AFLL Mini-Project Report

Team Members:

1. Vismaya Vadana (PES1UG22AM195)
2. Sujay S Katte (PES1UG22AM169)

Brief description about the project:

- The python programs use 'ply' library to tokenise and parse through a given input and check whether it is part of the grammar (in this case, the syntax of R programming language).
- The syntax for the following five constructs is checked:
    1. Variable Declaration
    2. Array Declaration
    3. If Construct
    4. While Loop
    5. Repeat (Do-While) Loop

Context Free Grammar of each construct in R:

1. Variable Declaration:

S -> id <- num; S | id <- num; | id <- float; S | id <- float;
num -> [0-9]+
float -> [+-]?([0-9]*[.])?[0-9]+
id -> [a-zA-Z][a-zA-Z0-9]*

2. Array Declaration:

S -> id <- array(V dim=(N))
V -> c(N), V | c(N), | c(id), V | c(id),
N -> num, N | num
id -> [a-zA-Z][a-zA-Z0-9]*
num -> [0-9]+

3. **If Construct:**

<if-statement> -> if (<expression>) <statement-block>
<expression> -> <literal> | <variable> | <binary-operation> | <unary-operation>
<statement-block> -> { <statement-list> }
<statement-list> -> <statement> ; <statement-list> | <statement>
<statement> -> <assignment> | <if-statement> | <expression>
<assignment> -> <identifier> <- <expression>
<literal> -> <number> | <string> | <boolean>
<variable> -> <identifier>
<identifier> -> [a-zA-Z][a-zA-Z0-9]*
<binary-operation> -> <expression> <binary-operator> <expression>
<unary-operation> -> <unary-operator> <expression>
<number> -> [0-9]+
<string> -> "[a-zA-Z0-9]*"
<boolean> -> TRUE | FALSE
<binary-operator> -> + | - | * | / | ^ | == | != | > | < | >= | <= | && | ||
<unary-operator> -> ! | -

4. **While Loop:**

<while-loop> -> while (<condition>) <statement-block>
<condition> -> <expression>
<statement-block> -> { <statement-list> }
<expression> -> <literal> | <variable> | <binary-operation> | <unary-operation>
<literal> -> <number> | <string> | <boolean>
<variable> -> <identifier>
<binary-operation> -> <expression> <binary-operator> <expression>
<unary-operation> -> <unary-operator> <expression>
<number> -> [0-9]+
<string> -> "[a-zA-Z0-9]*"
<boolean> -> TRUE | FALSE
<identifier> -> [a-zA-Z][a-zA-Z0-9]*
<binary-operator> -> + | - | * | / | ^ | == | != | > | < | >= | <= | && | ||
<unary-operator> -> ! | -

<statement-list> -> <statement> <statement-list> | <statement> | ε
<statement> -> <assignment> ; | <while-loop> | <if-statement> | <expression>
<assignment> -> <identifier> <- <expression>
<if-statement> -> if (<expression>) <statement-block> <else-clause>
<else-clause> -> else <statement-block> | ε

5. **Repeat (Do-While) Loop:**

<repeat-loop> -> repeat <statement-block>
<statement-block> -> { <statement-list> }
<statement-list> -> <statement> ; <statement-list> | <statement> ; | <if-statement> | break ;
<statement> -> <assignment> | <expression>
<assignment> -> <identifier> <- <expression>
<if-statement> -> if (<expression>) <statement-block>
<expression> -> <literal> | <variable> | <binary-operation> | <unary-operation>
<literal> -> <number> | <string> | <boolean>
<variable> -> <identifier>
<binary-operation> -> <expression> <binary-operator> <expression>
<unary-operation> -> <unary-operator> <expression>
<number> -> [0-9]+
<string> -> "[a-zA-Z0-9]*"
<boolean> -> TRUE | FALSE
<identifier> -> [a-zA-Z][a-zA-Z0-9]*
<binary-operator> -> + | - | * | / | ^ | == | != | > | < | >= | <= | && | ||
<unary-operator> -> ! | -

Program and Output of the above constructs in R:

1. **Variable Declaration:**

   Code:

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

flag = 0
```

```
tokens = ('ID', 'NUM', 'ASSIGN', 'SEMICOLON', 'FLOAT')

t_ASSIGN = r'<-'
t_SEMICOLON = r';'

def t_ID(t):
    r'[a-zA-Z][a-zA-Z0-9]*'
    return t

def t_FLOAT(t):
    r'\d+(\.\d+)?'
    t.value = float(t.value)
    return t

def t_NUM(t):
    r'[0-9]+'
    t.value = int(t.value)
    return t

t_ignore = ' \t'

