Entrega #0

1. Expresiones Regulares
    - Identificadores (ID):
      ```
      r'[a-zA-Z_][a-zA-Z_0-9]*'
      ```
      Se usa para nombrar variables, funciones, etc.

    - Constantes Enteras (CTE_INT):
      ```
      r'\d+'
      ```
      Representa números enteros positivos o negativos.

    - Constantes Punto Flotante (CTE_FLOAT):
      ```
      r'\d+\.\d+'
      ```
      Representa números con parte fraccionaria, positivos o negativos.

    - Constantes de String (CTE_STRING):
      ```
      r'("[^\n"]*")|(\'[^\n\']*\')'
      ```
      Representa secuencias de caracteres entre comillas, encapsulando cualquier secuencia válida, incluyendo caracteres de escape.

2. Tokens
    - Keywords:
      ```
      'program': 'PROGRAM',
      'main': 'MAIN',
      'end': 'END',
      'var': 'VAR',
      'int': 'INT',
      'float': 'FLOAT',
      'print': 'PRINT',
      'if': 'IF',
      'else': 'ELSE',
      'void': 'VOID',
      'while': 'WHILE',
      'do': 'DO'
      ```

    - Operadores:
        - Aritméticos:
          ```
          t_PLUS = r'\+'
          t_MINUS = r'-'
          t_TIMES = r'\*'
          t_DIVIDE = r'/'
          ```

        - Relacionales:

```
t_LT = r'<'
t_GT = r'>'
t_EQUALS = r'='
t_NE = r'!='
```

- Delimitadores:

```
t_COLON = r':'
t_SEMICOLON = r';'
t_COMMA = r','
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_LPAREN = r'\('
t_RPAREN = r'\)'
```

3. Gramática

```
def p_program(p):
    'program : PROGRAM ID SEMICOLON a_vars a_funcs MAIN body
END'
    pass

def p_a_vars(p):
    '''a_vars : empty
              | vars'''
    # Manejar la declaración de variables
    lista_variables = p[1]
    for var in lista_variables:
        nombre, tipo = var.split(':')
        variable_table.add_variable(nombre, tipo)

def p_a_funcs(p):
    '''a_funcs : empty
               | funcs b_funcs'''
    # Manejar la declaración de funciones
    pass

def p_b_funcs(p):
    '''b_funcs : funcs b_funcs
               | funcs'''
    # Manejar múltiples funciones
    pass
```

```
# Módulo de VARS
def p_vars(p):
    '''vars : VAR ID COLON type SEMICOLON list_vars'''
    # Manejar la declaración de variables
    list_vars = p[6]
    p[0] = (f'{p[2]}:{p[4]}', *list_vars.split(','))

def p_list_vars(p):
    '''list_vars : empty
                 | ID COLON type SEMICOLON list_vars'''
    # Manejar la lista de variables
    if len(p) > 2:
        if p[5] != None:
            p[0] = f'{p[1]}:{p[3]},{p[5]}'
        else:
            p[0] = f'{p[1]}:{p[3]}'

def p_type(p):
    '''type : INT
            | FLOAT'''
    # Manejar los tipos de datos
    p[0] = p[1]

def p_body(p):
    'body : LBRACE list_statements RBRACE'
    # Manejar el cuerpo de funciones o del programa principal
    pass

def p_list_statements(p):
    '''list_statements : statement body_rep
                       | empty
                       | statement'''
    # Manejar la lista de declaraciones
    pass

def p_statement(p):
    '''statement : assign
                 | condition
                 | cycle
                 | f_call
                 | print_stmt'''
    # Manejar una declaración (asignación, condición, ciclo,
llamada a función, impresión)
```

```
    pass

def p_body_rep(p):
    '''body_rep : statement body_rep
                | statement'''
    # Manejar declaraciones adicionales en el cuerpo
    pass

def p_print_stmt(p):
    'print_stmt : PRINT LPAREN list_expresion RPAREN
SEMICOLON'
    # Manejar la declaración de impresión
    pass

def p_list_expresion(p):
    '''list_expresion : expresion addPrint
                      | expresion addPrint COMMA list_expresion
                      | CTE_STRING addPrintString
                      | CTE_STRING addPrintString COMMA
list_expresion'''
    # Manejar la lista de expresiones en una impresión
    pass

def p_addPrint(p):
    '''addPrint : '''
    # Añadir la operación de impresión
    variable_table.add_print()

def p_addPrintString(p):
    '''addPrintString : '''
    # Añadir la operación de impresión de una cadena
    variable_table.add_print(string=p[-1])

def p_assign(p):
    'assign : ID add_operand EQUALS add_operador expresion
SEMICOLON'
    # Manejar la asignación de valores a variables
    variable_table.add_assing()
    pass

def p_add_operand(p):
    '''add_operand : '''
    # Añadir un operando a la pila
```

```
    variable_table.add_operand(p[-1])

def p_add_operador(p):
    '''add_operador : '''
    # Añadir un operador a la pila
    variable_table.append_pila_operador(p[-1])

def p_cycle(p):
    'cycle : DO ciclo_start body WHILE LPAREN expresion
RPAREN gotov SEMICOLON'
    # Manejar el ciclo do-while
    pass

def p_ciclo_start(p):
    '''ciclo_start : '''
    # Marcar el inicio del ciclo
    variable_table.start_while()

def p_gotov(p):
    '''gotov : '''
    # Añadir la instrucción de salto condicional para el
ciclo
    variable_table.add_gotov_while()

def p_condition(p):
    'condition : IF LPAREN expresion RPAREN gotof body
else_part SEMICOLON'
    # Manejar la declaración if-else
    variable_table.add_gotoFfill()

def p_gotof(p):
    '''gotof : '''
    # Añadir la instrucción de salto condicional para if
    variable_table.add_gotof()

def p_else_part(p):
    '''else_part : ELSE goto body
                 | empty'''
    # Manejar la parte else de una declaración if-else
    pass

def p_goto(p):
    '''goto : '''
```

```
    # Añadir una instrucción de salto incondicional
    variable_table.add_goto()

def p_expresion(p):
    '''expresion : exp comparar_exp exp
                 | exp'''
    # Manejar la evaluación de una expresión
    variable_table.add_expresion()

def p_comparar_exp(p):
    '''comparar_exp : LT
                    | GT
                    | NE'''
    # Añadir un operador de comparación a la pila
    variable_table.append_pila_operador(p[1])

def p_exp(p):
    '''exp : termino add_termino
           | termino add_termino next_termino'''
    # Manejar la evaluación de términos en una expresión
    pass

def p_add_termino(p):
    '''add_termino : '''
    # Añadir un término a la pila
    variable_table.add_termino()

def p_next_termino(p):
    '''next_termino : sum_rest exp '''
    # Manejar la evaluación de términos adicionales
    pass

def p_sum_rest(p):
    '''sum_rest : PLUS
                | MINUS'''
    # Añadir un operador de suma o resta a la pila
    variable_table.append_pila_operador(p[1])
    pass

def p_termino(p):
    '''termino : factor add_factor next_factor
               | factor add_factor'''
    # Manejar la evaluación de un término
```

```
    pass

def p_next_factor(p):
    '''next_factor : mult_div termino'''
    # Manejar la evaluación de factores adicionales
    pass

def p_mult_div(p):
    '''mult_div : TIMES
                | DIVIDE'''
    # Añadir un operador de multiplicación o división a la
pila
    variable_table.append_pila_operador(p[1])
    pass

def p_factor(p):
    '''factor : LPAREN expresion RPAREN
              | id_cte'''
    # Manejar la evaluación de un factor
    pass

def p_add_factor(p):
    '''add_factor : '''
    # Añadir un factor a la pila
    variable_table.add_factor()

def p_id_cte(p):
    '''id_cte : ID push_var
              | cte push_const'''
    # Manejar identificadores y constantes
    pass

def p_push_const(p):
    '''push_const : '''
    # Añadir una constante a la pila
    variable_table.add_operand(p[-1], True)

def p_push_var(p):
    '''push_var : '''
    # Añadir una variable a la pila
    variable_table.add_operand(p[-1])

def p_cte(p):
```

```
    '''cte : CTE_INT
           | CTE_FLOAT'''
    # Manejar constantes enteras y de punto flotante
    p[0] = p[1]

def p_funcs(p):
    'funcs : VOID ID LPAREN list_params RPAREN LBRACE
var_no_var body RBRACE SEMICOLON'
    # Manejar la declaración de funciones
    pass

def p_list_params(p):
    '''list_params : empty
                   | ID COLON type more_params'''
    # Manejar la lista de parámetros de una función
    pass

def p_more_params(p):
    '''more_params : empty
                   | COMMA ID COLON type more_params'''
    # Manejar parámetros adicionales en una función
    pass

def p_var_no_var(p):
    '''var_no_var : empty
                  | vars'''
    # Manejar variables locales en una función
    pass

def p_f_call(p):
    'f_call : ID LPAREN RPAREN SEMICOLON'
    # Manejar la llamada a una función
    pass

def p_empty(p):
    'empty :'
    # Regla para manejar producción vacía
    pass

def p_error(p):
    # Manejo de errores de sintaxis
    if p:
```

```
        print(f"Syntax error at '{p.value}' (line
{p.lineno})")
    else:
        print("Syntax error at EOF")
```

Entrega #1

1. Investigar algunas herramientas de generación automática de compiladores
    - ANTLR: es un generador de parser que se puede usar para leer, ejecutar o traducir texto estructurado o archivos binarios, y es usado bastante académicamente para desarrollar todo tipo de lenguajes, herramientas, frameworks, etc. Es muy popular se podría decir ya que X(Twitter) lo usa en su search bar, Oracle en su SQL Developer IDE y herramientas de migración, el IDE Netbeans lo usa para parsear C++, etc.

