



# Tecnológico de Monterrey

Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 503)

## **Documentación de Compilador BABY DUCK**

**Profesores:**

**Mauricio González Soto**

**Ivan Mauricio Amaya Contreras**

**Elda Guadalupe Quiroga González**

Valeria Enríquez Limón

A00832782

Junio 01 2025

# Primeras Definiciones

## Lenguaje de Programación:

Un lenguaje formal que proporciona instrucciones para crear programas que controlan el comportamiento de una computadora. Permite definir algoritmos, estructurar datos y comunicarse con el sistema. Durante esta entrega utilicé Python debido a su versatilidad y su baja complejidad de sintaxis.

### PLY (Python Lex-Yacc):

Herramienta para construir lexers (analizadores léxicos) y parsers (analizadores sintácticos) en Python. Implementa versiones simplificadas de las herramientas clásicas lex y yacc, usando métodos como `tokenize()` para el léxico y reglas gramaticales para la sintaxis.

### ANTLR (Another Tool for Language Recognition):

Generador de parsers más potente y versátil que soporta múltiples lenguajes de salida (Java, Python, C++, etc.). Usa gramáticas descriptivas para generar lexers y parsers, ideal para lenguajes complejos.

### ¿Por qué usé PLY?

- Simplicidad: Ideal para proyectos pequeños o medianos en Python.
- Integración: Se adapta bien a entornos Python sin requerir configuraciones externas.
- Flexibilidad: Permite definir tokens y gramáticas directamente en código Python.
- Ligero: Menos overhead que ANTLR para prototipos o compiladores sencillos.

(Nota: Si tu proyecto maneja gramáticas muy complejas o múltiples lenguajes objetivo, ANTLR podría ser mejor opción).

### Partes del Compilador

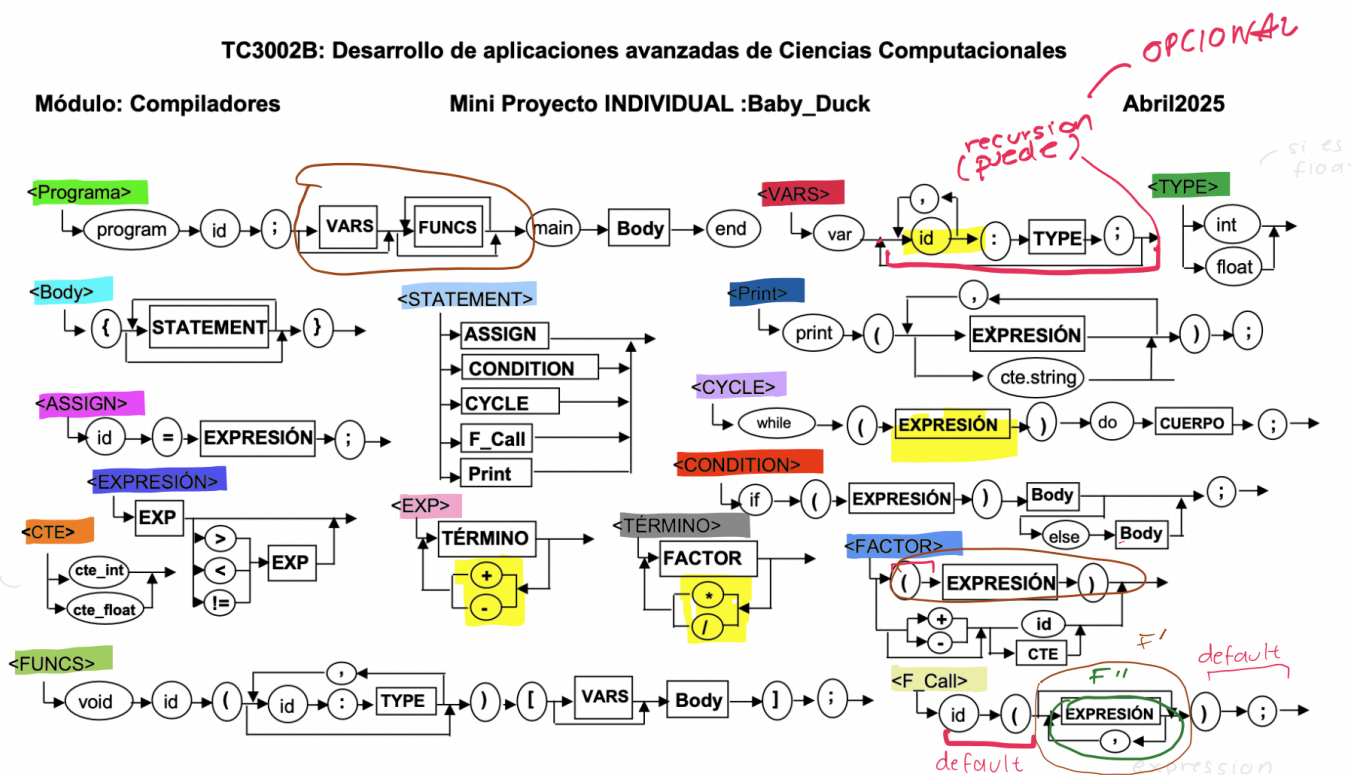
1. Lexer (Analizador Léxico):  
Convierte el código fuente en tokens (unidades significativas como palabras clave, operadores, identificadores).
2. Parser (Analizador Sintáctico):  
Verifica que la secuencia de tokens cumpla con la gramática del lenguaje y genera un árbol sintáctico.
3. Cubo Semántico:  
Estructura que define las reglas de compatibilidad entre operadores y tipos de datos (ej: no sumar entero + string).
4. Tabla de Variables:  
Registro que almacena información de variables (nombre, tipo, ámbito, dirección de memoria).

5. **Tabla de Funciones:**  
Almacena detalles de funciones (parámetros, tipo de retorno, dirección en memoria, ámbito).
6. **Máquina Virtual:**  
Ejecuta el código intermedio (ej: bytecode) generado por el compilador, simulando un CPU con manejo de memoria, pilas y registros.

## Principales Estructuras de Datos Utilizadas

- **Stack:** En el diseño de compiladores, un stack (o pila) es una estructura de datos fundamental que opera bajo el principio LIFO (Last In, First Out), donde el último elemento añadido es el primero en ser removido. Esta estructura es crítica en múltiples fases de la compilación, como la gestión de ámbitos anidados durante el análisis semántico o la evaluación de expresiones en la máquina virtual. Por ejemplo, al procesar bloques de código con variables locales, cada nuevo ámbito (como el cuerpo de una función o un bucle) se apila, y al finalizarlo, se desapila, eliminando automáticamente las variables declaradas en ese ámbito. Además, en la generación de código intermedio, el stack facilita la traducción de expresiones aritméticas complejas a notación polaca inversa, permitiendo un manejo eficiente de operaciones y operandos. Su simplicidad y eficiencia (operaciones en tiempo constante  $O(1)$ ) lo hacen ideal para tareas que requieren un orden estricto de procesamiento.
- **Diccionarios:** Los diccionarios, o tablas hash, son estructuras de datos que almacenan pares clave-valor, donde cada clave única permite acceder a un valor asociado de manera eficiente ( $O(1)$  en el caso ideal). En compiladores, los diccionarios son la base para implementar tablas de símbolos, que almacenan información sobre variables, funciones, y tipos durante las fases de análisis. Por ejemplo, una tabla de símbolos puede mapear el nombre de una variable (clave) a sus atributos (tipo, ámbito, dirección de memoria). Su capacidad de inserción, búsqueda y eliminación rápida los hace indispensables para gestionar el contexto del programa, especialmente en lenguajes con ámbitos dinámicos o sobrecarga de funciones. Además, en la optimización de código, los diccionarios ayudan a detectar redundancias (como expresiones repetidas) mediante el hashing de patrones. A diferencia del stack, los diccionarios no imponen un orden secuencial, pero ofrecen flexibilidad para acceder y modificar datos de forma no lineal, lo que es crucial en entornos donde la rapidez de acceso a información dispersa es prioritaria.
- Ambas estructuras, aunque con propósitos distintos, son complementarias: el stack asegura un manejo ordenado y jerárquico de información efímera, mientras que los diccionarios proporcionan acceso rápido a datos distribuidos, formando así la columna vertebral de la gestión de memoria y metadatos en un compilador.

