(p):
    'expression : ID'
    p[0] = ('expression', 'ID', p[1])
def p_expression_true(p):
    'expression : TRUE'
    p[0] = ('expression', 'TRUE', True)
def p_expression_false(p):
    'expression : FALSE'
    p[0] = ('expression', 'FALSE', False)

# --- Structures ---
def p_struct_declaration(p):
    '''struct_declaration : STRUCT ID generic_params_opt where_clause_opt LBRACE
property_declaration_list RBRACE'''
    p[0] = ('struct_decl', p[2], p[3], p[4], p[6])
def p_generic_params_opt(p):
    '''generic_params_opt : generic_params
                          | empty'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = None
def p_where_clause_opt(p):
    '''where_clause_opt : where_clause
                        | empty'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = None
def p_property_declaration_list(p):
    '''property_declaration_list : var_declaration
                                 | property_declaration_list var_declaration
                                 | empty'''
    if len(p) == 2:
        if p[1] is None:
            p[0] = []
        else:
            p[0] = [p[1]]
    else:
```

```
        p[0] = p[1] + [p[2]]
# --- Generics ---
def p_generic_params(p):
    'generic_params : LANGLE generic_parameter_list RANGLE'
    p[0] = ('generic_params', p[2])
def p_generic_parameter_list(p):
    '''generic_parameter_list : generic_parameter
                              | generic_parameter_list COMMA generic_parameter'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]
def p_generic_parameter(p):
    '''generic_parameter : ID
                         | constraint'''
    p[0] = p[1]

# --- Where Clause ---
def p_where_clause(p):
    'where_clause : WHERE constraint_list'
    p[0] = ('where_clause', p[2])
def p_constraint_list(p):
    '''constraint_list : constraint
                       | constraint_list COMMA constraint'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]
def p_constraint(p):
    '''constraint : ID COLON ID'''  # e.g., T: Equatable
    p[0] = ('constraint', p[1], p[3])
# --- Protocol Declaration ---
def p_protocol_declaration(p):
    '''protocol_declaration : PROTOCOL ID LBRACE RBRACE
                            | PROTOCOL ID'''
    if len(p) == 3:
        p[0] = ('protocol_decl', p[2])
    else:
        p[0] = ('protocol_decl', p[2])
# --- Empty Production ---
def p_empty(p):
    'empty :'
    pass
# --- Error Handling ---
def p_error(p):
    if p:
        print(f"Syntax error at token {p.type} ('{p.value}') on line {p.lineno}")
```

```
    else:
        print("Syntax error at EOF")
# Build the parser
parser = yacc.yacc(write_tables=False)

# --- Simple semantic/type checker ---
from typing import Tuple, List, Dict
def infer_literal_type(value) -> str:
    if isinstance(value, int):
        return 'Int'
    if isinstance(value, float):
        return 'Float'
    if isinstance(value, str):
        if value in ('true', 'false'):
            return 'Bool'
        return 'StringOrID'
    return 'Unknown'

def check(ast: Tuple) -> Tuple[bool, List[str]]:
    errors: List[str] = []
    symbols: Dict[str, str] = {}
    if not ast or ast[0] != 'program':
        errors.append('Invalid AST: expected top-level program')
        return False, errors
    decls = ast[1]
    for decl in decls:
        if not isinstance(decl, tuple):
            continue
        kind = decl[0]
        if kind == 'var_decl':
            _, kindstr, name, type_ann, assign = decl
            declared_type = None
            if type_ann and isinstance(type_ann, tuple) and type_ann[0] ==
'type_annotation':
                declared_type = type_ann[1]
            assigned_type = None
            if assign and isinstance(assign, tuple) and assign[0] == 'assignment':
                expr = assign[1]
                if isinstance(expr, tuple) and expr[0] == 'expression':
                    tag = expr[1]
                    val = expr[2]
                    if tag == 'NUMBER':
                        assigned_type = 'Int' if isinstance(val, int) else 'Float' if
isinstance(val, float) else 'Unknown'
                    elif tag == 'STRING':
                        assigned_type = 'String'
                    elif tag == 'TRUE' or tag == 'FALSE':
```