    - PLY: Implementa lex y yacc para Python, una función notable es que es implementado completamente en Python usando el parseo LARL(1), que es bastante bueno y eficiente para gramáticas más grandes. Tiene buena comunidad y buena documentación, ya que se podría decir que es la librería más popular de Python en este ámbito.

    - SLY(Sly Lex-Yacc): Es originado del anterior, y según la repo de github “es modernizado un poco”, los programas escritos en PLY no funcionan en SLY, pero básicamente casi cualquier cosa que se puede hacer en PLY se puede hacer en SLY. Lo que llama la atención de SLY son sus funciones distinguidas de informes detallados de errores, recuperación de errores, soporte para producciones vacías, especificadores de precedencia y gramáticas moderadamente ambiguas, utilizando metaprogramación para facilitar la construcción de analizadores sin archivos generados ni pasos adicionales.Hay un problema y es que está semi-retirado el proyecto desde octubre de 2022, además no hay una documentación muy amplia.

2. Seleccionar la que, a tu juicio, conecte mejor con el lenguaje de desarrollo (y que además tenga buena documentación)
   Al revisar los tres no puedo negar lo mucho que me llama la atención SLY me parece ser que sí es cierto el tema de lo moderno que es. Pero el tema de que está semi descontinuado y la documentación es muy escasa da problemas (lo se porque si intente desarrollar el lexer y parser usando SLY). Por lo tanto por su buena documentación e integración con Python me decantare por PLY.

3. Desarrollar el Scanner y el Parser utilizando las reglas gramaticales y expresiones regulares que generaste para la entrega anterior.
   Mi Scanner o Lexer es el siguiente:

```
import ply.lex as lex

# Lista de nombres de tokens
tokens = [
   'ID', 'CTE_STRING', 'CTE_INT', 'CTE_FLOAT', 'PLUS', 'MINUS',
'TIMES', 'DIVIDE', 'EQUALS',
   'LPAREN', 'RPAREN', 'SEMICOLON', 'COLON', 'COMMA', 'LBRACE',
'RBRACE', 'LT', 'GT', 'NE'
]

# Palabras reservadas
reserved = {
   'program': 'PROGRAM',
   'main': 'MAIN',
   'end': 'END',
   'var': 'VAR',
   'int': 'INT',
   'float': 'FLOAT',
   'print': 'PRINT',
   'if': 'IF',
   'else': 'ELSE',
   'void': 'VOID',
   'while': 'WHILE',
   'do': 'DO'
}

tokens = tokens + list(reserved.values())

# Expresiones regulares para tokens simples
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LT = r'<'
t_GT = r'>'
t_EQUALS = r'='
t_NE = r'!='
t_COLON = r':'
t_SEMICOLON = r';'
```

```
t_COMMA = r','
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# Reglas de expresiones regulares con código de acción
def t_CTE_FLOAT(t):
    r'\d+\.\d+'
    t.value = float(t.value)
    return t

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

def t_CTE_STRING(t):
    r'("[^\n"]*")|(\'[^\n\']*\')'
    t.value = t.value[1:-1]  # Eliminar las comillas
    return t

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'ID')  # Verificar palabras
reservadas
    return t

# Una cadena que contiene caracteres ignorados (espacios y
tabulaciones)
t_ignore = ' \t'

# Definir una regla para nuevas líneas para rastrear los números
de línea
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# Regla de manejo de errores
def t_error(t):
    print(f"Caracter ilegal '{t.value[0]}'")
    t.lexer.skip(1)
```

```
# Construir el lexer
lexer = lex.lex()

# Probarlo
if __name__ == "__main__":
    data = '''
    program test;

    var a : float;
        b : float;
        c : float;
        d : float;
        e : float;
        f : float;
        g : float;
        h : float;
        j : float;
        k : float;

    main {
        a = b + c * (d - e / f) * h;
        b = e - f;
        do {
            h = j * k + b;
            if (b < h) {
                b = h + j;
                do {
                    print (a + b * c, d - e);
                    b = b - j;
                } while (b > a + c);
            } else {
                do {
                    a = a + b;
                    print (b - d);
                } while (a - d < c + b);
            };
        } while (a * b - c > d * e / (g + h));
        f = a + b;
    }
    end
    '''
    lexer.input(data)
    while True:
```

```
        tok = lexer.token()
        if not tok:
            break
        print(tok)
```