## Diagrama de Baby Duck



Como puedes darte cuenta, **Baby\_Duck** es un micro-lenguaje imperativo procedural clásico. Para él deberás diseñar, como etapa 0: la gramática formal equivalente a estos diagramas, así como las expresiones regulares que representen a los tokens (a tu gusto). Implementar, como etapa 1, el proceso de análisis de léxico y sintaxis (se recomienda utilizar un generador automático de Scanners&Parsers, de los que investigados). Posteriormente harás las validaciones semánticas pertinentes para luego generar la representación equivalente en código intermedio. (etapas 2, 3 y 4) Finalmente, desarrollarás una Máquina Virtual que sea capaz de interpretar el código intermedio para ejecutar el programa inicial. (etapa final)

## Inicio de Reglas Gramaticales

Las reglas gramaticales son un conjunto de producciones que definen la estructura sintáctica de un lenguaje formal. A diferencia de las expresiones regulares, que se enfocan en el análisis léxico, las reglas gramaticales permiten describir cómo se combinan los símbolos y palabras para formar frases válidas dentro del lenguaje, es decir, su sintaxis. Estas reglas suelen representarse mediante gramáticas libres de contexto (CFG, por sus siglas en inglés), las cuales son ampliamente utilizadas en el diseño de lenguajes de programación y en la construcción de analizadores sintácticos. Una gramática está compuesta por un conjunto de símbolos terminales, no terminales, un símbolo inicial y un conjunto de producciones que especifican cómo los no terminales pueden transformarse. En esta sección se presentan las principales reglas gramaticales del lenguaje en estudio, así como su notación y análisis estructural.

1.  $cte \rightarrow cte - \text{int}$   
 $cte \rightarrow cte - \text{float}$
2.  $\text{type} \rightarrow \text{int}$   
 $\text{type} \rightarrow \text{float}$
3.  $\text{EXP} \rightarrow \text{TEMINO } E'$   
 $\text{op} \rightarrow +$   
 $\text{op} \rightarrow -$   
 $E \rightarrow \text{OP EXP}$   
 $E \rightarrow \varepsilon$
4.  $\text{TEMINO} \rightarrow \text{FACTOR } E'$   
 $\text{op} \rightarrow *$   
 $\text{op} \rightarrow /$   
 $E \rightarrow \text{op TEMINO}$   
 $E \rightarrow \varepsilon$

5. Statement  $\rightarrow$  assign  
 Statement  $\rightarrow$  condition  
 Statement  $\rightarrow$  cycle

Statement  $\rightarrow$  #-call  
 Statement  $\rightarrow$  Print

6. FUNCS  $\rightarrow$  void id ( F' ) [ V' Body ] ?  
 F'  $\rightarrow$  id : TYPE F'' F'''  
 F''  $\rightarrow \epsilon$   
 F''  $\rightarrow$  ,  
 F'''  $\rightarrow \epsilon$   
 F'''  $\rightarrow$  id : TYPE F'' F'''  
 V'  $\rightarrow \epsilon$   
 V'  $\rightarrow$  VARS V'

7. Body  $\rightarrow$  { S' }  
 s'  $\rightarrow$  STATEMENTS  
 s'  $\rightarrow \epsilon$

8. ASSIGN  $\rightarrow$  id = EXPRESSION

9. PRINT  $\rightarrow$  print ( P' ) ;  
 P'  $\rightarrow$  EXPRESSION P''  
 P''  $\rightarrow \epsilon$   
 P''  $\rightarrow$  , C'  
 C'  $\rightarrow \epsilon$   
 C'  $\rightarrow$  cte.strings

10. CONDITION  $\rightarrow$  if ( EXPRESSION ) BODY C'  
 C'  $\rightarrow \epsilon$   
 C'  $\rightarrow$  else Body

11. Expression  $\rightarrow$  EXP E' EXP  
 E'  $\rightarrow$  > | < | ! = | =  
 E'  $\rightarrow \epsilon$

12. TERM  $\rightarrow$  id | c.te | ( EXPRESSION )

13. programa  $\rightarrow$  program id, PP' main Body end  
 P  $\rightarrow \epsilon$   
 P  $\rightarrow$  VARS  
 P'  $\rightarrow \epsilon$   
 P  $\rightarrow$  FUNCS P'

14. VARS  $\rightarrow$  var id V' : TYPE ;  
 V'  $\rightarrow$  , id V'  
 V'  $\rightarrow \epsilon$   
 V''  $\rightarrow \epsilon$   
 V''  $\rightarrow$  id V' : TYPE ; V''

Token	Lexema(s)	Expresión Regular
PROGRAM	program	program
ID	Identificador	[a-zA-Z_] [a-zA-Z0-9_]*
SEMICOLON	;	;
COLON	:	:
COMMA	,	,
LPAREN	(	\(
RPAREN	)	\)
LBRACE	{	{
RBRACE	}	}
MAIN	main	main
END	end	end
VAR	var	var
TYPE	int, float	int float
VOID	void	void
ASSIGN	=	=
EQ	==	==
NEQ	!=	!=
LT	<	<
GT	>	>
PLUS	+	\+
MINUS	-	-
MULT	*	\*
DIV	/	/
IF	if	if
ELSE	else	else
WHILE	while	while
DO	do	do
PRINT	print	print
CTE_INT	Constante entera	[0-9]+
CTE_FLOAT	Constante flotante	[0-9]+\.[0-9]+

**Table 1.** Tokens del lenguaje Baby\_Duck y sus expresiones regulares

# Lexer

En la primera fase de un compilador, el scanner (también llamado analizador léxico o lexer) se encarga de traducir la secuencia bruta de caracteres del programa fuente en una secuencia más manejable de objetos llamados tokens. Cada token agrupa uno o varios caracteres que juntos tienen un significado sintáctico: palabras clave, identificadores, literales numéricos o de texto, operadores, signos de puntuación, etc.

