# UNIVERSITY OF NAIROBI

## FACULTY OF SCIENCE AND TECHNOLOGY

## DEPARTMENT OF COMPUTING AND INFORMATICS

CSC 326: COMPILER CONSTRUCTION

# ICG FOR OUR MINI LANGUAGE USING PLY TOOL


Done by Group 17:

| | |
|---|---|
| P15/137032/2019 | Hamud Abdulrahman Abeid |
| P15/1894/2020 | Emmanuel Mbithi Muthiani |
| P15/1915/2020 | Mugo Staicy Nelima Muthoni |

## SOURCE CODE FOR THE LEXER IN PYTHON

```
import ply.lex as lex



keywords = {

   'if'     : 'IF',

   'else'   : 'ELSE',

   'while'  : 'WHILE',

   'print'  : 'PRINT'

}



# Define tokens

tokens = [

    'ASSIGN',

    'PLUS',

    'MINUS',

    'MULT',

    'DIV',

    'LPAREN',

    'RPAREN',

    'LBRACE',

    'RBRACE',

    'ID',
```

```
    'INT',
    'FLOAT',
    'EOL',
    'LT',
    'GT',
    'LE',
    'GE',
    'EQ',
    'NE',
    'STRING',
    'SINGLEQUOTE',
    'DOUBLEQUOTE'
] + list(keywords.values())



# Define regular expression rules for tokens
t_ASSIGN = r'='
t_PLUS = r'\+'
t_MINUS = r'-'
t_MULT = r'\*'
t_DIV = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACE = r'{'
t_RBRACE = r'}'
t_FLOAT = r'\d+\.\d+'
t_INT = r'\d+'
t_EOL = r';'
```

```
t_LT = r'<'

t_GT = r'>'

t_LE = r'<='

t_GE = r'>='

t_EQ = r'=='

t_NE = r'!='

t_DOUBLEQUOTE = r'"'

t_SINGLEQUOTE = r"'"



# Define rule to check for keywords

def t_ID(t):

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

    t.type = keywords.get(t.value, 'ID')

    return t



# Define rule for string literals

def t_STRING(t):

    r'"([^\\\n]|(\\.))*?"|\'([^\\\n]|(\\.))*?\''

    return t





# Define ignored characters (whitespace)

t_ignore = ' \t'



# Define new line tracking rule

def t_newline(t):

    r'\n+'
```

```
    t.lexer.lineno += len(t.value)



# Define error handling rule

def t_error(t):

    print(f"SyntaxError: Invalid token '{t.value[0]}' at line {t.lineno}")

    t.lexer.skip(1)



# Build lexer

lexer = lex.lex()



# Test lexer with sample input

data = '''

    x = 42.4;

    if (x > 40){

        print(x);

        a = 3 + 1 / 5;

        print("hello, world");

    }

    else {

        print('a');

    }

'''



# Run the lexer

lexer.input(data)



# Tokenize
```

```
for token in lexer:

    print("Token[",token.type,"]        [", token.value,"]")
```

## SOURCE CODE FOR THE PARSER IN PYTHON

```
import ply.yacc as yacc


# Get the token list from the lexer

from lexer3 import tokens


# Define the grammar rules

def p_statement_assign(p):

    'statement : ID ASSIGN expression EOL'

    p[0] = ('ASSIGN', p[3], "to", p[1])


def p_expression_binop(p):

    '''expression : expression PLUS expression

                  | expression MINUS expression

                  | expression MULT expression

                  | expression DIV expression

                  | expression LT expression

                  | expression GT expression

                  | expression LE expression

                  | expression GE expression

                  | expression EQ expression

                  | expression NE expression'''

    p[0] = (p[1], p[2], p[3])
```

```
def p_expression_group(p):
    'expression : LPAREN expression RPAREN'
    p[0] = p[2]


def p_expression_number(p):
    '''expression : INT
                  | FLOAT
                  | ID'''
    p[0] = p[1]


def p_statement_if(p):
    'statement : IF expression LBRACE statements RBRACE'
    p[0] = ('IF', p[2], p[4])


def p_statement_else(p):
    'statement : IF expression LBRACE statements RBRACE ELSE LBRACE
statements RBRACE'
    p[0] = ('IF', p[2], p[4], 'ELSE', p[8])


def p_statement_while(p):
    'statement : WHILE expression LBRACE statements RBRACE'
    p[0] = ('WHILE', p[2], p[4])


def p_statement_print(p):
    '''statement : PRINT expression EOL
                 | PRINT STRING EOL
                 | PRINT SINGLEQUOTE STRING SINGLEQUOTE EOL
```

```
                | PRINT DOUBLEQUOTE STRING DOUBLEQUOTE EOL'''
    if len(p) == 4:
        p[0] = ('PRINT', p[2], p[3])
    else:
        p[0] = ('PRINT', p[2], p[3])


def p_expression_string(p):
    '''expression : STRING'''
    p[0] = p[1]


def p_statements(p):
    '''statements : statement
                  | statements statement'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[1].append(p[2])
        p[0] = p[1]


def p_program(p):
    'statement : statement statement'
    p[0] = (p[1], p[2])


def p_error(p):
    print(f"SyntaxError: Incorrect syntax at line {p.lineno-1}")


def p_empty(p):
```

```
    'empty :'
    pass

# Build parser
parser = yacc.yacc()

# Test parser with sample input
data = '''
    x = 42.4;
    if (x > 40){
        print(x);
        a = 3 + 1 / 5;
        print("hello, world");
    }
    else {
        print('a');
    }
'''

# Parse the input
result = parser.parse(data)

# Print the result
print(result)
```