Mi Parser es el siguiente:

```
import ply.yacc as yacc
from symbol_table import VariableTable
from little_duck_lex import tokens
import pickle

variable_table = VariableTable()

# Definir precedencia y asociatividad de los operadores
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('left', 'LT', 'GT', 'NE')
)

# Reglas gramaticales
def p_program(p):
    'program : PROGRAM ID SEMICOLON a_vars a_funcs MAIN body END'
    pass

def p_a_vars(p):
    '''a_vars : empty
              | vars'''
    # Manejar la declaración de variables
    lista_variables = p[1]
    for var in lista_variables:
        nombre, tipo = var.split(':')
        variable_table.add_variable(nombre, tipo)

def p_a_funcs(p):
    '''a_funcs : empty
               | funcs b_funcs'''
    # Manejar la declaración de funciones
    pass

def p_b_funcs(p):
```

```
    '''b_funcs : funcs b_funcs
                | funcs'''
    # Manejar múltiples funciones
    pass

# Módulo de VARS
def p_vars(p):
    '''vars : VAR ID COLON type SEMICOLON list_vars'''
    # Manejar la declaración de variables
    list_vars = p[6]
    p[0] = (f'{p[2]}:{p[4]}', *list_vars.split(','))

def p_list_vars(p):
    '''list_vars : empty
                 | ID COLON type SEMICOLON list_vars'''
    # Manejar la lista de variables
    if len(p) > 2:
        if p[5] != None:
            p[0] = f'{p[1]}:{p[3]},{p[5]}'
        else:
            p[0] = f'{p[1]}:{p[3]}'

def p_type(p):
    '''type : INT
            | FLOAT'''
    # Manejar los tipos de datos
    p[0] = p[1]

def p_body(p):
    'body : LBRACE list_statements RBRACE'
    # Manejar el cuerpo de funciones o del programa principal
    pass

def p_list_statements(p):
    '''list_statements : statement body_rep
                       | empty
                       | statement'''
    # Manejar la lista de declaraciones
    pass

def p_statement(p):
    '''statement : assign
                 | condition
```

```
                | cycle
                | f_call
                | print_stmt'''
    # Manejar una declaración (asignación, condición, ciclo,
llamada a función, impresión)
    pass

def p_body_rep(p):
    '''body_rep : statement body_rep
                        | statement'''
    # Manejar declaraciones adicionales en el cuerpo
    pass

def p_print_stmt(p):
    'print_stmt : PRINT LPAREN list_expresion RPAREN SEMICOLON'
    # Manejar la declaración de impresión
    pass

def p_list_expresion(p):
    '''list_expresion : expresion addPrint
                    | expresion addPrint COMMA list_expresion
                    | CTE_STRING addPrintString
                    | CTE_STRING addPrintString COMMA
list_expresion'''
    # Manejar la lista de expresiones en una impresión
    pass

def p_addPrint(p):
    '''addPrint : '''
    # Añadir la operación de impresión
    variable_table.add_print()

def p_addPrintString(p):
    '''addPrintString : '''
    # Añadir la operación de impresión de una cadena
    variable_table.add_print(string=p[-1])

def p_assign(p):
    'assign : ID add_operand EQUALS add_operador expresion
SEMICOLON'
    # Manejar la asignación de valores a variables
    variable_table.add_assing()
    pass
```

```
def p_add_operand(p):
    '''add_operand : '''
    # Añadir un operando a la pila
    variable_table.add_operand(p[-1])

def p_add_operador(p):
    '''add_operador : '''
    # Añadir un operador a la pila
    variable_table.append_pila_operador(p[-1])

def p_cycle(p):
    'cycle : DO ciclo_start body WHILE LPAREN expresion RPAREN
gotov SEMICOLON'
    # Manejar el ciclo do-while
    pass

def p_ciclo_start(p):
    '''ciclo_start : '''
    # Marcar el inicio del ciclo
    variable_table.start_while()

def p_gotov(p):
    '''gotov : '''
    # Añadir la instrucción de salto condicional para el ciclo
    variable_table.add_gotov_while()

def p_condition(p):
    'condition : IF LPAREN expresion RPAREN gotof body else_part
SEMICOLON'
    # Manejar la declaración if-else
    variable_table.add_gotoFfill()

def p_gotof(p):
    '''gotof : '''
    # Añadir la instrucción de salto condicional para if
    variable_table.add_gotof()

def p_else_part(p):
    '''else_part : ELSE goto body
                 | empty'''
    # Manejar la parte else de una declaración if-else
    pass
```

```
def p_goto(p):
    '''goto : '''
    # Añadir una instrucción de salto incondicional
    variable_table.add_goto()

def p_expresion(p):
    '''expresion : exp comparar_exp exp
                 | exp'''
    # Manejar la evaluación de una expresión
    variable_table.add_expresion()

def p_comparar_exp(p):
    '''comparar_exp : LT
                    | GT
                    | NE'''
    # Añadir un operador de comparación a la pila
    variable_table.append_pila_operador(p[1])

def p_exp(p):
    '''exp : termino add_termino
           | termino add_termino next_termino'''
    # Manejar la evaluación de términos en una expresión
    pass

def p_add_termino(p):
    '''add_termino : '''
    # Añadir un término a la pila
    variable_table.add_termino()

def p_next_termino(p):
    '''next_termino : sum_rest exp '''
    # Manejar la evaluación de términos adicionales
    pass

def p_sum_rest(p):
    '''sum_rest : PLUS
                | MINUS'''
    # Añadir un operador de suma o resta a la pila
    variable_table.append_pila_operador(p[1])
    pass

def p_termino(p):
```

```
    '''termino : factor add_factor next_factor
               | factor add_factor'''
    # Manejar la evaluación de un término
    pass

def p_next_factor(p):
    '''next_factor : mult_div termino'''
    # Manejar la evaluación de factores adicionales
    pass

def p_mult_div(p):
    '''mult_div : TIMES
                | DIVIDE'''
    # Añadir un operador de multiplicación o división a la pila
    variable_table.append_pila_operador(p[1])
    pass

def p_factor(p):
    '''factor : LPAREN expresion RPAREN
              | id_cte'''
    # Manejar la evaluación de un factor
    pass

def p_add_factor(p):
    '''add_factor : '''
    # Añadir un factor a la pila
    variable_table.add_factor()

def p_id_cte(p):
    '''id_cte : ID push_var
              | cte push_const'''
    # Manejar identificadores y constantes
    pass

def p_push_const(p):
    '''push_const : '''
    # Añadir una constante a la pila
    variable_table.add_operand(p[-1], True)

def p_push_var(p):
    '''push_var : '''
    # Añadir una variable a la pila
    variable_table.add_operand(p[-1])
```

```
def p_cte(p):
    '''cte : CTE_INT
           | CTE_FLOAT'''
    # Manejar constantes enteras y de punto flotante
    p[0] = p[1]

def p_funcs(p):
    'funcs : VOID ID LPAREN list_params RPAREN LBRACE var_no_var
body RBRACE SEMICOLON'
    # Manejar la declaración de funciones
    pass

def p_list_params(p):
    '''list_params : empty
                   | ID COLON type more_params'''
    # Manejar la lista de parámetros de una función
    pass

def p_more_params(p):
    '''more_params : empty
                   | COMMA ID COLON type more_params'''
    # Manejar parámetros adicionales en una función
    pass

def p_var_no_var(p):
    '''var_no_var : empty
                  | vars'''
    # Manejar variables locales en una función
    pass

def p_f_call(p):
    'f_call : ID LPAREN RPAREN SEMICOLON'
    # Manejar la llamada a una función
    pass

def p_empty(p):
    'empty :'
    # Regla para manejar producción vacía
    pass

def p_error(p):
    # Manejo de errores de sintaxis
```

```
    if p:
        print(f"Syntax error at '{p.value}' (line {p.lineno})")
    else:
        print("Syntax error at EOF")

# Construir el parser
parser = yacc.yacc()

# Leer el código desde un archivo de texto
with open('file.txt', 'r') as file:
    data = file.read()

# Probar el parser
parser.parse(data, tracking=True)
print(variable_table.pila_operandos)
print(variable_table.pila_operadores)
print(variable_table.pila_tipos)
print(variable_table.pila_saltos)
print(variable_table.pila_cuadruplos)

# Guardar variable_table.pila_cuadruplos en un archivo pkl
with
open('/home/ricardo/Desktop/Little_Duck/pila_cuadruplos.pkl',
'wb') as file:
    pickle.dump(variable_table.pila_cuadruplos, file)

# Guardar variable_table.constant_table en un archivo pkl
def invert_dict(d):
    return {v: k for k, v in d.items()}

constant_table = invert_dict(variable_table.constant_table)

# Guardar el diccionario en un archivo pickle
with open('constant_table.pkl', 'wb') as pickle_file:
    pickle.dump(constant_table, pickle_file)
```

En ambos casos hice uso debido de la documentación de PLY para su desarrollo, también me ayudó a entender algunas cosas que no tenía tan claras y a darme cuenta de errores que tenía en mi primera entrega y que en esta nueva he modificado.

Entrega #2

1. Diseñar la Tabla de consideraciones semánticas (cubo semántico) para el lenguaje.

```python
class SemanticCube:
    def __init__(self):
        self.types = ['int', 'float', 'bool']
        self.operations = ['+', '-', '*', '/', '&&', '||', '==',
'!=', '>', '<', '>=', '<=', '=']
        self.cube = self._create_cube()

    def _create_cube(self):
        cube = {}
        for op in self.operations:
            cube[op] = {}
            for type1 in self.types:
                cube[op][type1] = {}
                for type2 in self.types:
                    # Definir las reglas del cubo semántico aquí
                    if op in ['+', '-', '*', '/']:
                        if type1 == 'int' and type2 == 'int':
                            cube[op][type1][type2] = 'int'
                        elif type1 == 'float' or type2 == 'float':
                            cube[op][type1][type2] = 'float'
                        else:
                            cube[op][type1][type2] = 'error'
                    elif op in ['&&', '||']:
                        if type1 == 'bool' and type2 == 'bool':
                            cube[op][type1][type2] = 'bool'
                        else:
                            cube[op][type1][type2] = 'error'
                    elif op in ['==', '!=', '>', '<', '>=', '<=']:
                        if type1 == type2:
                            cube[op][type1][type2] = 'bool'
                        else:
                            cube[op][type1][type2] = 'error'

                    elif op == '=':
                        if type1 == type2:
                            cube[op][type1][type2] = type1
                        elif type1 == 'float' and type2 == 'int':
                            cube[op][type1][type2] = type1
                        elif type1 == 'float' and type2 == 'bool':
                            cube[op][type1][type2] = type2
```

```
                    elif type1 == 'int' and type2 == 'bool':
                        cube[op][type1][type2] = type2
                    else:
                        cube[op][type1][type2] = 'error'
        return cube

    def get_result_type(self, op, type1, type2):
        return self.cube.get(op, {}).get(type1, {}).get(type2,
'error')
```