## - Palabras reservadas:

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

## - Tokens:

```
tokens = [  
  'ID',      # Identificador (por ejemplo: variable, nombre de función)  
  'CTE_STRING', # Constante de cadena (string) -> "hola", 'mundo'  
  'CTE_INT',   # Constante entera -> 123, -5  
  'CTE_FLOAT', # Constante de punto flotante -> 3.14, -0.01  
  'PLUS',     # Símbolo de suma: +  
  'MINUS',    # Símbolo de resta: -  
  'TIMES',    # Símbolo de multiplicación: *  
  'DIVIDE',   # Símbolo de división: /  
  'EQUALS',   # Símbolo de asignación o comparación: =  
  
  'LPAREN',   # Paréntesis izquierdo: (  
  'RPAREN',   # Paréntesis derecho: )  
  'SEMICOLON', # Punto y coma: ;  
  'COLON',    # Dos puntos: :  
  'COMMA',    # Coma: ,  
  'LBRACE',   # Llave izquierda: {
```



```

'RBACE',    # Llave derecha: }
'LT',       # Menor que: <
'GT',       # Mayor que: >
'NE'        # Diferente de (Not Equal): !=
]
+ list(reserved.values())

```

#### - Tokens:

```

t_GT = r'>'
t_LT = r'<'
t_NOTEQUAL = r'!='
t_PLUS = r'\+'
t_MINUS = r'\-'
t_TIMES = r'\*'
t_DIVIDE = r'/'

t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_COMMA = r','
t_COLON = r':'
t_SEMICOLON = r';'
t_DOUBLEEQUAL = r'=='
t_EQUAL = r'='

```

# → Saltar espacios horizontales, para poder contar lineas

```
t_ignore = ' \t\r'
```

#### - Reconocimiento de Sintaxis:

# → número flotante

```

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

```

# → número entero

```

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

```

# → quitar comillas

```
def t_CTE_STRING(t):  
    r'"([^\\n]|(\\.))*?"'  
    t.value = t.value[1:-1]  
    return t
```

# → verificar palabra reservada, de esta manera evitamos usar palabras como: if, while, return, etc.

```
def t_ID(t):  
    r'[A-Za-z_][A-Za-z0-9_]*'  
    t.type = reserved.get(t.value.lower(), 'ID')  
    return t
```

# → verificar tipo

```
def t_TYPE(t):  
    r'int|float'  
    return t
```

# → lineas nuevas

```
def t_newline(t):  
    r'\n+'  
    t.lexer.lineno += len(t.value)
```

# → agregar comentario

```
def t_COMMENT(t):  
    r'\#.*'  
    pass
```

# → errores encontrados

```
def t_error(t):  
    print(f"Carácter ilegal '{t.value[0]}' en la linea {t.lineno}")  
    t.lexer.skip(1)
```

## - Construir el Lexer:

```
lexer = lex.lex(debug=1)  
data_path = Path(__file__).with_name("archivo.txt")
```

with data\_path.open("r", encoding="utf-8") as file:

```
    data = file.read()  
lexer.input(data)  
while True:  
    tok = lexer.token()  
    if tok.value == 'end':  
        break  
    print("Token: ", tok)
```

# Parser

*"A parse is only successful if the parser reaches a state where the symbol stack is empty and there are no more input tokens. It is important to note that the underlying implementation is built around a large finite-state machine that is encoded in a collection of tables. The construction of these tables is non-trivial and beyond the scope of this discussion. However, subtle details of this process explain why, in the example above, the parser chooses to shift a token onto the stack in step 9 rather than reducing the rule  $\text{expr} \rightarrow \text{expr} + \text{term}$ ."*

Es el arquitecto que comprueba que esas fichas encajan siguiendo el reglamento del idioma del lenguaje y construye la primera representación estructurada del programa.

Comprobar la sintaxis: verifica que la secuencia de tokens cumple la gramática libre de contexto definida para el lenguaje (las reglas tipo  $\text{Stmt} \rightarrow \text{Expr};$ ,  $\text{Expr} \rightarrow \text{Expr} + \text{Term}$ , etc.)

Construir estructura: genera un árbol de análisis —normalmente un AST (Abstract Syntax Tree)— que captura la jerarquía lógica del código y servirá a las fases posteriores (semántica, generación de código, optimizaciones).

- En el parser no todo depende del orden, por ejemplo la tabla LR(1), ya que PLY analiza todas las reglas antes de construir los automatas; por lo que no ejecuta las funciones secuencialmente.
- La asociatividad o precedencia de los operadores depende sólo de la variable precedence, y no de donde se encuentre la regla en el archivo.
- Debido a que no declaré explícitamente la variable de 'start', se tomará la primera regla como 'símbolo inicial'.
- Se deben agrupar las reglas relacionadas y documentar cada no terminal.

## Precedencia y asociatividad utilizada

```
precedence = (  
    ('left', 'PLUS', 'MINUS'),  
    ('left', 'TIMES', 'DIVIDE'),  
    ('left', 'LT', 'GT', 'NE')  
)
```

En un compilador, la precedencia y asociatividad son reglas que determinan el orden en que se evalúan los operadores cuando una expresión tiene múltiples operadores (ej:  $3 + 5 * 2$ ). En PLY, estas reglas se definen en la tupla precedence para resolver ambigüedades en la gramática.

## Reglas Aplicadas:

1. Prioridad (de mayor a menor):
  - LT, GT, NE (comparaciones) → Mayor prioridad.
  - TIMES, DIVIDE → Prioridad media.
  - PLUS, MINUS → Menor prioridad.
2. Asociatividad (left):
  - Los operadores se agrupan de izquierda a derecha.
  - Ejemplo:
    - $5 - 3 + 2 \rightarrow (5 - 3) + 2$  (asociatividad izquierda).
    - $10 / 2 * 3 \rightarrow (10 / 2) * 3$ .

## Parser

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

**Define la estructura general de un programa: declaración de variables, funciones, el cuerpo principal y cierre.**

```
def p_a_vars(p):  
    """a_vars : empty  
    | vars"""  
    variables_list = p[1]  
    for var in variables_list:  
        the_name, the_type = var.split(':')  
        st.push_variable(the_name, the_type)
```

**Procesa y almacena las declaraciones de variables globales o locales.**

```
def p_vars(p):  
    """vars : MY_VARS ID COLON type SEMICOLON list_vars"""  
    list_vars = p[6]  
    p[0] = (f'{p[2]}:{p[4]}', *list_vars.split(','))
```

**Concatena una declaración de variable con las siguientes (si existen).**

```
def p_list_vars(p):  
    """list_vars : empty  
    | ID COLON type SEMICOLON list_vars"""  
    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]}'
```

**Interpreta una lista adicional de variables separadas por punto y coma.**

```
def p_type(p):  
    """type : INT  
    | FLOAT"""  
    p[0] = p[1]
```

**Asigna el tipo de dato a una variable.**

```
def p_body(p):  
    'body : LBRACE list_statements RBRACE'  
    pass
```

**Representa un bloque de código delimitado por llaves.**

```
def p_list_statements(p):  
    '''list_statements : statement body_rep  
        | empty  
        | statement'''  
    pass
```

```
def p_body_rep(p):  
    '''body_rep : statement body_rep  
        | statement'''  
    pass
```

**Agrupar múltiples sentencias dentro de un bloque.**

```
def p_statement(p):  
    '''statement : assign  
        | condition  
        | do_cycle  
        | cycle  
        | f_call  
        | print_statement'''  
    pass
```

