AFLL MINI PROJECT

Name: Snigdha SV
SRN: PES1UG22CS600
Name: Sri Lakshmi Mothkur
SRN: PES1UG22CS608
'J' Section

Language Chosen: Javascript

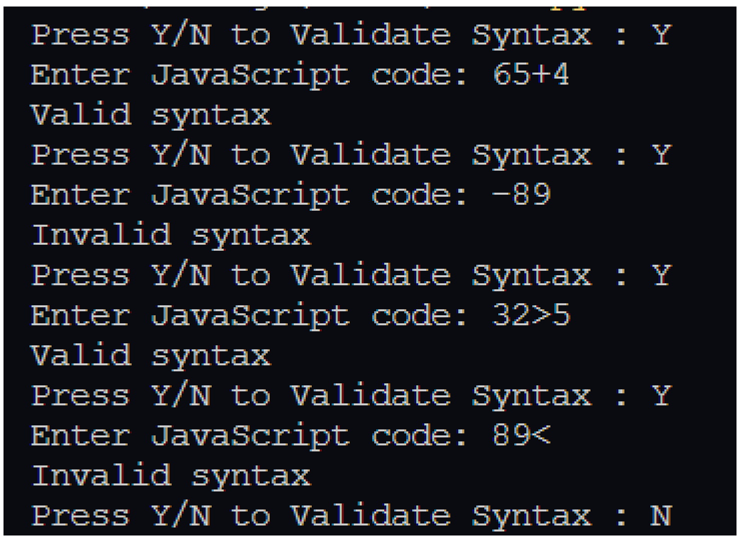
Constructs:

1. Arithmetic and comparison operators
2. Arrays
3. If statement
4. String operations
5. While loop

CFG:

1. **Arithmetic and conversion operators**

```
expression -> expression + term
          | expression - term
          | term

term -> term * factor
    | term / factor
    | factor

factor -> number
      | ( expression )
      | variable
      | expression == expression
      | expression != expression
      | expression < expression
      | expression <= expression
      | expression > expression
      | expression >= expression
```



```
Press Y/N to Validate Syntax : Y
Enter JavaScript code: 65+4
Valid syntax
Press Y/N to Validate Syntax : Y
Enter JavaScript code: -89
Invalid syntax
Press Y/N to Validate Syntax : Y
Enter JavaScript code: 32>5
Valid syntax
Press Y/N to Validate Syntax : Y
Enter JavaScript code: 89<
Invalid syntax
Press Y/N to Validate Syntax : N
```

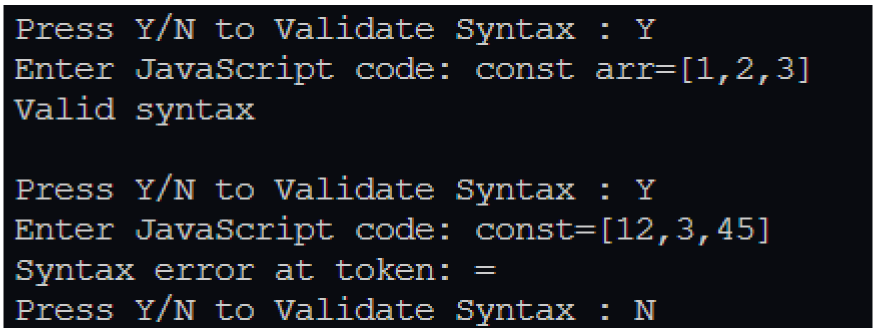
2. **Arrays**

   array -> [elements]