2. Implementar las estructuras de datos que representan al Directorio de Funciones y a las Tablas de Variables de LittleDuck.

```
from semantic_cube import SemanticCube

class VariableTable:

    def __init__(self):
        self.table = {}
        self.semantic_cube = SemanticCube()
        self.var_int = 0
        self.var_float = 100

        self.var_temp_int = 200
        self.var_temp_float = 300
        self.var_temp_bool = 400

        self.var_const_int = 500
        self.var_const_float = 600
        self.var_const_string = 700

        self.pila_operadores = []
        self.pila_operandos = []
        self.pila_tipos = []
        self.pila_saltos = []
        self.pila_cuadruplos = []
        self.constant_table = {}

    def append_pila_operador(self, operator):
        # Añadir un operador a la pila de operadores
        self.pila_operadores.append(operator)
```

```
    def add_variable(self, name, var_type):
        # Añadir una variable a la tabla de variables
        if name in self.table:
            raise ValueError(f"Variable '{name}' already exists in
the table.")
        self.table[name] = {'type': var_type, 'memory_address': 0}
        self.add_memory_address(name)

    def get_variable(self, name):
        # Obtener una variable de la tabla de variables
        return self.table.get(name, None)

    def add_memory_address(self, name):
        # Añadir una dirección de memoria a la variable
        if self.table[name]['type'] == 'int':
            memory_address = self.var_int
            self.var_int += 1
        elif self.table[name]['type'] == 'float':
            memory_address = self.var_float
            self.var_float += 1
        self.table[name]['memory_address'] = memory_address

    def print_table(self):
        # Imprimir la tabla de variables
        for var_name, details in self.table.items():
            print(f"Variable: {var_name}, Type:
{details['type']}")

    def add_operand(self, operand, is_cte=False):
        # Añadir un operando a la pila de operandos
        try:
            if is_cte:
                # Añadir la dirección de memoria de la constante
                tipo = type(operand).__name__
                if tipo == 'int':
                    if operand not in self.constant_table:
                        self.var_const_int += 1
                        self.constant_table[operand] =
self.var_const_int


self.pila_operandos.append(self.constant_table[operand])
                    self.pila_tipos.append(tipo)
```

```
                elif tipo == 'float':
                    if operand not in self.constant_table:
                        self.var_const_float += 1
                        self.constant_table[operand] =
self.var_const_float

self.pila_operandos.append(self.constant_table[operand])
                    self.pila_tipos.append(tipo)
            else:
                # Añadir la dirección de memoria de la variable
                tipo = self.get_variable(operand)["type"]
                dir_memoria =
self.get_variable(operand)['memory_address']
                self.pila_operandos.append(dir_memoria)
                self.pila_tipos.append(tipo)
        except:
            raise KeyError(f"Variable '{operand}' was not
declared.")

    def add_factor(self):
        # Añadir un factor (multiplicación o división)
        if self.pila_operadores:
            op = self.pila_operadores[-1]
            if op == '*' or op == '/':
                right_operand = self.pila_operandos.pop()
                left_operand = self.pila_operandos.pop()
                right_operand_tipo = self.pila_tipos.pop()
                left_operand_tipo = self.pila_tipos.pop()
                operator = self.pila_operadores.pop()

                res_tipo =
self.semantic_cube.get_result_type(operator, left_operand_tipo,
right_operand_tipo)

                if res_tipo == 'error':
                    raise ValueError(f"Invalid operation:
{left_operand_tipo} {operator} {right_operand_tipo}")

                elif res_tipo == 'int':
                    res_address = self.var_temp_int
                    self.pila_operandos.append(res_address)
                    self.var_temp_int += 1
```

```
                    self.pila_tipos.append(res_tipo)
                    self.pila_cuadruplos.append([operator,
left_operand, right_operand, res_address])

                elif res_tipo == 'float':
                    res_address = self.var_temp_float
                    self.pila_operandos.append(res_address)
                    self.var_temp_float += 1
                    self.pila_tipos.append(res_tipo)
                    self.pila_cuadruplos.append([operator,
left_operand, right_operand, res_address])

    def add_termino(self):
        # Añadir un término (suma o resta)
        if self.pila_operadores:
            op = self.pila_operadores[-1]
            if op == '+' or op == '-':
                right_operand = self.pila_operandos.pop()
                left_operand = self.pila_operandos.pop()
                right_operand_tipo = self.pila_tipos.pop()
                left_operand_tipo = self.pila_tipos.pop()
                operator = self.pila_operadores.pop()

                res_tipo =
self.semantic_cube.get_result_type(operator, left_operand_tipo,
right_operand_tipo)

                if res_tipo == 'error':
                    raise ValueError(f"Invalid operation:
{left_operand_tipo} {operator} {right_operand_tipo}")

                elif res_tipo == 'int':
                    res_address = self.var_temp_int
                    self.pila_operandos.append(res_address)
                    self.var_temp_int += 1
                    self.pila_tipos.append(res_tipo)
                    self.pila_cuadruplos.append([operator,
left_operand, right_operand, res_address])

                elif res_tipo == 'float':
                    res_address = self.var_temp_float
                    self.pila_operandos.append(res_address)
                    self.var_temp_float += 1
```

```
                    self.pila_tipos.append(res_tipo)
                    self.pila_cuadruplos.append([operator,
left_operand, right_operand, res_address])

    def add_expresion(self):
        # Añadir una expresión (comparación)
        if self.pila_operadores:
            op = self.pila_operadores[-1]
            if op == '>' or op == '<' or op == '!=':
                right_operand = self.pila_operandos.pop()
                left_operand = self.pila_operandos.pop()
                right_operand_tipo = self.pila_tipos.pop()
                left_operand_tipo = self.pila_tipos.pop()
                operator = self.pila_operadores.pop()

                res_tipo =
self.semantic_cube.get_result_type(operator, left_operand_tipo,
right_operand_tipo)

                if res_tipo == 'bool':
                    self.var_temp_bool += 1
                    res_address = self.var_temp_bool
                    self.pila_operandos.append(res_address)
                    self.pila_tipos.append(res_tipo)
                    self.pila_cuadruplos.append([operator,
left_operand, right_operand, res_address])
                else:
                    raise ValueError(f"Invalid operation:
{left_operand_tipo} {operator} {right_operand_tipo}")

    def add_assing(self):
        # Añadir una asignación
        if self.pila_operadores:
            op = self.pila_operadores[-1]
            if op == '=':
                right_operand = self.pila_operandos.pop()
                left_operand = self.pila_operandos.pop()
                right_operand_tipo = self.pila_tipos.pop()
                left_operand_tipo = self.pila_tipos.pop()
                operator = self.pila_operadores.pop()
```

```
                res_tipo =
self.semantic_cube.get_result_type(operator, left_operand_tipo,
right_operand_tipo)

                if res_tipo != 'error':
                    self.pila_cuadruplos.append([operator,
right_operand, None, left_operand])
                else:
                    raise ValueError(f"Invalid operation:
{left_operand_tipo} {operator} {right_operand_tipo}")

    def add_print(self, string=[]):
        # Añadir una operación de impresión
        if string == []:
            right_operand = self.pila_operandos.pop()
            right_operand_tipo = self.pila_tipos.pop()
            self.pila_cuadruplos.append(['print', right_operand,
None, None])
        else:
            self.pila_cuadruplos.append(['print', string, None,
None])

    def add_gotof(self):
        # Añadir una instrucción GOTOF
        condition = self.pila_operandos.pop()
        condition_type = self.pila_tipos.pop()

        if condition_type != 'bool':
            raise ValueError("Condition for if statement must be a
boolean")

        self.pila_cuadruplos.append(['GOTOF', condition, None,
None])
        self.pila_saltos.append(len(self.pila_cuadruplos) - 1)

    def add_goto(self):
        # Añadir una instrucción GOTO
        self.pila_cuadruplos.append(['GOTO', None, None, None])
        false_jump = self.pila_saltos.pop()

        self.pila_saltos.append(len(self.pila_cuadruplos) - 1)
        self.pila_cuadruplos[false_jump][-1] =
len(self.pila_cuadruplos)
```

```
    def add_gotoFfill(self):
        # Completar una instrucción GOTO
        false_jump = self.pila_saltos.pop()
        self.pila_cuadruplos[false_jump][-1] =
len(self.pila_cuadruplos)

    def start_while(self):
        # Marcar el inicio del ciclo while
        self.pila_saltos.append(len(self.pila_cuadruplos))

    def add_gotov_while(self):
        # Añadir una instrucción GOTOV para el ciclo while
        condition = self.pila_operandos.pop()
        condition_type = self.pila_tipos.pop()

        if condition_type != 'bool':
            raise ValueError("Condition for while statement must
be a boolean")

        direccion_salto = self.pila_saltos.pop()
        self.pila_cuadruplos.append(['GOTOV', condition, None,
direccion_salto])
```

3. Establecer los puntos neurálgicos que permitan crear y llenar, tanto el Directorio de Funciones como las Tablas de variables del programa con las validaciones pertinentes (como Variable doblemente declarada).