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

lexer = lex.lex()

def p_statement_assign(p):
    '''statement : ID ASSIGN NUM SEMICOLON statement
                 | ID ASSIGN NUM SEMICOLON
                 | ID ASSIGN FLOAT SEMICOLON statement
                 | ID ASSIGN FLOAT SEMICOLON'''

def p_error(p):
    print("Syntax error")
    global flag
    flag = 1

parser = yacc.yacc()

while True:
   flag = 0
   try:
       s = input('Enter the variable declaration to check: ')
   except EOFError:
       break
   if not s:
        flag = 0
```

```
        continue
    result = parser.parse(s)
    if flag == 0:
        print("VALID SYNTAX")
```

Output:

```
PS C:\Users\vadan\Desktop\Vismaya's Folder\PES\Sem 2\AFLL> python -u "c:\Users\vadan\Desktop\Vismaya's Folder\PES\Sem 2\AFLL\variable.py"
Enter the variable declaration to check: a <- 23;
VALID SYNTAX
Enter the variable declaration to check: a <- 23; b <- 2.5;
VALID SYNTAX
Enter the variable declaration to check: a <- 23; b <- 2.5
Syntax error
```

2. Array Declaration:

Code:

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

flag = 0

tokens = ('ID', 'NUM', 'COMMA', 'LPAREN', 'RPAREN', 'ASSIGN', 'array', 'dim', 'c', 'SEMICOLON', 'EQUALS')

t_COMMA = r','
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_ASSIGN = r'<-'
t_array = r'array'
t_dim = r'dim'
t_c = r'c'
t_SEMICOLON = r';'
t_EQUALS = R'='

reserved = {
            'array' : 'array',
            'dim' : 'dim',
            'c' : 'c'
            }

def t_ID(t):
    r'[a-zA-Z][a-zA-Z0-9]*'
    t.type = reserved.get(t.value, 'ID')
    return t

def t_NUM(t):
```

```
    r'[0-9]+'
    t.value = int(t.value)
    return t

t_ignore = ' \t'

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

lexer = lex.lex()

def p_statement(p):
    '''statement : ID ASSIGN array LPAREN vector dim EQUALS LPAREN N RPAREN
RPAREN SEMICOLON'''

def p_vector_recursive(p):
    '''vector : c LPAREN ID RPAREN COMMA vector
              | c LPAREN ID RPAREN COMMA
              | c LPAREN N RPAREN COMMA vector
              | c LPAREN N RPAREN COMMA'''

def p_N_recursive(p):
    '''N : NUM COMMA N
              | NUM'''

def p_error(p):
    print(f"Syntax error at line {p.lineno}, position {p.lexpos}: Unexpected
token '{p.value}'")
    global flag
    flag = 1

parser = yacc.yacc()

while True:
   flag = 0
   try:
       s = input('Enter the array declaration to check: ')
   except EOFError:
       break
   if not s:
        flag = 0
        continue
   result = parser.parse(s)
   if flag == 0:
        print("VALID SYNTAX")
```

Output:

```
PS C:\Users\vadan\Desktop\Vismaya's Folder\PES\Sem 2\AFLL> python -u "c:\Users\vadan\Desktop\Vismaya's Folder\PES\Sem
 2\AFLL\array_1.py"
Enter the array declaration to check: arr <- array(c(1,2,3), c(4,5,6), c(7,8,9), dim=(3,2,1));
VALID SYNTAX
Enter the array declaration to check: arr <- array(c(1,2,3), c(4,5,6), c(7,8,9) dim=(3,2,1));
Syntax error at line 1, position 42: Unexpected token 'dim'
```

3. ## If Construct:

Code:

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

flag = 0

tokens = (
    'NUMBER',
    'STRING',
    'TRUE',
    'FALSE',
    'IDENTIFIER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'EXPONENT',
    'EQUALS',
    'NOT_EQUALS',
    'GREATER',
    'LESS',
    'GREATER_EQUAL',
    'LESS_EQUAL',
    'AND',
    'OR',
    'NOT',
    'LPAREN',
    'RPAREN',
    'LBRACE',
    'RBRACE',
    'SEMICOLON',
    'IF',
    'ASSIGN'
)

t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
```

```
t_DIVIDE = r'/'
t_EXPONENT = r'\^'
t_EQUALS = r'=='
t_NOT_EQUALS = r'!='
t_GREATER = r'>'
t_LESS = r'<'
t_GREATER_EQUAL = r'>='
t_LESS_EQUAL = r'<='
t_AND = r'&&'
t_OR = r'\|\|'
t_NOT = r'!'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_SEMICOLON = r';'
t_TRUE = r'TRUE'
t_FALSE = r'FALSE'
t_STRING = r'"[a-zA-Z0-9]*"'
t_NUMBER = r'[0-9]+'
t_IDENTIFIER = r'[a-zA-Z][a-zA-Z0-9]*'
t_IF = r'if'
t_ASSIGN = r'<-'

reserved = {
    'if': 'IF',
    'TRUE': 'TRUE',
    'FALSE': 'FALSE'
}

t_ignore = ' \t\n'