   elements -> element

   | element , elements

   element -> expression



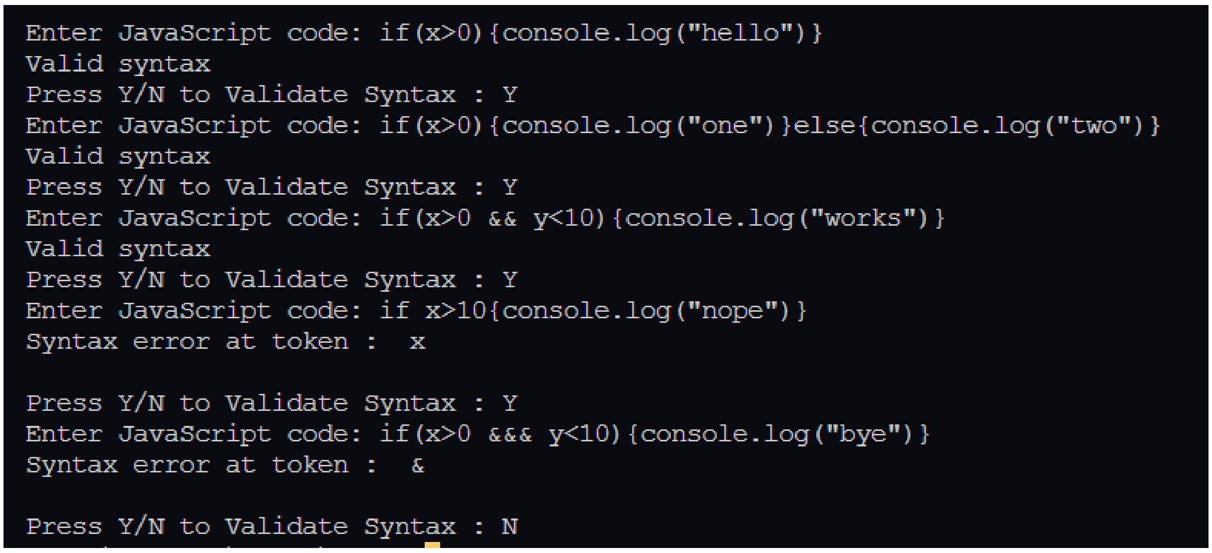
```
Press Y/N to Validate Syntax : Y
Enter JavaScript code: const arr=[1,2,3]
Valid syntax

Press Y/N to Validate Syntax : Y
Enter JavaScript code: const=[12,3,45]
Syntax error at token: =
Press Y/N to Validate Syntax : N
```

3. **If statement**

   ifStatement -> if (condition) statement

   | if (condition) statement else statement

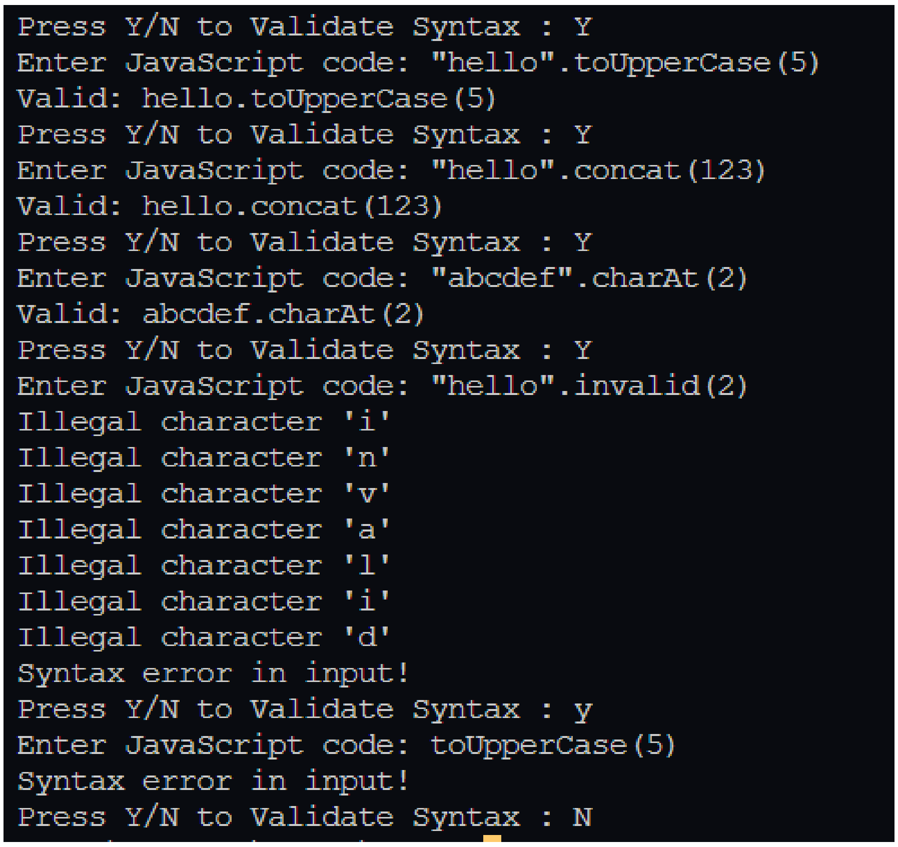
   condition -> expression


```
Enter JavaScript code: if(x>0){console.log("hello")}
Valid syntax
Press Y/N to Validate Syntax : Y
Enter JavaScript code: if(x>0){console.log("one")}else{console.log("two")}
Valid syntax
Press Y/N to Validate Syntax : Y
Enter JavaScript code: if(x>0 && y<10){console.log("works")}
Valid syntax
Press Y/N to Validate Syntax : Y
Enter JavaScript code: if x>10{console.log("nope")}
Syntax error at token :  x

Press Y/N to Validate Syntax : Y
Enter JavaScript code: if(x>0 &&& y<10){console.log("bye")}
Syntax error at token :  &

Press Y/N to Validate Syntax : N
```