Puntos Neurálgicos en el Parser para Crear y Llenar el Directorio de Funciones y las Tablas de Variables

```
    variable_table = VariableTable()
```

La clase VariableTable se inicializa al principio del archivo y se utiliza para almacenar todas las variables del programa.

```
def p_vars(p):
    '''vars : VAR ID COLON type SEMICOLON list_vars'''
    # Manejar la declaración de variables
    list_vars = p[6]
    p[0] = (f'{p[2]}:{p[4]}', *list_vars.split(','))

def p_list_vars(p):
```

```
    '''list_vars : empty
                 | ID COLON type SEMICOLON list_vars'''
    # Manejar la lista de variables
    if len(p) > 2:
        if p[5] != None:
            p[0] = f'{p[1]}:{p[3]},{p[5]}'
        else:
            p[0] = f'{p[1]}:{p[3]}'

def p_type(p):
    '''type : INT
            | FLOAT'''
    # Manejar los tipos de datos
    p[0] = p[1]
```

En la regla p_vars, se maneja la declaración de variables, extrayendo el nombre y tipo de cada variable y añadiéndolas a la tabla de variables.

```
def p_a_vars(p):
    '''a_vars : empty
              | vars'''
    # Manejar la declaración de variables
    lista_variables = p[1]
    for var in lista_variables:
        nombre, tipo = var.split(':')
        variable_table.add_variable(nombre, tipo)
```

En la regla p_a_vars, se añade cada variable declarada a la tabla de variables y se verifica si ya ha sido declarada anteriormente.

```
def p_assign(p):
    'assign : ID add_operand EQUALS add_operador expresion
SEMICOLON'
    # Manejar la asignación de valores a variables
    variable_table.add_assing()
    pass
```

En la regla p_assign, se maneja la asignación de valores a variables, asegurándose de que la variable exista y añadiendo la operación correspondiente a la lista de cuádruplos.

```
def p_condition(p):
    'condition : IF LPAREN expresion RPAREN gotof body else_part
SEMICOLON'
    # Manejar la declaración if-else
    variable_table.add_gotoFfill()

def p_cycle(p):
    'cycle : DO ciclo_start body WHILE LPAREN expresion RPAREN
gotov SEMICOLON'
    # Manejar el ciclo do-while
    pass
```

Las reglas p_condition y p_cycle manejan las estructuras de control del lenguaje (if-else y do-while), añadiendo las operaciones de salto condicional y no condicional a la lista de cuádruplos.

```
def p_exp(p):
    '''exp : termino add_termino
           | termino add_termino next_termino'''
    # Manejar la evaluación de términos en una expresión
    pass

def p_termino(p):
    '''termino : factor add_factor next_factor
               | factor add_factor'''
    # Manejar la evaluación de un término
    pass

def p_factor(p):
    '''factor : LPAREN expresion RPAREN
              | id_cte'''
    # Manejar la evaluación de un factor
    pass
```

Las reglas p_exp, p_termino, y p_factor manejan las expresiones aritméticas y lógicas, añadiendo los operadores y operandos a las pilas correspondientes para generar los cuádruplos.

Validaciones Pertinentes

Variable Doblamente Declarada:

```
 def add_variable(self, name, var_type):
        # Añadir una variable a la tabla de variables
        if name in self.table:
            raise ValueError(f"Variable '{name}' already exists in
the table.")
        self.table[name] = {'type': var_type, 'memory_address': 0}
        self.add_memory_address(name)
```

Tipos de Datos Consistentes:

```
def p_type(p):
    '''type : INT
            | FLOAT'''
    # Manejar los tipos de datos
    p[0] = p[1]

def p_list_vars(p):
    '''list_vars : empty
                 | ID COLON type SEMICOLON list_vars'''
    # Manejar la lista de variables
    if len(p) > 2:
        if p[5] != None:
            p[0] = f'{p[1]}:{p[3]},{p[5]}'
        else:
            p[0] = f'{p[1]}:{p[3]}'
```

Entrega #3

1. Implementar las PILAS necesarias (para Operadores, Operandos, Tipos, Saltos), así como una FILA (para los cuádruplos).

Inicialización de las Pilas y la Fila de Cuádruplos

```
class VariableTable:

    def __init__(self):
        self.table = {}
        self.semantic_cube = SemanticCube()
        self.var_int = 0
        self.var_float = 100

        self.var_temp_int = 200
```

```
        self.var_temp_float = 300
        self.var_temp_bool = 400

        self.var_const_int = 500
        self.var_const_float = 600
        self.var_const_string = 700

        self.pila_operadores = []
        self.pila_operandos = []
        self.pila_tipos = []
        self.pila_saltos = []
        self.pila_cuadruplos = []
        self.constant_table = {}
```

Estas estructuras están definidas dentro de la clase VariableTable y se inicializan en el constructor “__init__”

Operadores

```
    def append_pila_operador(self, operator):
        # Añadir un operador a la pila de operadores
        self.pila_operadores.append(operator)
```

La pila de operadores se gestiona principalmente con el método append_pila_operador que añade un operador a la pila.

Operandos

```
    def add_operand(self, operand, is_cte=False):
        # Añadir un operando a la pila de operandos
        try:
            if is_cte:
                # Añadir la dirección de memoria de la constante
                tipo = type(operand).__name__
                if tipo == 'int':
                    if operand not in self.constant_table:
                        self.var_const_int += 1
                        self.constant_table[operand] =
self.var_const_int


self.pila_operandos.append(self.constant_table[operand])
                    self.pila_tipos.append(tipo)

                elif tipo == 'float':
```

```
                    if operand not in self.constant_table:
                        self.var_const_float += 1
                        self.constant_table[operand] =
self.var_const_float

self.pila_operandos.append(self.constant_table[operand])
                    self.pila_tipos.append(tipo)
            else:
                # Añadir la dirección de memoria de la variable
                tipo = self.get_variable(operand)["type"]
                dir_memoria =
self.get_variable(operand)['memory_address']
                self.pila_operandos.append(dir_memoria)
                self.pila_tipos.append(tipo)
        except:
            raise KeyError(f"Variable '{operand}' was not
declared.")
```

La pila de operandos se gestiona con el método add_operand que añade un operando a la pila, aquí también se manejan los tipos de datos y se añaden a la pila de tipos.

Saltos

```
    def add_gotof(self):
        # Añadir una instrucción GOTOF
        condition = self.pila_operandos.pop()
        condition_type = self.pila_tipos.pop()

        if condition_type != 'bool':
            raise ValueError("Condition for if statement must be a
boolean")

        self.pila_cuadruplos.append(['GOTOF', condition, None, None])
        self.pila_saltos.append(len(self.pila_cuadruplos) - 1)

    def add_goto(self):
        # Añadir una instrucción GOTO
        self.pila_cuadruplos.append(['GOTO', None, None, None])
        false_jump = self.pila_saltos.pop()

        self.pila_saltos.append(len(self.pila_cuadruplos) - 1)
        self.pila_cuadruplos[false_jump][-1] = len(self.pila_cuadruplos)

    def add_gotoFfill(self):
```

```
        # Completar una instrucción GOTO
        false_jump = self.pila_saltos.pop()
        self.pila_cuadruplos[false_jump][-1] = len(self.pila_cuadruplos)

    def start_while(self):
        # Marcar el inicio del ciclo while
        self.pila_saltos.append(len(self.pila_cuadruplos))

    def add_gotov_while(self):
        # Añadir una instrucción GOTOV para el ciclo while
        condition = self.pila_operandos.pop()
        condition_type = self.pila_tipos.pop()

        if condition_type != 'bool':
            raise ValueError("Condition for while statement must be a
boolean")

        direccion_salto = self.pila_saltos.pop()
        self.pila_cuadruplos.append(['GOTOV', condition, None,
direccion_salto])
```

La pila de saltos se gestiona con los métodos add_gotof, add_goto, add_gotoFfill, add_gotov_while y start_while

Cuádruplos

```
def add_factor(self):
        # Añadir un factor (multiplicación o división)
        if self.pila_operadores:
            op = self.pila_operadores[-1]
            if op == '*' or op == '/':
                right_operand = self.pila_operandos.pop()
                left_operand = self.pila_operandos.pop()
                right_operand_tipo = self.pila_tipos.pop()
                left_operand_tipo = self.pila_tipos.pop()
                operator = self.pila_operadores.pop()

                res_tipo =
self.semantic_cube.get_result_type(operator, left_operand_tipo,
right_operand_tipo)

                if res_tipo == 'error':
```

```
                    raise ValueError(f"Invalid operation:
{left_operand_tipo} {operator} {right_operand_tipo}")

                elif res_tipo == 'int':
                    res_address = self.var_temp_int
                    self.pila_operandos.append(res_address)
                    self.var_temp_int += 1
                    self.pila_tipos.append(res_tipo)
                    self.pila_cuadruplos.append([operator,
left_operand, right_operand, res_address])

                elif res_tipo == 'float':
                    res_address = self.var_temp_float
                    self.pila_operandos.append(res_address)
                    self.var_temp_float += 1
                    self.pila_tipos.append(res_tipo)
                    self.pila_cuadruplos.append([operator,
left_operand, right_operand, res_address])
```

Los cuádruplos se generan y almacenan en la pila_cuadruplos mediante varios métodos en la clase VariableTable.