**Define los posibles tipos de sentencias.**

```
def p_print_stmt(p):  
    'print_statement : PRINT LPAREN list_expresion RPAREN SEMICOLON'  
    pass
```

**Regla que construye una instrucción de impresión.**

```
def p_assign(p):  
    'assign : ID add_operand EQUALS add_operador expresion SEMICOLON'  
    st.add_assing()
```

**Genera un cuádruplo de asignación a partir del identificador, operador y resultado de expresión.**

```
def p_add_operand(p):  
    '''add_operand : '''  
    st.add_operand(p[-1])
```

```
def p_add_operador(p):  
    '''add_operador : '''  
    st.push_operador(p[-1])
```

**Agregan el operando y el operador correspondiente a sus pilas.**

```
def p_list_expresion(p):  
    '''list_expresion : expresion addPrint  
        | expresion addPrint COMMA list_expresion  
        | CTE_STRING addPrintString  
        | CTE_STRING addPrintString COMMA list_expresion'''
```

```

    pass
    Permite imprimir múltiples expresiones o cadenas.
    def p_addPrint(p):
        "addPrint : "
        st.add_print()

```

```

    def p_addPrintString(p):
        "addPrintString : "
        st.add_print(string=p[-1])

```

**Invocan funciones para registrar que se imprimirá una variable o una cadena.**

```

    def p_condition(p):
        'condition : IF LPAREN expresion RPAREN goto_false body else_part'
        st.add_goto_False_fill()

```

```

    def p_else_part(p):
        "else_part : ELSE goto body
        | empty"
        pass

```

**Gramática para condiciones IF - ELSE**

```

    def p_goto_false(p):
        "goto_false : "
        # Añadir la instrucción de salto condicional para if
        st.add_goto_false()

```

```

    def p_goto(p):
        "goto : "
        # Añadir una instrucción de salto incondicional
        st.add_goto()

```

**Gramática para GOTO-S DE IF - ELSE**

```

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

```

```

    def p_comparar_exp(p):
        "comparar_exp : LT
        | GT
        | NE"
        # Añadir un operador de comparación a la pila
        st.push_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
        st.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
    st.push_operador(p[1])
    pass

```

### Gramática para operaciones matemáticas

```

def p_termino(p):
    """termino : factor add_factor next_factor
    | factor add_factor"""
    Pass

```

### Gramática para evaluación de términos + -

```

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
    st.push_operador(p[1])
    pass

```

### Gramática para evaluación de factores \* /

```

def p_factor(p):
    """factor : LPAREN expresion RPAREN
    | id_cte"""
    pass

def p_add_factor(p):
    """add_factor : """
    # Añadir un factor a la pila
    st.add_factor()

```

### Gramática para Factores y añadir más

```

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
    st.add_operand(p[-1], True)

def p_cte(p):
    """cte : CTE_INT

```

```

    | CTE_FLOAT""
# Manejar constantes enteras y de punto flotante
p[0] = p[1]

```

### Gramática para definir ID y CTES

```

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

```

### Gramática para añadir valores a tabla de variables

```

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_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

```

```

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

```



## Gramática para manejar Funciones

```
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")
```

## Gramática para manejar Errores

### Construir el parser

```
def build_parser():  
    return yacc.yacc(write_tables=False, debug=False)
```

## Cubo Semántico

Es el arquitecto que comprueba que esas fichas encajen siguiendo el reglamento del idioma del lenguaje y construye la primera representación estructurada del programa.

Comprobar la sintaxis: verifica que la secuencia de tokens cumple la gramática libre de contexto definida para el lenguaje (las reglas tipo Stmt  $\rightarrow$  Expr ';', Expr  $\rightarrow$  Expr '+' Term, etc.).

```
def semantic_validation(left_operand, right_operand, operator):  
    left_parts = left_operand.split('.')  
    right_parts = right_operand.split('.')  
  
    if not left_parts or not right_parts:  
        raise ValueError(f"Malformed operand(s): '{left_operand}', '{right_operand}'")  
  
    left_type = left_parts[-1]  
    right_type = right_parts[-1]  
  
    if operator not in semantic_cube:  
        raise ValueError(f"Unknown operator: '{operator}'")  
  
    operator_table = semantic_cube[operator]  
  
    if right_type not in operator_table:  
        raise ValueError(f"Operator '{operator}' does not support type '{right_type}' as right operand")  
  
    if left_type not in operator_table[right_type]:  
        raise ValueError(f"Operator '{operator}' does not support type '{left_type}' as left operand with right type '{right_type}'")  
  
    result_type = operator_table[right_type][left_type]  
  
    if result_type == 'error':  
        print(f"Invalid operation: {right_type} {operator} {left_type}")  
  
    return result_type
```

La función `semantic_validation` realiza una verificación semántica entre dos operandos y un operador, utilizando un cubo semántico predefinido para determinar si la operación es válida según los tipos de datos involucrados. Primero, descompone cada operando (que se asume en formato `contexto.tipo`, como `global.int`) para extraer su tipo base (ej: `int`). Luego, valida que ambos operandos estén correctamente formados y consulta el cubo semántico —una estructura jerárquica que define las reglas de compatibilidad entre operadores y tipos— para verificar si el operador soporta los tipos de los operandos. Si el operador o los tipos no son compatibles, lanza excepciones descriptivas (ej: `Operator '+' does not support type 'string' as right operand`). Finalmente, retorna el tipo resultante de la operación (ej: `float` para `int + float`) o un error si la combinación es inválida. Esta función es clave en la fase de análisis semántico de un compilador, asegurando que las operaciones cumplan con las reglas del lenguaje antes de generar código.

## Tabla de Símbolos

```
def __init__(self):
    self.var_table = {}
    self.constant_table = {}

    self.semantic_cube = semantic_validation
    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.stack_operators = []
    self.stack_operands = []
    self.stack_types = []
    self.stack_jumps = []
    self.stack_quadruples = []

    def clear(self):
        # Resetear todos los stacks
        self.var_table.clear()
        self.stack_operands.clear()
        self.stack_operators.clear()
        self.stack_types.clear()
        self.stack_jumps.clear()
        self.stack_quadruples.clear()
        self.constant_table.clear()
```

La clase `SymbolTable` es responsable de gestionar la memoria simbólica de un compilador, encapsulando tanto las estructuras para el almacenamiento de variables como el control de la generación semántica de cuádruplos. En su constructor `__init__`, se inicializan los diccionarios `var_table` y `constant_table` para almacenar las variables declaradas y las constantes utilizadas, respectivamente. También se define un cubo semántico que permite validar operaciones entre tipos de datos. Para simular la memoria, se asignan rangos de direcciones diferenciadas para variables enteras, flotantes, temporales, booleanas y constantes, permitiendo una organización clara y segmentada. Además, la clase mantiene pilas auxiliares (`stack_operands`, `stack_operators`, `stack_types`, `stack_jumps`, `stack_quadruples`) utilizadas durante el análisis semántico y la generación de código intermedio. Por último, el método `clear()` permite limpiar completamente el estado de la tabla entre ejecuciones o pruebas, garantizando un entorno limpio para nuevas compilaciones.

```
def push_variable(self, name, var_type):
    if name in self.var_table:
        raise ValueError(f"Variable '{name}' already exists in the var_table.")
    self.var_table[name] = {'type': var_type, 'memory_address': 0}
    self.add_memory_address(name)
    self.debug_stacks("después de add_operand")

def get_variable(self, name):
    return self.var_table.get(name, None)
    self.debug_stacks("después de add_operand")
```