## SOURCE CODE FOR THE ICG IN PYTHON

```
import ply.yacc as yacc


# Get the token list from the lexer
from lexer3 import tokens
from parser_1 import result


# Global variable to generate unique temporary variables
TEMP_COUNT = 0


# Define the intermediate code generator
def p_statement_assign(p):
    'statement : ID ASSIGN expression EOL'
    p[0] = ('ASSIGN', p[3], "to", p[1])
    global TEMP_COUNT
    temp = f"t{TEMP_COUNT}"
    TEMP_COUNT += 1
    print(f"{temp} = {p[3]}")
    print(f"{p[1]} = {temp}")


def p_expression_binop(p):
    '''expression : expression PLUS expression
                  | expression MINUS expression
                  | expression MULT expression
                  | expression DIV expression
                  | expression LT expression
                  | expression GT expression
```

```
                  | expression LE expression
                  | expression GE expression
                  | expression EQ expression
                  | expression NE expression'''
    p[0] = (p[1], p[2], p[3])
    global TEMP_COUNT
    temp = f"t{TEMP_COUNT}"
    TEMP_COUNT += 1
    print(f"{temp} = {p[1]} {p[2]} {p[3]}")
    p[0] = temp


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


def p_expression_number(p):
    '''expression : INT
                  | FLOAT
                  | ID'''
    p[0] = p[1]


def p_statement_if(p):
    'statement : IF expression LBRACE statements RBRACE'
    p[0] = ('IF', p[2], p[4])
    print(f"if {p[2]} goto L1")
    print("goto L2")
    print("L1:")
```

```
    for stmt in p[4]:
        print(stmt)
    print("L2:")


def p_statement_else(p):
    'statement : IF expression LBRACE statements RBRACE ELSE LBRACE
statements RBRACE'
    p[0] = ('IF', p[2], p[4], 'ELSE', p[8])
    print(f"if {p[2]} goto L1")
    print("goto L2")
    print("L1:")
    for stmt in p[4]:
        print(stmt)
    print("goto L3")
    print("L2:")
    for stmt in p[8]:
        print(stmt)
    print("L3:")


def p_statement_while(p):
    'statement : WHILE expression LBRACE statements RBRACE'
    p[0] = ('WHILE', p[2], p[4])
    print("L1:")
    print(f"if not {p[2]} goto L2")
    for stmt in p[4]:
        print(stmt)
    print("goto L1")
```

```
    print("L2:")



def p_statement_print(p):
    '''statement : PRINT expression EOL
    | PRINT STRING EOL
    | PRINT SINGLEQUOTE STRING SINGLEQUOTE EOL
    | PRINT DOUBLEQUOTE STRING DOUBLEQUOTE EOL'''
    if len(p) == 4:
        p[0] = ('PRINT', p[2], p[3])
        print(f"print {p[2]}")
    else:
        p[0] = ('PRINT', p[2], None)
        p[0] = ('PRINT', p[2], p[3])


        print(f"print {p[2]}")


def p_expression_string(p):
    '''expression : STRING'''
    p[0] = p[1]


def p_statements(p):
    '''statements : statement
                  | statements statement'''
    if len(p) == 2:
        p[0] = [p[1]]
        for stmt in p[1]:
```

```
            print(stmt)
    else:
        p[1].append(p[2])
        p[0] = p[1]
        for stmt in p[1]:
            print(stmt)
        for stmt in p[2]:
            print(stmt)

def p_program(p):
    'statement : statement statement'
    p[0] = (p[1], p[2])
    for stmt in p[1]:
        print(stmt)

def p_error(p):
    print(f"SyntaxError: Incorrect syntax at line {p.lineno}")

def p_empty(p):
    'empty :'
    pass

# Build intermediate code
parser = yacc.yacc()

# Test intermediate code with sample input
data = '''x = 42;
```

```
y = 3.14;

z = a;

if (x < y) {

    print("x is less than y");

} else {

    print('x is greater than or equal to y');

}

while (x < 50) {

    x = x + 1;

}

print(x);

'''









# Create the input

result = parser.parse(data)


# Print the result

print(result)
```

## SCREENSHOT SHOWING OUTPUT RUN-TIME