4. **String operations**

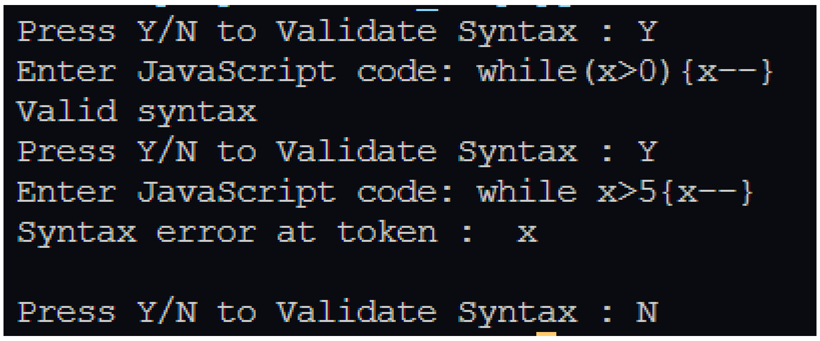
    stringOperation -> stringVariable = stringExpression
                    | stringVariable += stringExpression

    stringExpression -> "string"
                    | stringVariable
                    | stringExpression + stringExpression


```
Press Y/N to Validate Syntax : Y
Enter JavaScript code: "hello".toUpperCase(5)
Valid: hello.toUpperCase(5)
Press Y/N to Validate Syntax : Y
Enter JavaScript code: "hello".concat(123)
Valid: hello.concat(123)
Press Y/N to Validate Syntax : Y
Enter JavaScript code: "abcdef".charAt(2)
Valid: abcdef.charAt(2)
Press Y/N to Validate Syntax : Y
Enter JavaScript code: "hello".invalid(2)
Illegal character 'i'
Illegal character 'n'
Illegal character 'v'
Illegal character 'a'
Illegal character 'l'
Illegal character 'i'
Illegal character 'd'
Syntax error in input!
Press Y/N to Validate Syntax : y
Enter JavaScript code: toUpperCase(5)
Syntax error in input!
Press Y/N to Validate Syntax : N
```

5. **While loop**
   whileLoop -> while (condition) statement

   condition -> expression



```
Press Y/N to Validate Syntax : Y
Enter JavaScript code: while(x>0){x--}
Valid syntax
Press Y/N to Validate Syntax : Y
Enter JavaScript code: while x>5{x--}
Syntax error at token :  x

Press Y/N to Validate Syntax : N
```

CODES:

1. **Arithmetic and conversion operators**

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

# List of token names
tokens = (
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
    'COMPARISON_OPERATOR',
)

# Declare precedence and associativity
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('left', 'COMPARISON_OPERATOR'),
)
```

```python
# Regular expression rules for simple tokens
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# Additional tokens
t_COMPARISON_OPERATOR = r'==|===|!=|!==|>|>=|<|<='

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

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore = ' \t'

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

# Build the lexer
lexer = lex.lex()

# Parser
def p_expression_binop(p):
    '''expression : expression PLUS term
                  | expression MINUS term
                  | expression TIMES term
                  | expression DIVIDE term'''
    p[0] = (p[2], p[1], p[3])

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_binop(p):
    '''term : term PLUS factor
            | term MINUS factor
            | term TIMES factor
            | term DIVIDE factor'''
    p[0] = (p[2], p[1], p[3])

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Additional grammar rules for new constructs
def p_expression_comparison(p):
    'expression : expression COMPARISON_OPERATOR expression'
    p[0] = ('comparison', p[2], p[1], p[3])

def p_error(p):
    raise SyntaxError("Invalid syntax")
# Build the parser
parser = yacc.yacc()

# Test the parser
```

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

2. Arrays

```
from ply import lex
from ply import yacc

# List of token names
tokens = (
    'TYPE',
    'IDENTIFIER',
    'ASSIGN',
    'LBRACKET',
    'RBRACKET',
    'COMMA',
    'NUMBER',
    'STRING',
    'BOOLEAN',
    'NULL',
    '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
    return t