Los métodos `push_variable` y `get_variable` pertenecen a la clase encargada de manejar la tabla de símbolos en el compilador. El método `push_variable` tiene como objetivo registrar una nueva variable en el diccionario `var_table`, asegurándose primero de que el identificador no haya sido previamente declarado para evitar conflictos. Una vez validado, asigna temporalmente una dirección de memoria genérica y delega a `add_memory_address` la tarea de asignar una dirección real conforme al tipo de dato. Además, invoca una función de depuración (`debug_stacks`) para mostrar el estado actual de las estructuras internas. Por su parte, `get_variable` permite recuperar la información asociada a una variable previamente declarada. Esta función consulta el diccionario `var_table` y devuelve los metadatos de la variable —como su tipo o dirección de memoria— si existe, o `None` si no ha sido registrada. Ambos métodos son fundamentales para gestionar la memoria simbólica y garantizar la integridad del análisis semántico y la generación de código.

```
def add_memory_address(self, name):
    if self.var_table[name]['type'] == 'int':
        memory_address = self.var_int
        self.var_int += 1
    elif self.var_table[name]['type'] == 'float':
        memory_address = self.var_float
        self.var_float += 1
    self.var_table[name]['memory_address'] = memory_address
```

El método `add_memory_address` es responsable de asignar una dirección de memoria virtual a una variable recién registrada en la tabla de símbolos. Utiliza como criterio el tipo de dato de la variable (`int` o `float`), previamente almacenado en el campo `'type'` del diccionario `var_table`. Dependiendo del tipo, la función selecciona el segmento correspondiente en memoria (`self.var_int` para enteros o `self.var_float` para flotantes) y le asigna la siguiente dirección disponible. Luego, incrementa el contador del segmento para futuras asignaciones y actualiza el campo `'memory_address'` de la variable con la dirección asignada. Esta estrategia garantiza una organización lineal y no superpuesta de las direcciones en la memoria simbólica, facilitando la interpretación y ejecución del programa en etapas posteriores del compilador.

```

def push_operador(self, operator):
    self.stack_operators.append(operator)
    self.debug_stacks("después de add_operand")

def push_operand(self, operand, is_cte=False):
    try:
        if is_cte:
            # 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.stack_operands.append(self.constant_table[operand])
                self.stack_types.append(tipo)
                self.debug_stacks("add_operand: int")
            elif tipo == 'float':
                if operand not in self.constant_table:
                    self.var_const_float += 1
                    self.constant_table[operand] = self.var_const_float
                self.stack_operands.append(self.constant_table[operand])
                self.stack_types.append(tipo)
                self.debug_stacks("add_operand: float")
            else:
                # Añadir la dirección de memoria de la variable
                tipo = self.get_variable(operand)["type"]
                dir_memoria = self.get_variable(operand)['memory_address']
                self.stack_operands.append(dir_memoria)
                self.stack_types.append(tipo)
                self.debug_stacks("add_operand")
    except:
        raise KeyError(f"Variable '{operand}' was not declared.")

```

El método `push_operador` agrega un operador (como `+`, `-`, `=`, etc.) a la pila `stack_operators`, la cual se utiliza para mantener el control de la precedencia y el orden de operaciones durante el análisis semántico. Después de cada inserción, se llama a `debug_stacks` para facilitar la depuración del estado actual de las pilas involucradas.

Por otro lado, `push_operand` inserta operandos en la pila `stack_operands` y registra sus respectivos tipos en la pila `stack_types`. Si el operando es una constante (`is_cte=True`), el método identifica su tipo (`int` o `float`), asigna una dirección de memoria virtual única si no ha sido registrada previamente, y la guarda en la tabla de constantes `constant_table`. Si el operando es una variable, se recupera su tipo y dirección de memoria desde la tabla de símbolos `var_table`. En ambos casos, se actualizan las pilas de operandos y tipos, y se invoca `debug_stacks` para visualizar el estado actual. Si el operando no ha sido declarado, se lanza una excepción clara, lo que permite detectar errores semánticos de manera eficiente durante la compilación.

```

def add_factor(self):
    if self.stack_operators:
        op = self.stack_operators[-1]
        if op == '*' or op == '/':
            right_operand = self.stack_operands.pop()
            left_operand = self.stack_operands.pop()
            right_operand_tipo = self.stack_types.pop()
            left_operand_tipo = self.stack_types.pop()
            operator = self.stack_operators.pop()
            res_tipo = self.semantic_cube(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.stack_operands.append(res_address)
                self.var_temp_int += 1
                self.stack_types.append(res_tipo)
                self.stack_quadruples.append([operator, left_operand, right_operand, res_address])
                self.debug_stacks("add_factor: int")
            elif res_tipo == 'float':
                res_address = self.var_temp_float
                self.stack_operands.append(res_address)
                self.var_temp_float += 1
                self.stack_types.append(res_tipo)
                self.stack_quadruples.append([operator, left_operand, right_operand, res_address])
                self.debug_stacks("add_factor: float")

def add_termino(self):
    # Añadir un término (suma o resta)
    if self.stack_operators:
        op = self.stack_operators[-1]
        if op == '+' or op == '-':
            right_operand = self.stack_operands.pop()
            left_operand = self.stack_operands.pop()
            right_operand_tipo = self.stack_types.pop()
            left_operand_tipo = self.stack_types.pop()
            operator = self.stack_operators.pop()
            res_tipo = self.semantic_cube(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.stack_operands.append(res_address)
                self.var_temp_int += 1
                self.stack_types.append(res_tipo)
                self.stack_quadruples.append([operator, left_operand, right_operand, res_address])
                self.debug_stacks("add_termino: int")
            elif res_tipo == 'float':
                res_address = self.var_temp_float
                self.stack_operands.append(res_address)
                self.var_temp_float += 1
                self.stack_types.append(res_tipo)
                self.stack_quadruples.append([operator, left_operand, right_operand, res_address])
                self.debug_stacks("add_termino: float")

```

Los métodos `add_factor` y `add_termino` están diseñados para procesar expresiones aritméticas durante la generación de cuádruplos en un compilador. Ambos operan sobre pilas semánticas que contienen operandos, tipos de datos y operadores, y ejecutan validaciones semánticas a través de un cubo semántico (`semantic_cube`).

El método `add_factor` se activa cuando el operador en la cima de la pila es `*` o `/`, lo que indica una operación de multiplicación o división. Este método extrae los dos operandos y sus tipos, valida la operación con el cubo semántico y, si es válida, genera una nueva dirección temporal (entera o flotante según el tipo de resultado), actualiza las pilas, y genera el cuádruplo correspondiente con el operador y los operandos.

De manera análoga, `add_termino` se encarga de las operaciones de suma y resta (`+` o `-`). El proceso sigue la misma lógica: extracción y validación de operandos, asignación de dirección temporal de resultado, actualización de pilas, y generación del cuádruplo.