2. Implementar los algoritmos de traducción a cuádruplos para: Expresiones aritméticas y relacionales, así como los Estatutos lineales y NO-Lineales del lenguaje.

   Expresiones Aritméticas

```
def p_exp(p):
    '''exp : termino add_termino
           | termino add_termino next_termino'''
    # Manejar la evaluación de términos en una expresión
    pass

def p_add_termino(p):
    '''add_termino : '''
    # Añadir un término a la pila
    variable_table.add_termino()

def p_next_termino(p):
    '''next_termino : sum_rest exp '''
    # Manejar la evaluación de términos adicionales
    pass
```

```
def p_sum_rest(p):
    '''sum_rest : PLUS
                | MINUS'''
    # Añadir un operador de suma o resta a la pila
    variable_table.append_pila_operador(p[1])
    pass

def p_termino(p):
    '''termino : factor add_factor next_factor
               | factor add_factor'''
    # Manejar la evaluación de un término
    pass

def p_next_factor(p):
    '''next_factor : mult_div termino'''
    # Manejar la evaluación de factores adicionales
    pass

def p_mult_div(p):
    '''mult_div : TIMES
                | DIVIDE'''
    # Añadir un operador de multiplicación o división a la pila
    variable_table.append_pila_operador(p[1])
    pass

def p_factor(p):
    '''factor : LPAREN expresion RPAREN
              | id_cte'''
    # Manejar la evaluación de un factor
    pass

def p_add_factor(p):
    '''add_factor : '''
    # Añadir un factor a la pila
    variable_table.add_factor()
```

Las reglas p_exp, p_termino, y p_factor manejan las expresiones aritméticas, añadiendo operadores y operandos a las pilas correspondientes y generando los cuádruplos necesarios.

Expresiones Relacionales

```
def p_expresion(p):
    '''expresion : exp comparar_exp exp
                 | exp'''
    # Manejar la evaluación de una expresión
    variable_table.add_expresion()

def p_comparar_exp(p):
    '''comparar_exp : LT
                    | GT
                    | NE'''
    # Añadir un operador de comparación a la pila
    variable_table.append_pila_operador(p[1])
```

Las reglas p_expresion y p_comparar_exp manejan las expresiones relacionales, añadiendo operadores de comparación y generando los cuádruplos correspondientes.

Estatutos Lineales (Asignaciones)

```
def p_assign(p):
    'assign : ID add_operand EQUALS add_operador expresion
SEMICOLON'
    # Manejar la asignación de valores a variables
    variable_table.add_assing()
    pass
```

La regla p_assign maneja la asignación de valores a variables, generando el cuádruplo correspondiente.

Estatutos No-Lineales (Condiciones y Ciclos)

```
def p_condition(p):
    'condition : IF LPAREN expresion RPAREN gotof body else_part
SEMICOLON'
    # Manejar la declaración if-else
    variable_table.add_gotoFfill()

def p_cycle(p):
    'cycle : DO ciclo_start body WHILE LPAREN expresion RPAREN
gotov SEMICOLON'
    # Manejar el ciclo do-while
    pass
```

```
def p_gotof(p):
    '''gotof : '''
    # Añadir la instrucción de salto condicional para if
    variable_table.add_gotof()

def p_gotov(p):
    '''gotov : '''
    # Añadir la instrucción de salto condicional para el ciclo
    variable_table.add_gotov_while()
```

3. Desplegar el contenido final de la Fila de cuádruplos que se generan para diversos programas de prueba.

```
# Probar el parser
parser.parse(data, tracking=True)
print(variable_table.pila_operandos)
print(variable_table.pila_operadores)
print(variable_table.pila_tipos)
print(variable_table.pila_saltos)
print(variable_table.pila_cuadruplos)

# Guardar variable_table.pila_cuadruplos en un archivo pkl
with open('/home/ricardo/Desktop/Little_Duck/pila_cuadruplos.pkl',
'wb') as file:
    pickle.dump(variable_table.pila_cuadruplos, file)

# Guardar variable_table.constant_table en un archivo pkl
def invert_dict(d):
    return {v: k for k, v in d.items()}

constant_table = invert_dict(variable_table.constant_table)

# Guardar el diccionario en un archivo pickle
with open('constant_table.pkl', 'wb') as pickle_file:
    pickle.dump(constant_table, pickle_file)
```

Entrega#4

1. Desarrollar una pequeña Máquina Virtual que interprete los diferentes códigos de operación que generaste como representación intermedia y que produzca resultados.

```
import pickle
```

```
class VirtualMachine:
    def __init__(self, const_table):
        self.const_table = const_table
        self.memory = {}
        self.instruction_pointer = 0
        self.var_int = 0
        self.var_float = 100

        self.var_temp_int = 200
        self.var_temp_float = 300
        self.var_temp_bool = 400

        self.var_const_int = 500
        self.var_const_float = 600
        self.var_const_string = 700

    def initialize_memory(self):
        # Inicializar memoria para variables globales enteras
        for address in range(self.var_int, self.var_float):
            self.memory[address] = 0

        # Inicializar memoria para variables globales de punto flotante
        for address in range(self.var_float, self.var_temp_int):
            self.memory[address] = 0.0

        # Inicializar memoria para variables temporales enteras
        for address in range(self.var_temp_int, self.var_temp_float):
            self.memory[address] = 0

        # Inicializar memoria para variables temporales de punto
flotante
        for address in range(self.var_temp_float, self.var_temp_bool):
            self.memory[address] = 0.0

        # Inicializar memoria para variables temporales booleanas
        for address in range(self.var_temp_bool, self.var_const_int):
            self.memory[address] = False

        # Inicializar memoria para constantes enteras
        for address in range(self.var_const_int, self.var_const_float):
            self.memory[address] = 0
```

```
        # Inicializar memoria para constantes de punto flotante
        for address in range(self.var_const_float,
self.var_const_string):
            self.memory[address] = 0.0

        # Inicializar memoria para constantes de cadena
        for address in range(self.var_const_string,
self.var_const_string + 50):
            self.memory[address] = ""

    def load_quadruples(self, quadruples):
        # Cargar cuádruplos en la máquina virtual
        self.quadruples = quadruples

    def execute(self):
        # Ejecutar los cuádruplos
        while self.instruction_pointer < len(self.quadruples):
            quad = self.quadruples[self.instruction_pointer]
            op = quad[0]
            left = quad[1]
            right = quad[2]
            result = quad[3]

            if op == '+':
                self.memory[result] = self.memory[left] +
self.memory[right]
            elif op == '-':
                self.memory[result] = self.memory[left] -
self.memory[right]
            elif op == '*':
                self.memory[result] = self.memory[left] *
self.memory[right]
            elif op == '/':
                self.memory[result] = self.memory[left] /
self.memory[right]
            elif op == '=':
                self.memory[result] = self.memory[left]
            elif op == '!=':
                self.memory[result] = self.memory[left] !=
self.memory[right]
            elif op == '>':
                self.memory[result] = self.memory[left] >
self.memory[right]
```

```
            elif op == '<':
                self.memory[result] = self.memory[left] <
self.memory[right]
            elif op == 'GOTO':
                self.instruction_pointer = result
                continue
            elif op == 'GOTOV':
                if self.memory[left]:
                    self.instruction_pointer = result
                    continue
            elif op == 'GOTOF':
                if not self.memory[left]:
                    self.instruction_pointer = result
                    continue
            elif op == 'print':
                if type(left) == str:
                    print(left)
                else:
                    print(self.memory[left])
            else:
                raise Exception(f"Unknown operator: {op}")

            self.instruction_pointer += 1

    def set_memory(self, address, value):
        # Establecer un valor en una dirección de memoria específica
        self.memory[address] = value

    def get_memory(self, address):
        # Obtener el valor de una dirección de memoria específica
        return self.memory.get(address, None)

# Cargar el diccionario desde el archivo pickle
with open('constant_table.pkl', 'rb') as pickle_file:
    constant_table = pickle.load(pickle_file)

# Cargar el diccionario desde el archivo pickle
with open('pila_cuadruplos.pkl', 'rb') as pickle_file:
    cuadruplos = pickle.load(pickle_file)

# Crear la máquina virtual e inicializar la memoria
vm = VirtualMachine(constant_table)
```

```
vm.initialize_memory()

# Configurar la memoria con las constantes cargadas
for address, value in constant_table.items():
    vm.set_memory(address, value)

# Cargar y ejecutar los cuádruplos
vm.load_quadruples(cuadruplos)

vm.execute()
```

2. Generar un conjunto de pruebas que sirvan para validar el funcionamiento (en términos de Léxico, Sintaxis, Semántica) así como ejecución de tu proyecto.