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

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

def t_NULL(t):
    r'null'
    return t

t_ignore = ' \t'

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

```python
    t.lexer.skip(1)

lexer = lex.lex()

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

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'''  # 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
             | NULL
             | 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 JavaScript code: ')
        except EOFError:
            break
        if not s:
            continue
        result = parser.parse(s)
        if(result=="Valid") :
            print("Valid syntax \n")
```

3. If statement

```
from ply import lex
from ply import yacc
```

```
tokens = (
    'IF',
    'ELSE',
    'TYPE',
    'IDENTIFIER',
    'NUMBER',
    'STRING',
    'ASSIGN',
    'SHORTHAND',
    'LOGICAL',
    'ULOGICAL',
    'BITWISE',
    'UBITWISE',
    'COMPARISON',
    'LPAREN',
    'RPAREN',
    'LBRACKET',
    'RBRACKET',
    'C_LOG',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'NULL',
    'BOOLEAN',
    'COMMA',
)

t_IDENTIFIER = r'[a-zA-Z_][a-zA-Z0-9_]*'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACKET = r'\{'
t_RBRACKET = r'\}'
t_ASSIGN = r'\='
t_PLUS = r'\+'
t_MINUS = r'-'
```

```python
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_COMMA = r','

t_ignore = ' \t'

def t_IF(t):
    r'''if'''
    return t

def t_ELSE(t):
    r'''else'''
    return t

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

def t_C_LOG(t):
    r'''console\.log'''
    return t

def t_NUMBER(t):
    r'''\d+\.\d+|\d+'''
    return t

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

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

def t_NULL(t):
    r'''null'''
```

```python
def t_LOGICAL(t):
    r'''\&\& | \|\|'''
    return t

def t_SHORTHAND(t):
    r'''\^\= | \&\= | \|\= | \~\= | \>\>\= | \<\<\= | \>\>\>\= | \+\= |
\-\= | \*\= | \/\='''
    return t

def t_ULOGICAL(t):
    r'''\!'''
    return t

def t_BITWISE(t):
    r'''\& | \| | \^ | \>\> | \<\< | \>\>\>'''
    return t

def t_UBITWISE(t):
    r'''\~'''
    return t

def t_COMPARISON(t):
    r'''\=\=| \=\=\= | \>|\<|\>\=|\<\=|\!\='''
    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]}'")
    t.lexer.skip(1)

lexer = lex.lex()



def p_if_else(p):
    '''if_else : IF LPAREN expressions RPAREN LBRACKET statements
RBRACKET ELSE LBRACKET statements RBRACKET
               | IF LPAREN expressions RPAREN LBRACKET statements
RBRACKET'''
    p[0] = "Valid"

def p_expressions(p):
    '''expressions : expression LOGICAL expression
                   | expression'''

def p_expression(p):
    '''expression : expression COMPARISON expression
                  | expression BITWISE expression
                  | expression PLUS expression
                  | expression MINUS expression
                  | expression DIVIDE expression
                  | expression TIMES expression
                  | ULOGICAL expression
                  | UBITWISE expression
                  | LPAREN expression RPAREN
                  | IDENTIFIER
                  | NUMBER
                  | BOOLEAN
                  | NULL
                  | STRING'''

def p_statements(p):
    '''statements : statement statements
                  | statement
                  |'''

def p_statement(p):
    '''statement : assign_stmt
                 | c_log_stmt
                 | IDENTIFIER'''

def p_c_log_stmt(p):
    '''c_log_stmt : C_LOG LPAREN args RPAREN'''

def p_args(p):
    '''args : expression
            | expression COMMA args'''

def p_assign_stmt(p):
    '''assign_stmt : TYPE IDENTIFIER ASSIGN expressions
                   | TYPE IDENTIFIER ASSIGN expression COMMA
multiple_assign
                   | IDENTIFIER ASSIGN expressions
                   | IDENTIFIER SHORTHAND expressions'''

def p_multiple_assign(p):
    '''multiple_assign : IDENTIFIER ASSIGN expressions
                       | IDENTIFIER ASSIGN expressions COMMA
multiple_assign'''

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

parser = yacc.yacc()

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

4. **String Operators**

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

# List of token names
tokens = (
    'STRING_METHOD',
    'STRING_LITERAL',
    'DOT',
    'LPAREN',
    'RPAREN',
    'NUMBER',
)

# Regular expressions for tokens
t_DOT = r'\.'
t_LPAREN = r'\('
t_RPAREN = r'\)'

def t_STRING_METHOD(t):
    r'charAt|concat|includes|indexOf|substring|toLowerCase|toUpperCase'
    return t

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

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

