**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]\*

1. **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]+

1. **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> -> ! | -

1. **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

1. **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:**

A screenshot of a computer

Description automatically generated

1. **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

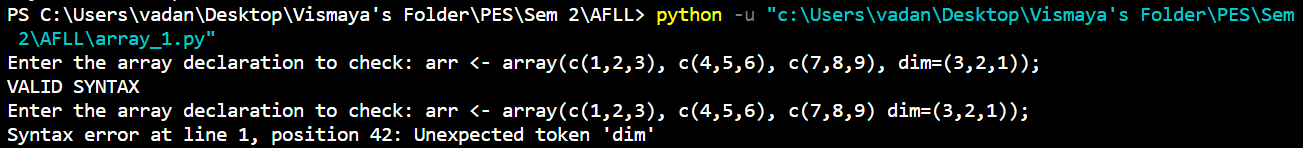
        continue

   result = parser.parse(s)

   if flag == 0:

        print("VALID SYNTAX")

**Output:**

1. **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:**

1. A screenshot of a computer program

   Description automatically generated**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:**

1. A computer screen with text and numbers

   Description automatically generated with medium confidence**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:**

A screenshot of a computer code

Description automatically generated