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

def t_IDENTIFER(t):
    r'[a-zA-Z][a-zA-Z0-9]*'
    t.type = reserved.get(t.value, 'IDENTIFIER')
    return t

def p_if_statement(p):
    '''if_statement : IF LPAREN expression RPAREN statement_block'''

def p_expression_literal(p):
    '''expression : literal'''

def p_expression_variable(p):
```

```
    '''expression : variable'''

def p_expression_binary_operation(p):
    '''expression : binary_operation'''

def p_expression_unary_operation(p):
    '''expression : unary_operation'''

def p_literal(p):
    '''literal : NUMBER
               | STRING
               | TRUE
               | FALSE'''

def p_variable(p):
    'variable : IDENTIFIER'

def p_binary_operation(p):
    '''binary_operation : expression binary_operator expression'''

def p_unary_operation(p):
    '''unary_operation : unary_operator expression'''

def p_binary_operator(p):
    '''binary_operator : PLUS
                       | MINUS
                       | TIMES
                       | DIVIDE
                       | EXPONENT
                       | EQUALS
                       | NOT_EQUALS
                       | GREATER
                       | LESS
                       | GREATER_EQUAL
                       | LESS_EQUAL
                       | AND
                       | OR'''

def p_assignment(p):
    '''assignment : IDENTIFIER ASSIGN expression'''

def p_unary_operator(p):
    '''unary_operator : NOT
                      | MINUS'''

def p_statement_block(p):
    'statement_block : LBRACE statement_list RBRACE'
```

```
def p_statement_list(p):
    '''statement_list : statement SEMICOLON statement_list
                      | statement SEMICOLON'''

def p_statement(p):
    '''statement : if_statement
                 | expression
                 | assignment'''

def p_error(p):
    print(f"Syntax error at line {p.lineno}, position {p.lexpos}")
    global flag
    flag = 1

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

while True:
   flag = 0
   try:
       s = input('Enter the if-construct to check: ')
   except EOFError:
       break
   if not s:
        flag = 0
        continue
   result = parser.parse(s)
   if flag == 0:
        print("VALID SYNTAX")
```

Output:

```
PS C:\Users\vadan\Desktop\Vismaya's Folder\PES\Sem 2\AFLL> python -u "c:\Users\vadan\Desktop\Vismaya's Folder\PES\Sem
 2\AFLL\if.py"
Enter the if-construct to check: if(x<=0){x1 <- 4; x2 <- x1*4;}
VALID SYNTAX
Enter the if-construct to check: if(x<=0){x1 <- 4; x2 <- x1*4}
Syntax error at line 1, position 28
Enter the if-construct to check: if(x){x1 <- 4; x2 <- x1*4;}
VALID SYNTAX
Enter the if-construct to check: if(TRUE){x1 <- 4; x2 <- x1*4;}
VALID SYNTAX
```

4. **While Loop:**

Code:

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

```
flag = 0

tokens = (
    'NUMBER', 'STRING', 'TRUE', 'FALSE', 'IDENTIFIER',
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'EXPONENT', 'EQUALS',
    'NOTEQUALS', 'GREATER', 'LESS', 'GREATEREQ', 'LESSEQ', 'AND', 'OR',
    'NOT', 'LPAREN', 'RPAREN', 'LBRACE', 'RBRACE', 'SEMICOLON', 'IF', 'ELSE',
'ASSIGN', 'WHILE'
)

t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_EXPONENT = r'\^'
t_EQUALS = r'=='
t_NOTEQUALS = r'!='
t_GREATER = r'>'
t_LESS = r'<'
t_GREATEREQ = r'>='
t_LESSEQ = r'<='
t_AND = r'&&'
t_OR = r'\|\|'
t_NOT = r'!'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_SEMICOLON = r';'
t_IF = r'if'
t_ELSE = r'else'
t_TRUE = r'TRUE'
t_FALSE = r'FALSE'
t_STRING = r'"[a-zA-Z0-9]*"'
t_NUMBER = r'[0-9]+'
t_IDENTIFIER = r'[a-zA-Z][a-zA-Z0-9]*'
t_ASSIGN = r'<-'
t_WHILE = r'while'
t_ignore = ' \t\n'

reserved = {
    'if': 'IF',
    'else': 'ELSE',
    'while': 'WHILE',
    'TRUE': 'TRUE',
    'FALSE': 'FALSE'
}
```