Ambos métodos incorporan llamadas a `debug_stacks` para facilitar la depuración del estado de las pilas en cada etapa. Esta estructura modular y validada garantiza la correcta construcción de expresiones aritméticas complejas en el código fuente del programa.

```
def add_expresion(self):
    # Añadir una expresión (comparación)
    if self.stack_operators:
        op = self.stack_operators[-1]
        if op == '>' or op == '<' or op == '!=':
            right_operand = self.stack_operands.pop()
            left_operand = self.stack_operands.pop()
            right_operand_tipo = self.stack_types.pop()
            left_operand_tipo = self.stack_types.pop()
            operator = self.stack_operators.pop()
            res_tipo = self.semantic_cube(operator, left_operand_tipo, right_operand_tipo)
            if res_tipo == 'bool':
                self.var_temp_bool += 1
                res_address = self.var_temp_bool
                self.stack_operands.append(res_address)
                self.stack_types.append(res_tipo)
                self.stack_quadruples.append([operator, left_operand, right_operand, res_address])
                self.debug_stacks("if bool add_expresion")
            else:
                raise ValueError(f"Invalid operation: {left_operand_tipo} {operator} {right_operand_tipo}")
```

El método `add_expresion` se encarga de procesar comparaciones lógicas dentro del análisis semántico de un compilador. Su propósito principal es detectar si el operador en la cima de la pila corresponde a una operación relacional (`>`, `<` o `!=`) y, en ese caso, proceder con la validación semántica y la generación del cuádruplo correspondiente.

El método extrae los operandos izquierdo y derecho junto con sus tipos de dato desde las pilas respectivas. Luego, elimina el operador relacional de la pila de operadores y utiliza el cubo semántico (`semantic_cube`) para verificar si la comparación entre los tipos es válida. Si el resultado de la operación es del tipo booleano (`bool`), se asigna una nueva dirección temporal para almacenar dicho resultado, se actualizan las pilas semánticas, y se genera el cuádruplo que representa la operación de comparación.

Si la comparación no es válida según el cubo semántico, se lanza una excepción indicando el tipo de error semántico. Esta lógica garantiza que las comparaciones sean verificadas de forma segura antes de su traducción a código intermedio, asegurando la integridad de las expresiones condicionales durante la compilación.

```
def add_assing(self):
    # Añadir una asignación
    if self.stack_operators:
        op = self.stack_operators[-1]
        if op == '=':
            right_operand, right_operand_tipo = self.stack_operands.pop(), self.stack_types.pop()
            left_operand, left_operand_tipo = self.stack_operands.pop(), self.stack_types.pop()
            operator = self.stack_operators.pop()
            res_tipo = self.semantic_cube(operator, left_operand_tipo, right_operand_tipo)
            self.debug_stacks("add_assing")
            if res_tipo != 'error':
                self.stack_quadruples.append([operator, right_operand, None, left_operand])
            else:
                raise ValueError(f"Invalid operation: {left_operand_tipo} {operator} {right_operand_tipo}")
```

La función `add_assing` se encarga de generar un cuádruplo correspondiente a una instrucción de asignación (=) dentro del proceso de compilación. Primero, verifica si el operador en la cima de la pila es una asignación. De ser así, extrae los operandos y sus respectivos tipos desde las pilas semánticas, y remueve el operador = de la pila. A continuación, valida la operación mediante el cubo semántico (`semantic_cube`) para asegurarse de que el tipo del valor que se asigna sea compatible con el tipo de la variable receptora. Si la validación es exitosa, se genera un cuádruplo de la forma `[=, valor, None, variable]` y se añade a la lista de cuádruplos; en caso contrario, se lanza una excepción indicando un error semántico.

```
def add_print(self, string=[]):
    # Añadir una operación de impresión
    if string == []:
        right_operand = self.stack_operands.pop()
        right_operand_tipo = self.stack_types.pop()
        self.stack_quadruples.append(['print', right_operand, None, None])
    else:
        self.stack_quadruples.append(['print', string, None, None])
```

Por su parte, la función `add_print` es responsable de generar el cuádruplo para una instrucción de impresión (`print`). Si se trata de una impresión de una variable o expresión, extrae el operando y su tipo desde las pilas y genera el cuádruplo correspondiente. Si en cambio se trata de una cadena de texto literal, utiliza directamente el valor proporcionado para generar el cuádruplo. Este diseño permite una integración flexible entre la impresión de valores computados y literales, manteniendo la consistencia del código intermedio generado.



## Reglas de Saltos

```
def add_goto_false(self):
    # Añadir una instrucción GOTOF
    print("add_goto_false ----> ", self.stack_operands, self.stack_types)
    condition = self.stack_operands.pop()
    condition_type = self.stack_types.pop()

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

    self.stack_quadruples.append(['GOTOF', condition, None, None])
    self.stack_jumps.append(len(self.stack_quadruples) - 1)
    self.debug_stacks("add_goto_false")

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

    self.stack_jumps.append(len(self.stack_quadruples) - 1)
    # Actualiza el ultimo campo, el destino del salto
    self.stack_quadruples[false_jump][-1] = len(self.stack_quadruples)
    self.debug_stacks("add_goto")

def add_goto_False_fill(self):
    # Completar una instrucción GOTO
    false_jump = self.stack_jumps.pop()
    self.stack_quadruples[false_jump][-1] = len(self.stack_quadruples)
    self.debug_stacks("add_goto_False_fill")

def cycle_start(self):
    # puntero para luego regresarse
    # Marcar el inicio del ciclo while
    self.stack_jumps.append(len(self.stack_quadruples))
    self.debug_stacks("cycle_start")

def add_goto_true_while(self):
    # Añadir una instrucción GOTOV para el ciclo while
    condition = self.stack_operands.pop()
    condition_type = self.stack_types.pop()

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

    direccion_salto = self.stack_jumps.pop()
    self.stack_quadruples.append(['GOTOV', condition, None, direccion_salto])
    self.debug_stacks("add_goto_true_while")

def add_goto_loop(self):
    # Recuperamos la "etiqueta_inicio" que guardamos en cycle_start()
    etiqueta_inicio = self.stack_jumps.pop() - 1 # esta stack_jumps era la primera que ingresó
    cycle_start()
    # Insertamos el GOTO
    self.stack_quadruples.append(['GOTO', None, None, etiqueta_inicio])
    # Ahora "etiqueta_fin" = len(cuadruplos) (la posición a parchear)
    self.debug_stacks("add_goto_loop")

def patch_gotof(self):
    self.debug_stacks("antes de: patch_gotof")

    # Recuperamos la posición del GOTOF que creamos en add_gotof_placeholder()
    pos_gotof = self.stack_jumps.pop() + 1
    # "etiqueta_fin" es la posición actual (primer cuádruplo fuera del cuerpo)
    etiqueta_fin = len(self.stack_quadruples)
    # Reemplazamos el "None" del destino con esa etiqueta
    self.stack_quadruples[pos_gotof][3] = etiqueta_fin
    self.debug_stacks("después de: patch_gotof")
```

- **add\_goto\_false()**

Esta función se activa después de evaluar una condición booleana para un `if` o `while`. Si la condición es **falsa**, se debe saltar a otra parte del programa (como evitar ejecutar un bloque). Se genera:

```
['GOTOF', condition, None, destino]
```

El `destino` aún no se conoce, así que se guarda el índice del cuádruplo incompleto en `stack_jumps`.