   Prueba Lexer

```
import sys
import os
import pytest

# Añadir el directorio raíz del proyecto al path de Python
sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from little_duck_lex import lexer, tokens

def test_reserved_words():
    data = 'program main end var int float print if else void while do'
    lexer.input(data)
    expected_tokens = ['PROGRAM', 'MAIN', 'END', 'VAR', 'INT', 'FLOAT',
'PRINT', 'IF', 'ELSE', 'VOID', 'WHILE', 'DO']

    for expected_token in expected_tokens:
        tok = lexer.token()
        assert tok.type == expected_token

def test_operators():
    data = '+ - * / = != < >'
    lexer.input(data)
    expected_tokens = ['PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'EQUALS',
'NE', 'LT', 'GT']

    for expected_token in expected_tokens:
```

```
        tok = lexer.token()
        assert tok.type == expected_token

def test_delimiters():
    data = '( ) ; : , { }'
    lexer.input(data)
    expected_tokens = ['LPAREN', 'RPAREN', 'SEMICOLON', 'COLON',
'COMMA', 'LBRACE', 'RBRACE']

    for expected_token in expected_tokens:
        tok = lexer.token()
        assert tok.type == expected_token

def test_identifiers():
    data = 'var1 var_2 _var3'
    lexer.input(data)
    expected_tokens = ['ID', 'ID', 'ID']

    for expected_token in expected_tokens:
        tok = lexer.token()
        assert tok.type == expected_token
        assert tok.value in ['var1', 'var_2', '_var3']

def test_constants():
    data = '123 45.67 "hello world" \'test string\''
    lexer.input(data)
    expected_tokens = [('CTE_INT', 123), ('CTE_FLOAT', 45.67),
('CTE_STRING', 'hello world'), ('CTE_STRING', 'test string')]

    for expected_token in expected_tokens:
        tok = lexer.token()
        assert tok.type == expected_token[0]
        assert tok.value == expected_token[1]

def test_illegal_character():
    data = '@'
    lexer.input(data)
    tok = lexer.token()
    assert tok is None  # Illegal character should be skipped
```

Prueba Parser

```
import sys
```

```
import os
import pytest
from io import StringIO
from contextlib import redirect_stdout

# Añadir el directorio raíz del proyecto al path de Python
sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from little_duck_pars import parser, variable_table

def parse_input(input_data):
   variable_table.__init__()  # Reiniciar la tabla de variables
   parser.parse(input_data)

def capture_output(function):
   f = StringIO()
   with redirect_stdout(f):
       function()
   return f.getvalue().strip()

def test_variable_declaration():
   data = '''
   program PanchoProgram;
   var
     pancho_age: int;
     pancho_weight: float;
   main { }
   end'''

   parse_input(data)

   expected_operandos = "[]"
   expected_operadores = "[]"
   expected_tipos = "[]"
   expected_saltos = "[]"
   expected_cuadruplos = "[]"

   actual_operandos = capture_output(lambda:
print(variable_table.pila_operandos))
   actual_operadores = capture_output(lambda:
print(variable_table.pila_operadores))
```

```
    actual_tipos = capture_output(lambda:
print(variable_table.pila_tipos))
    actual_saltos = capture_output(lambda:
print(variable_table.pila_saltos))
    actual_cuadruplos = capture_output(lambda:
print(variable_table.pila_cuadruplos))

    assert actual_operandos == expected_operandos
    assert actual_operadores == expected_operadores
    assert actual_tipos == expected_tipos
    assert actual_saltos == expected_saltos
    assert actual_cuadruplos == expected_cuadruplos


def test_arithmetic_operations():
    data = '''
    program PanchoProgram;
    var
      pancho_age: int;
      pancho_weight: float;
    main {
      pancho_age = 4 + 4;
      pancho_weight = pancho_age * 2.2;
    }
    end
    '''
    parse_input(data)

    expected_operandos = "[]"
    expected_operadores = "[]"
    expected_tipos = "[]"
    expected_saltos = "[]"
    expected_cuadruplos = "[['+', 501, 501, 200], ['=', 200, None, 0],
['*', 0, 601, 300], ['=', 300, None, 100]]"

    actual_operandos = capture_output(lambda:
print(variable_table.pila_operandos))
    actual_operadores = capture_output(lambda:
print(variable_table.pila_operadores))
    actual_tipos = capture_output(lambda:
print(variable_table.pila_tipos))
    actual_saltos = capture_output(lambda:
print(variable_table.pila_saltos))
```

```
    actual_cuadruplos = capture_output(lambda:
print(variable_table.pila_cuadruplos))

    assert actual_operandos == expected_operandos
    assert actual_operadores == expected_operadores
    assert actual_tipos == expected_tipos
    assert actual_saltos == expected_saltos
    assert actual_cuadruplos == expected_cuadruplos

def test_if_statement():
    data = '''
    program MyProgram;
    var
        pancho_age: int;
        pancho_weight: float;

    main
    {
        pancho_age = 8;

        if (pancho_age > 10) {
            print("Viejo");
        }else{
            print("ar");
        };
    }

    end
    '''
    parse_input(data)

    expected_operandos = "[]"
    expected_operadores = "[]"
    expected_tipos = "[]"
    expected_saltos = "[]"
    expected_cuadruplos = "[['=', 501, None, 0], ['>', 0, 502, 401],
['GOTOF', 401, None, 5], ['print', 'Viejo', None, None], ['GOTO', None,
None, 6], ['print', 'ar', None, None]]"

    actual_operandos = capture_output(lambda:
print(variable_table.pila_operandos))
    actual_operadores = capture_output(lambda:
print(variable_table.pila_operadores))
```

```
    actual_tipos = capture_output(lambda:
print(variable_table.pila_tipos))
    actual_saltos = capture_output(lambda:
print(variable_table.pila_saltos))
    actual_cuadruplos = capture_output(lambda:
print(variable_table.pila_cuadruplos))

    assert actual_operandos == expected_operandos
    assert actual_operadores == expected_operadores
    assert actual_tipos == expected_tipos
    assert actual_saltos == expected_saltos
    assert actual_cuadruplos == expected_cuadruplos


def test_while_loop():
    data = '''
    program PanchoProgram;
    var
      pancho_age: int;
      pancho_weight: float;
      x: int;

    main {
        pancho_age = 8;
        x = 5;

        if (pancho_age > 10) {
            print("Viejo");
        }else{
            print("ar");
        };
      do {
        print("Pancho ladra");
        x = x - 1;
      } while (x > 0);
    }
    end
    '''
    parse_input(data)

    expected_operandos = "[]"
    expected_operadores = "[]"
    expected_tipos = "[]"
```

```
    expected_saltos = "[]"
    expected_cuadruplos = "[['=', 501, None, 0], ['=', 502, None, 1],
['>', 0, 503, 401], ['GOTOF', 401, None, 6], ['print', 'Viejo', None,
None], ['GOTO', None, None, 7], ['print', 'ar', None, None], ['print',
'Pancho ladra', None, None], ['-', 1, 504, 200], ['=', 200, None, 1],
['>', 1, 505, 402], ['GOTOV', 402, None, 7]]"

    actual_operandos = capture_output(lambda:
print(variable_table.pila_operandos))
    actual_operadores = capture_output(lambda:
print(variable_table.pila_operadores))
    actual_tipos = capture_output(lambda:
print(variable_table.pila_tipos))
    actual_saltos = capture_output(lambda:
print(variable_table.pila_saltos))
    actual_cuadruplos = capture_output(lambda:
print(variable_table.pila_cuadruplos))

    assert actual_operandos == expected_operandos
    assert actual_operadores == expected_operadores
    assert actual_tipos == expected_tipos
    assert actual_saltos == expected_saltos
    assert actual_cuadruplos == expected_cuadruplos
```