```

```python
# Ignored characters (whitespace)
t_ignore = ' \t'

# Error handling rule
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_expression_string_method(p):
    'expression : STRING_LITERAL DOT STRING_METHOD LPAREN NUMBER
RPAREN'
    print(f'Valid: {p[1]}.{p[3]}({p[5]})')

def p_error(p):
    print("Syntax error in input!")

# Build the parser
parser = yacc.yacc()

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

5. **While Loop**

```
from ply import lex
from ply import yacc

tokens = (
    'WHILE',
    'TYPE',
    'IDENTIFIER',
    'NUMBER',
    'STRING',
    'ASSIGN',
    'SHORTHAND',
    'LOGICAL',
    'ULOGICAL',
    'BITWISE',
    'UBITWISE',
    'COMPARISON',
    'LPAREN',
    'RPAREN',
    'LBRACKET',
    'RBRACKET',
    'C_LOG',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'NULL',
    'BOOLEAN',
    'COMMA',
    'INCREMENT',
    'DECREMENT',
)

t_IDENTIFIER = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

```
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACKET = r'\{'
t_RBRACKET = r'\}'
t_ASSIGN = r'\='
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_COMMA = r','
t_INCREMENT = r'\+\+'
t_DECREMENT = r'\-\-'

t_ignore = ' \t'

def t_WHILE(t):
    r'''while'''
    return t

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

def t_C_LOG(t):
    r'''console\.log'''
    return t

def t_NUMBER(t):
    r'''\d+\.\d+|\d+'''
    return t

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

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

```python
    return t

def t_NULL(t):
    r'''null'''
    return t

def t_LOGICAL(t):
    r'''\&\&|\|\|'''
    return t

def t_SHORTHAND(t):

r'''\^\=|\&\=|\|\=|\~\=|\>\>\=|\<\<\=|\>\>\>\=|\+\=|\-\=|\*\=|\/\='''
    return t

def t_ULOGICAL(t):
    r'''\!'''
    return t

def t_BITWISE(t):
    r'''\&|\||\^|\>\>|\<\<|\>\>\>'''
    return t

def t_UBITWISE(t):
    r'''\~'''
    return t

def t_COMPARISON(t):
    r'''\=\=|\=\=\=|\>|\<|\>\=|\<\=|\!\='''
    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]}'")
    t.lexer.skip(1)

lexer = lex.lex()


def p_statement_while(p):
    '''statement_while : WHILE LPAREN expressions RPAREN LBRACKET
statements RBRACKET'''
    p[0] = 'Valid'

def p_expressions(p):
    '''expressions : expression
                   | expression LOGICAL expressions'''

def p_expression(p):
    '''expression : IDENTIFIER
                  | NUMBER
                  | BOOLEAN
                  | NULL
                  | STRING
                  | expression BITWISE expression
                  | UBITWISE expression
                  | expression COMPARISON expression
                  | expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression
                  | LPAREN expression RPAREN
                  | ULOGICAL expression'''

def p_statements(p):
    '''statements : statement
                  | statement statements'''

def p_statement(p):
    '''statement : assign_stmt
                 | c_log_stmt
                 | statement_while'''

def p_c_log_stmt(p):
    '''c_log_stmt : C_LOG LPAREN args RPAREN'''

def p_args(p):
    '''args : expression
            | expression COMMA args'''

def p_assign_stmt(p):
    '''assign_stmt : TYPE IDENTIFIER ASSIGN expressions
                   | TYPE IDENTIFIER ASSIGN expression COMMA
multiple_assign
                   | IDENTIFIER ASSIGN expressions
                   | IDENTIFIER SHORTHAND expressions
                   | IDENTIFIER INCREMENT
                   | IDENTIFIER DECREMENT
                   | INCREMENT IDENTIFIER
                   | DECREMENT IDENTIFIER'''

def p_multiple_assign(p):
    '''multiple_assign : IDENTIFIER ASSIGN expressions
                       | IDENTIFIER ASSIGN expressions COMMA
multiple_assign'''

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

parser = yacc.yacc()

while True:
    try:
        check = input("Press Y/N to Validate Syntax : ")
        if(check=='N') :
```

```python
            exit(0)
        else :
            s = input('Enter JavaScript code: ')
    except EOFError:
        break
    if not s:
        continue
    result = parser.parse(s)
    if(result=="Valid") :
        print("Valid syntax")
```