- **add\_goto()**

Esta función se utiliza al terminar un bloque `if` cuando hay un `else`, para asegurarse de que se salta el bloque `else` si la condición inicial fue verdadera.

```
['GOTO', None, None, destino]
```

Y parchea el cuádruplo GOTOF anterior para que sepa a dónde saltar si la condición fue falsa.

- **add\_goto\_False\_fill()**

Se utiliza al final de un `if` sin `else` para parchar el `GOTOF` que quedó pendiente y señalar a dónde debe saltar si la condición fue falsa.

- **cycle\_start()**

Marca el inicio de un ciclo `while`, guardando en `stack_jumps` la posición actual de los cuádruplos. Esto servirá para regresar al comienzo del ciclo después de cada iteración.

- **add\_goto\_true\_while()**

Este es un cuádruplo inverso al `GOTOF`, usado si tu lenguaje usa `do-while`. Genera un salto al inicio del ciclo solo si la condición es verdadera:

```
['GOTOV', condition, None, direccion_salto]
```

- **add\_goto\_loop()**

Se invoca al final del cuerpo de un **while**. Agrega un **GOTO** incondicional para regresar al inicio del ciclo, usando la dirección que se guardó con **cycle\_start()**.

- **patch\_gotof()**

Esta función parcha el **GOTOF** generado al inicio del ciclo con la dirección del primer cuádruplo después del ciclo.

## Lógica de IF-ELSE, WHILE, DO-WHILE

### Lógica de IF-ELSE

```
def p_condition(p):
    'condition : IF LPAREN expresion RPAREN goto_false body else_part'
    # Manejar la declaración if-else
    st.add_goto_False_fill()
def p_goto_false(p):
    '''goto_false : '''
    # Añadir la instrucción de salto condicional para if
    st.add_goto_false()
def p_else_part(p):
    '''else_part : ELSE goto body
                  | empty'''
    # Manejar la parte else de una declaración if-else
    pass
```

Primero necesita checar que sea un valor booleano, para posteriormente crear el cuádruplo **self.stack\_quadruples.append(['GOTOF', condition, None, None])** , ahí crea el cuádruplo y agrega su respectivo salto.

Salto : [7]

Cuádruplos :

0: ['print', 'Haciendo un IF y ELSE', None, None]

1: ['=', 501, None, 0]

2: ['=', 502, None, 1]


3: ['+', 0, 1, 200]

4: ['=', 200, None, 2]

```
5: ['=', 503, None, 3]
6: ['>', 2, 503, 401]
7: ['GOTO', 401, None, None]
```

Sin embargo en ADD\_GO\_TO, le hace pop a ese salto, para agregar en ese cuádruplo la longitud de los cuádruplos, en base a que sepa en donde se está DESPUÉS DEL GOTO. Asimismo, tome la Longitud de los cuádruplos - 1 para tener la dirección de GOTO

Salto : [9]

 Cuádruplos :

```
0: ['print', 'Haciendo un IF y ELSE', None, None]
1: ['=', 501, None, 0]
2: ['=', 502, None, 1]
3: ['+', 0, 1, 200]
4: ['=', 200, None, 2]
5: ['=', 503, None, 3]
6: ['>', 2, 503, 401]
7: ['GOTO', 401, None, 10]
8: ['print', 'C es mayor a 10', None, None]
9: ['GOTO', None, None, None]
```

Después entra a ADD\_GO\_TO\_FILL para remover los saltos, en este caso 9, y agregar al final del GOTO la longitud de los cuádruplos lo cual indica salir del GOTO.

```
0: ['print', 'Haciendo un IF y ELSE', None, None]
1: ['=', 501, None, 0]
2: ['=', 502, None, 1]
3: ['+', 0, 1, 200]
4: ['=', 200, None, 2]
5: ['=', 503, None, 3]
6: ['>', 2, 503, 401]
7: ['GOTO', 401, None, 10]
8: ['print', 'C es mayor a 10', None, None]
9: ['GOTO', None, None, 11]
10: ['print', 'C es menor a 10', None, None]
```

## Lógica de WHILE

```

def p_cycle(p):
    '''cycle : WHILE cycle_start LPAREN expresion RPAREN goto_false body generate_goto
    finish_gotoof'''
    pass

def p_goto_false(p):
    '''goto_false : '''
    # Añadir la instrucción de salto condicional para if
    st.add_goto_false()


def p_generate_goto(p):
    '''generate_goto : '''
    # 3) Emitimos un GOTO incondicional al inicio del ciclo
    st.add_goto_loop()

def p_finish_gotoof(p):
    '''finish_gotoof : '''
    # 4) Parcheamos la dirección del GOTOF para que apunte justo después del GOTO

    st.patch_gotoof()

```

## Después de la lógica “normal”, cuando inicia cycle start

 Saltos : [6]

```

0: ['print', 'Haciendo WHILE', None, None]
1: ['=', 501, None, 0]
2: ['=', 502, None, 1]
3: ['+', 0, 1, 200]
4: ['=', 200, None, 2]
5: ['=', 503, None, 3]

```

Después de la operación Booleana **6: ['>', 3, 504, 401]** , nos encontramos con **add\_goto\_false** agregas el cuádruplo, pero también guardas en saltos su respectivo lugar.

 Saltos : [6, 7]

Cuádruplos :

```

0: ['print', 'Haciendo WHILE', None, None]
1: ['=', 501, None, 0]
2: ['=', 502, None, 1]
3: ['+', 0, 1, 200]
4: ['=', 200, None, 2]
5: ['=', 503, None, 3]
6: ['>', 3, 504, 401]
7: ['GOTO', 401, None, None]

```

Luego llega a `add_goto_loop` y ahí hace `pop` al primer de nuestros saltos, en este caso es 6 , y hace 11: ['GOTO', None, None, 6]

Finalmente llegamos a Patch de GOTOFALSE y ahí con

```
pos_gotof = self.stack_jumps.pop() + 1
etiqueta_fin = len(self.stack_quadruples)
self.stack_quadruples[pos_gotof][3] = etiqueta_fin
llenamos 7: ['GOTO', 401, None, 12],
```

## Lógica de DO-WHILE

```
def p_do_cycle(p):
    '''do_cycle : DO cycle_start body WHILE LPAREN expresion RPAREN goto_true'''
    # La primera línea es "while"
    # La segunda línea es tu "do-while" original
    pass
def p_goto_true(p):
    '''goto_true : '''
    # Añadir la instrucción de salto condicional para el ciclo
    st.add_goto_true_while()
```

Inicia el cycle start y con ello se agrega

Salto : [6]

```
0: ['print', 'Haciendo un DO - WHILE', None, None]
1: ['=', 501, None, 0]
2: ['=', 502, None, 1]
3: ['+', 0, 1, 200]
4: ['=', 200, None, 2]
5: ['=', 503, None, 3]
6: ['print', 'Probando Do while', None, None]
7: ['-', 0, 504, 201]
8: ['=', 201, None, 0]
9: ['>', 0, 505, 401]
```