```
def t_IDENTIFER(t):
    r'[a-zA-Z][a-zA-Z0-9]*'
    t.type = reserved.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 index {t.lexpos}")
    t.lexer.skip(1)

def p_while_loop(p):
    'while_loop : WHILE LPAREN condition RPAREN statement_block'
    p[0] = ('while-loop', p[3], p[5])

def p_condition(p):
    'condition : expression'
    p[0] = ('condition', p[1])

def p_statement_block(p):
    'statement_block : LBRACE statement_list RBRACE'
    p[0] = ('statement-block', p[2])

def p_expression(p):
    '''expression : literal
                  | variable
                  | binary_operation
                  | unary_operation'''
    p[0] = ('expression', p[1])

def p_literal(p):
    '''literal : number
                  | STRING
                  | boolean'''
    p[0] = ('literal', p[1])

def p_variable(p):
    '''variable : IDENTIFIER'''
    p[0] = ('variable', p[1])

def p_binary_operation(p):
    '''binary_operation : expression binary_operator expression'''
    p[0] = ('binary-operation', p[1], p[2], p[3])

def p_unary_operation(p):
    '''unary_operation : unary_operator expression'''
```

```
    p[0] = ('unary-operation', p[1], p[2])

def p_number(p):
    '''number : NUMBER'''
    p[0] = ('number', p[1])

def p_boolean(p):
    '''boolean : TRUE
               | FALSE'''
    p[0] = ('boolean', p[1])

def p_identifier(p):
    '''identifier : IDENTIFIER'''
    p[0] = ('identifier', p[1])

def p_binary_operator(p):
    '''binary_operator : PLUS
                       | MINUS
                       | TIMES
                       | DIVIDE
                       | EXPONENT
                       | EQUALS
                       | NOTEQUALS
                       | GREATER
                       | LESS
                       | GREATEREQ
                       | LESSEQ
                       | AND
                       | OR'''
    p[0] = ('binary-operator', p[1])

def p_unary_operator(p):
    '''unary_operator : NOT
                      | MINUS'''
    p[0] = ('unary-operator', p[1])

def p_statement_list(p):
    '''statement_list : statement statement_list
                      | statement
                      | empty'''
    if len(p) == 4:
        p[0] = ('statement-list', p[1], p[3])
    else:
        p[0] = ('statement-list', p[1])

def p_statement(p):
    '''statement : assignment SEMICOLON
                 | while_loop
```

```
                    | if_statement
                    | expression '''
    p[0] = ('statement', p[1])

def p_assignment(p):
    '''assignment : identifier ASSIGN expression'''
    p[0] = ('assignment', p[1], p[3])

def p_if_statement(p):
    '''if_statement : IF LPAREN expression RPAREN statement_block
else_clause'''
    p[0] = ('if-statement', p[3], p[5], p[6])

def p_else_clause(p):
    '''else_clause : ELSE statement_block
                   | empty'''
    if len(p) == 3:
        p[0] = ('else-clause', p[2])
    else:
        p[0] = ('else-clause', None)

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

def p_error(p):
    print(f"Syntax error at line {p.lineno}, position {p.lexpos}: Unexpected
token '{p.value}'")
    global flag
    flag = 1

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

while True:
   flag = 0
   try:
       s = input('Enter the while loop to check: ')
   except EOFError:
       break
   if not s:
        flag = 0
        continue
   result = parser.parse(s)
   if flag == 0:
        print("VALID SYNTAX")
```

Output:

```
PS C:\Users\vadan\Desktop\Vismaya's Folder\PES\Sem 2\AFLL> python -u "c:\Users\vadan\Desktop\Vismaya's Folder\PES\Sem
 2\AFLL\tempCodeRunnerFile.py"
Enter the while loop to check: while(i>=0){num <- 6;}
VALID SYNTAX
Enter the while loop to check: while(i>=0){num <- 6; while(i>=4){num <- num*2;}}
VALID SYNTAX
Enter the while loop to check: while(i>=0){num <- 6; while(i>=4){num <- num*2}}
Syntax error at line 1, position 46: Unexpected token '}'
```

5. ## Repeat (Do-While) Loop:

Code:

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

flag = 0

tokens = (
    'REPEAT', 'LEFT_BRACE', 'RIGHT_BRACE',
    'LEFT_PAREN', 'RIGHT_PAREN', 'SEMICOLON',
    'BREAK', 'IF',
    'ASSIGN', 'IDENTIFIER',
    'NUMBER', 'STRING', 'TRUE', 'FALSE',
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'POWER',
    'EQUALS', 'NOT_EQUALS', 'GREATER', 'LESS',
    'GREATER_EQUAL', 'LESS_EQUAL', 'AND', 'OR',
    'NOT'
)

reserved = {
            'repeat' : 'REPEAT',
            'break' : 'BREAK',
            'if' : 'IF',
            'TRUE' : 'TRUE',
            'FALSE' : 'FALSE'
}

t_REPEAT = r'repeat'
t_LEFT_BRACE = r'{'
t_RIGHT_BRACE = r'}'
t_LEFT_PAREN = r'\('
t_RIGHT_PAREN = r'\)'
t_SEMICOLON = r';'
t_BREAK = r'break'
t_IF = r'if'
t_ASSIGN = r'<-'
t_NUMBER = r'[0-9]+'
t_STRING = r'"[a-zA-Z0-9]*"'
```

```
t_TRUE = r'TRUE'
t_FALSE = r'FALSE'
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_POWER = r'\^'
t_EQUALS = r'=='
t_NOT_EQUALS = r'!='
t_GREATER = r'>'
t_LESS = r'<'
t_GREATER_EQUAL = r'>='
t_LESS_EQUAL = r'<='
t_AND = r'&&'
t_OR = r'\|\|'
t_NOT = r'!'

t_ignore = ' \t\n'

def t_IDENTIFER(t):
    r'[a-zA-Z][a-zA-Z0-9]*'
    t.type = reserved.get(t.value, 'IDENTIFIER')
    return t

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

def p_repeat_loop(p):
    '''repeat_loop : REPEAT statement_block'''

def p_statement_block(p):
    '''statement_block : LEFT_BRACE statement_list RIGHT_BRACE'''

def p_statement_list(p):
    '''statement_list : statement SEMICOLON statement_list
                      | statement SEMICOLON
                      | if_statement
                      | BREAK SEMICOLON'''

def p_statement(p):
    '''statement : assignment
                 | expression'''

def p_assignment(p):
    '''assignment : IDENTIFIER ASSIGN expression'''

def p_if_statement(p):
```

```
    '''if_statement : IF LEFT_PAREN expression RIGHT_PAREN statement_block'''

def p_expression(p):
    '''expression : literal
                  | variable
                  | binary_operation
                  | unary_operation'''

def p_literal(p):
    '''literal : NUMBER
               | STRING
               | TRUE
               | FALSE'''

def p_variable(p):
    '''variable : IDENTIFIER'''

def p_binary_operation(p):
    '''binary_operation : expression binary_operator expression'''

def p_unary_operation(p):
    '''unary_operation : unary_operator expression'''

def p_binary_operator(p):
    '''binary_operator : PLUS
                       | MINUS
                       | TIMES
                       | DIVIDE
                       | POWER
                       | EQUALS
                       | NOT_EQUALS
                       | GREATER
                       | LESS
                       | GREATER_EQUAL
                       | LESS_EQUAL
                       | AND
                       | OR'''

def p_unary_operator(p):
    '''unary_operator : NOT
                      | MINUS'''

def p_error(p):
    print(f"Syntax error at line {p.lineno}, position {p.lexpos}")
    global flag
    flag = 1

lexer = lex.lex()
```

```
parser = yacc.yacc()

while True:
   flag = 0
   try:
       s = input('Enter the repeat(do-while) loop to check: ')
   except EOFError:
       break
   if not s:
        flag = 0
        continue
   result = parser.parse(s)
   if flag == 0:
        print("VALID SYNTAX")
```

Output:

```
PS C:\Users\vadan\Desktop\Vismaya's Folder\PES\Sem 2\AFLL> python -u "c:\Users\vadan\Desktop\Vismaya's Folder\PES\Sem
 2\AFLL\tempCodeRunnerFile.py"
Enter the repeat(do-while) loop to check: repeat{num <- 6; if(x>0){num <- 7; break;}}
VALID SYNTAX
Enter the repeat(do-while) loop to check: repeat{num <- 6; if(x){num <- 7; break;}}
VALID SYNTAX
Enter the repeat(do-while) loop to check: repeat{if(x){num <- 7; break;}}
VALID SYNTAX
Enter the repeat(do-while) loop to check: repeat{num <- num + 1; if(x){num <- 7; break;}}
VALID SYNTAX
```