Prueba Ejecucion (VM)

```
import sys
import os
import pytest
from io import StringIO
from contextlib import redirect_stdout

# Añadir el directorio raíz del proyecto al path de Python
sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from virtual_machine import VirtualMachine

def test_addition():
    const_table = {500: 10, 501: 20}
    quadruples = [
        ('+', 500, 501, 200),  # t0 = 10 + 20
```

```
        ('print', 200, None, None)  # print(t0)
    ]

    vm = VirtualMachine(const_table)
    vm.initialize_memory()

    for address, value in const_table.items():
        vm.set_memory(address, value)

    vm.load_quadruples(quadruples)

    f = StringIO()
    with redirect_stdout(f):
        vm.execute()

    output = f.getvalue().strip()
    assert output == '30'

def test_subtraction():
    const_table = {500: 30, 501: 10}
    quadruples = [
        ('-', 500, 501, 200),  # t0 = 30 - 10
        ('print', 200, None, None)  # print(t0)
    ]

    vm = VirtualMachine(const_table)
    vm.initialize_memory()

    for address, value in const_table.items():
        vm.set_memory(address, value)

    vm.load_quadruples(quadruples)

    f = StringIO()
    with redirect_stdout(f):
        vm.execute()

    output = f.getvalue().strip()
    assert output == '20'

def test_multiplication():
    const_table = {500: 5, 501: 4}
    quadruples = [
```

```
        ('*', 500, 501, 200),  # t0 = 5 * 4
        ('print', 200, None, None)  # print(t0)
    ]

    vm = VirtualMachine(const_table)
    vm.initialize_memory()

    for address, value in const_table.items():
        vm.set_memory(address, value)

    vm.load_quadruples(quadruples)

    f = StringIO()
    with redirect_stdout(f):
        vm.execute()

    output = f.getvalue().strip()
    assert output == '20'

def test_division():
    const_table = {500: 20, 501: 5}
    quadruples = [
        ('/', 500, 501, 200),  # t0 = 20 / 5
        ('print', 200, None, None)  # print(t0)
    ]

    vm = VirtualMachine(const_table)
    vm.initialize_memory()

    for address, value in const_table.items():
        vm.set_memory(address, value)

    vm.load_quadruples(quadruples)

    f = StringIO()
    with redirect_stdout(f):
        vm.execute()

    output = f.getvalue().strip()
    assert output == '4.0'

def test_comparison():
    const_table = {500: 10, 501: 20}
```

```
quadruples = [
    ('>', 500, 501, 400),   # t0 = 10 > 20
    ('print', 400, None, None)   # print(t0)
]

vm = VirtualMachine(const_table)
vm.initialize_memory()

for address, value in const_table.items():
    vm.set_memory(address, value)

vm.load_quadruples(quadruples)

f = StringIO()
with redirect_stdout(f):
    vm.execute()

output = f.getvalue().strip()
assert output == 'False'
```

3. Agrega a tu documentación técnica la descripción de las estructuras de datos usadas en la Máquina Virtual para simular el Mapa de Memoria y para manejar los Cuádruplos; incluye la descripción de los principales algoritmos desarrollados. para acceder a estas estructuras. De igual forma deberás describir el algoritmo con el que simulas el CPU.

   Descripción de las Estructuras de Datos
   - La clase VirtualMachine simula el mapa de memoria utilizando un diccionario (self.memory). Las direcciones de memoria están divididas en varias secciones para diferentes tipos de variables (globales, temporales, constantes) y tipos de datos (int, float, string).

   - Los cuádruplos se almacenan en una lista (self.quadruples) y cada cuádruplo es una lista de cuatro elementos: el operador, el operando izquierdo, el operando derecho y el resultado.

   Algoritmos:

   - Inicialización de Memoria: Inicializa diferentes secciones de la memoria para variables globales, temporales y constantes.
   - Carga de Cuádruplos: Carga la lista de cuádruplos a ejecutar.

- Ejecución de Cuádruplos: Itera sobre los cuádruplos y ejecuta las operaciones correspondientes, actualizando la memoria y manejando los saltos de control.

Algoritmo del CPU Simulado

```
    def execute(self):
        # Ejecutar los cuádruplos
        while self.instruction_pointer < len(self.quadruples):
            quad = self.quadruples[self.instruction_pointer]
            op = quad[0]
            left = quad[1]
            right = quad[2]
            result = quad[3]

            if op == '+':
                self.memory[result] = self.memory[left] +
self.memory[right]
            elif op == '-':
                self.memory[result] = self.memory[left] -
self.memory[right]
            elif op == '*':
                self.memory[result] = self.memory[left] *
self.memory[right]
            elif op == '/':
                self.memory[result] = self.memory[left] /
self.memory[right]
            elif op == '=':
                self.memory[result] = self.memory[left]
            elif op == '!=':
                self.memory[result] = self.memory[left] !=
self.memory[right]
            elif op == '>':
                self.memory[result] = self.memory[left] >
self.memory[right]
            elif op == '<':
                self.memory[result] = self.memory[left] <
self.memory[right]
            elif op == 'GOTO':
                self.instruction_pointer = result
                continue
            elif op == 'GOTOV':
                if self.memory[left]:
                    self.instruction_pointer = result
```

```
                continue
            elif op == 'GOTOF':
                if not self.memory[left]:
                    self.instruction_pointer = result
                    continue
            elif op == 'print':
                if type(left) == str:
                    print(left)
                else:
                    print(self.memory[left])
            else:
                raise Exception(f"Unknown operator: {op}")

            self.instruction_pointer += 1
```

Este algoritmo simula un CPU básico, ejecutando cuádruplos y actualizando el estado de la memoria en cada paso.

Log de Cambios

Implementación de las Pilas y la Fila

- Pilas Implementadas:
    - pila_operadores: Para almacenar operadores.
    - pila_operandos: Para almacenar operandos.
    - pila_tipos: Para almacenar los tipos de datos de los operandos.
    - pila_saltos: Para manejar las direcciones de salto.
- Fila Implementada:
    - pila_cuadruplos: Para almacenar los cuádruplos generados durante la traducción.

2. Algoritmos de Traducción a Cuádruplos

- Expresiones Aritméticas y Relacionales:
    - Implementación de la generación de cuádruplos para operaciones aritméticas (+, -, *, /).
    - Implementación de la generación de cuádruplos para operaciones relacionales (<, >, !=).
- Estatutos Lineales:
    - Asignaciones (ID = expresion).
    - Impresiones (print(expresion)).
- Estatutos No-Lineales:
    - Condiciones (if (expresion) { cuerpo } else { cuerpo }).
    - Ciclos (do { cuerpo } while (expresion)).

3. Máquina Virtual

- Mapa de Memoria:
    - Se añadió la clase VirtualMachine que simula el mapa de memoria utilizando un diccionario.
    - Se implementaron métodos para inicializar la memoria, cargar cuádruplos y ejecutar los cuádruplos.
- Ejecución de Cuádruplos:
    - Implementación del algoritmo del CPU para ejecutar los cuádruplos.
    - Manejo de operaciones aritméticas, relacionales y de control de flujo (GOTO, GOTOF, GOTOV).

4. Pruebas

- Pruebas para el Lexer:
    - Se añadieron pruebas para palabras reservadas, delimitadores, expresiones aritméticas, literales y estructuras condicionales.
- Pruebas para el Parser:
    - Se añadieron pruebas para la declaración de variables, asignaciones simples, operaciones aritméticas y declaraciones de impresión.
- Pruebas para la Máquina Virtual:

    - Se añadieron pruebas para validar operaciones simples de suma y la instrucción de impresión.

5. Documentación Técnica

- Descripción de las Estructuras de Datos:
    - Descripción detallada del mapa de memoria y los cuádruplos.
    - Explicación de los principales algoritmos desarrollados para manejar estas estructuras.
- Algoritmo del CPU:
    - Se añadió una explicación del algoritmo del CPU utilizado en la máquina virtual para ejecutar los cuádruplos.

6. Modificaciones Adicionales

- Eliminación de Funciones:
    - Se eliminaron las declaraciones y el manejo de funciones del parser.
- Eliminación de Números Negativos:
    - Se removió el soporte para números negativos en el lexer y parser.
- Reestructuración de la Gramática:
    - Se ajustaron las reglas gramaticales para simplificar la implementación y alinearse con los nuevos requerimientos.

Resumen de Pruebas

Lexer

- Pruebas Añadidas:
    - test_reserved_keywords_and_delimiters
    - test_arithmetic_expressions
    - test_floats_and_strings
    - test_conditionals

Parser

- Pruebas Añadidas:
    - test_variable_declaration
    - test_simple_assignment
    - test_arithmetic_operations
    - test_print_statement

Máquina Virtual

- Pruebas Añadidas:
    - test_vm_simple_addition
    - test_vm_print

Referencias:

About the ANTLR Parser Generator. (s. f.). https://www.antlr.org/about.html

ply. (2018, 15 febrero). PyPI. https://pypi.org/project/ply/

SLY (Sly Lex Yacc) — sly 0.0 documentation. (s. f.).
https://sly.readthedocs.io/en/latest/sly.html#