```
                        assigned_type = 'Bool'
                    elif tag == 'ID':
                        if val in symbols:
                            assigned_type = symbols[val]
                        else:
                            errors.append(f"Undefined identifier '{val}' used in
assignment to '{name}'")
                            assigned_type = 'Unknown'
                    else:
                        assigned_type = 'Unknown'
                else:
                    assigned_type = 'Unknown'
            if declared_type:
                if assigned_type:
                    if assigned_type != declared_type:
                        errors.append(
                            f"Type error: cannot assign value of type
'{assigned_type}' to '{name}' of type '{declared_type}'")
                symbols[name] = declared_type
            else:
                if assigned_type and assigned_type != 'Unknown':
                    symbols[name] = assigned_type
                else:
                    symbols[name] = 'Any'
    ok = len(errors) == 0
    return ok, errors
```

main.py

```
import ply.lex as lex
import ply.yacc as yacc
import sys
from lexer import tokens, lexer, lexer_error
from parser import parser, check as type_check

def read_multiline_input(prompt="Enter Swift code (end with empty line):\n"):
    """Read multiline Swift code until an empty line."""
    print(prompt)
    lines = []
    while True:
        try:
            line = input()
            if not line.strip():  # Empty line ends input
                break
            lines.append(line)
        except EOFError:
            break
```

```
        except KeyboardInterrupt:
            print("\nExiting validator.")
            sys.exit(0)
    return "\n".join(lines)

def validate_swift_code(data):
    """Run lexical, syntax, and semantic validation for Swift subset."""
    global lexer_error
    lexer_error = False  # Reset before every parse
    lexer.lineno = 1
    lexer.input(data)
    result = parser.parse(data, lexer=lexer)
    # Step 1: Lexical check
    if lexer_error:
        print("\n [FAILURE] Lexical error(s) found — invalid Swift syntax.")
        return

    # Step 2: Syntax + Semantic checks
    if result is not None:
        ok, errors = type_check(result)
        if ok:
            print("\n [SUCCESS] Syntax Validation Passed.")
            print("\n--- Abstract Syntax Tree (AST) ---")
            print(result)
        else:
            print("\n[FAILURE] Semantic Validation Failed.")
            for e in errors:
                print(e)
    else:
        print("\n [FAILURE] Syntax Validation Failed.")
if __name__ == '__main__':
    print("Swift Subset Syntax Validator (PLY)")
    print("Supports: Variables, Constants, Structures, Generics, and Protocols")
    print("Type 'quit' or press Ctrl+C to exit.")
    print("-" * 55)
    while True:
        try:
            data = read_multiline_input()
            if data.strip().lower() == 'quit':
                break
            if not data.strip():
                continue
            validate_swift_code(data)
            print("-" * 55)
        except KeyboardInterrupt:
            print("\nExiting validator.")
            break
```

Output:

1)Variables and constants

```
Enter Swift code (end with empty line):

let studentName: String = "Vageesh"
var studentAge: Int = 20


 [SUCCESS] Syntax Validation Passed.
```

```
Enter Swift code (end with empty line):

let a ; int = 12.4

Syntax error at token SEMICOLON (';') on line 1

 [FAILURE] Syntax Validation Failed.
```

```
Enter Swift code (end with empty line):

let pi: Float = 3.14
var isPassed: Bool = true


 [SUCCESS] Syntax Validation Passed.
```

2)Structures.

```
Enter Swift code (end with empty line):

struct Student {
    let name: String = "Vageesh"
    var age: Int = 18
}


 [SUCCESS] Syntax Validation Passed.
```

```
------------------------------------------------------------
 Enter Swift code (end with empty line):

 truct student{
        let name : String = 10
        var age : Int = "abc"
 }

 Syntax error at token ID ('truct') on line 1
 Syntax error at token RBRACE ('}') on line 1

  [FAILURE] Syntax Validation Failed.
```

3)Generics

```
Enter Swift code (end with empty line):

struct Box<T> {
    let item: T
}


 [SUCCESS] Syntax Validation Passed.
```

```
------------------------------------------------------------
Enter Swift code (end with empty line):

struct MyStruct<T: Equatable, U> where T: Protocol {
    let property: T
    var anotherProperty: U
}


 [SUCCESS] Syntax Validation Passed.
```

```
struct MyStruct<T: Equatable, U> where T: Protocol {
    let property: T = 5
    var anotherProperty: U
}


[FAILURE] Semantic Validation Failed.
Type error in struct 'MyStruct': cannot assign value of type 'Int' to 'property' of type 'T'
```