Cuando llega a add\_goto\_true\_while se hace pop de este salto y se agrega al cuádruplo

```
0: ['print', 'Haciendo un DO - WHILE', None, None]
1: ['=', 501, None, 0]
2: ['=', 502, None, 1]
3: ['+', 0, 1, 200]
4: ['=', 200, None, 2]
5: ['=', 503, None, 3]
6: ['print', 'Probando Do while', None, None]
7: ['-', 0, 504, 201]
8: ['=', 201, None, 0]
9: ['>', 0, 505, 401]
10: ['GOTOV', 401, None, 6]
```

# Máquina Virtual

Es una estructura que interpreta y ejecuta una lista de instrucciones llamadas **cuádruplos**, las cuales representan operaciones en un lenguaje intermedio. Cada cuádruplo es una lista con cuatro elementos:

**[operador, operando\_izquierdo, operando\_derecho, resultado]**

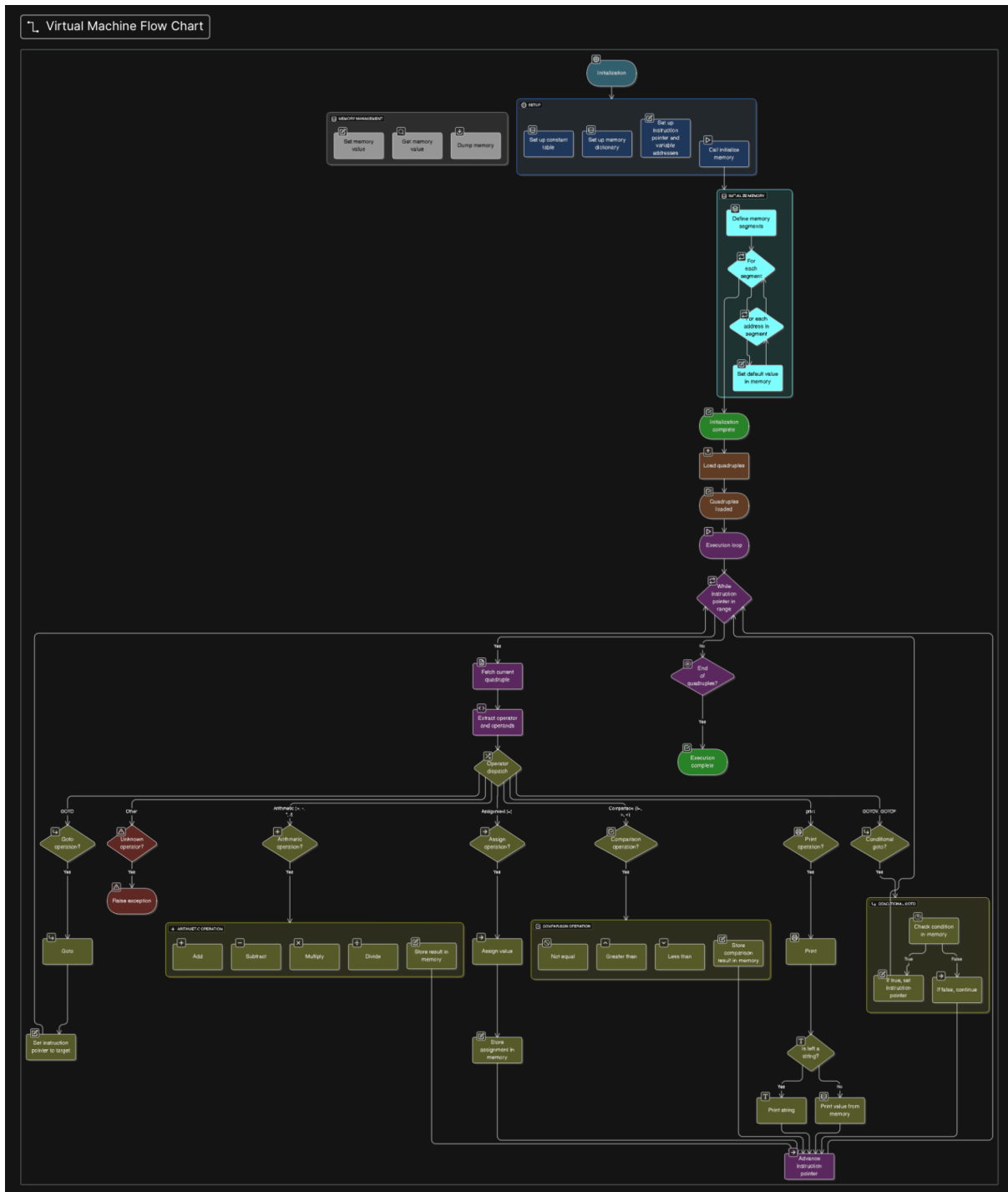
Implementé una función que recorre uno por uno los cuádruplos usando un puntero de instrucción (`instruction_pointer`) y ejecuta lo que indican:

```
if op == '+':  
    self.memory[result] = self.memory[left] + self.memory[right]
```

Por ejemplo, también copia el valor de una dirección de memoria (`left`) a otra (`result`), simulando una asignación `a = b`.

```
elif op == '=':  
    self.memory[result] = self.memory[left]
```

- **GOTO**: Salta a otra posición sin importar nada.
- **GOTOV**: Salta solo si el operando es `True`.
- **GOTOF**: Salta solo si el operando es `False`.



Está diseñada para ejecutar un conjunto de instrucciones conocidas como **cuádruplos**, los cuales representan operaciones aritméticas, lógicas, de control de flujo y de impresión, generadas previamente por el compilador. Su núcleo operativo es el método *execute*, que recorre secuencialmente la lista de cuádruplos y los interpreta uno a uno mediante una estructura condicional.



En cada iteración, se extraen los componentes del cuádruplo: el operador (`op`), los operandos (`left` y `right`) y el resultado (`result`). Dependiendo del operador, la máquina virtual realiza la operación correspondiente. Por ejemplo, en operaciones aritméticas como suma, resta, multiplicación o división, la máquina toma los valores almacenados en las direcciones de memoria indicadas por `left` y `right`, realiza la operación y almacena el resultado en la dirección `result`.

Para operaciones de asignación (`=`), la máquina simplemente copia el valor de la dirección `left` hacia la dirección `result`. En operaciones lógicas y de comparación (`!=`, `>`, `<`), se evalúan las condiciones y se almacena el resultado booleano correspondiente. Este comportamiento es fundamental para evaluar condiciones en estructuras como `if` o `while`.

En cuanto al control de flujo, la máquina virtual implementa instrucciones de salto incondicional (`GOTO`) y condicional (`GOTOV`, `GOTOF`). Estas instrucciones alteran el flujo normal de ejecución cambiando directamente el valor del puntero de instrucción (`instruction_pointer`). Por ejemplo, si la condición de un `GOTOF` es falsa, la ejecución salta a la posición especificada en `result`, permitiendo así implementar estructuras como condicionales y ciclos.

Además, la máquina virtual admite instrucciones de impresión (`print`). Estas se manejan con una lógica diferenciada: si el argumento a imprimir es una cadena literal, se muestra directamente; si es una dirección de memoria, se recupera el valor almacenado y se imprime. Esta dualidad permite imprimir tanto valores dinámicos como mensajes estáticos.

Finalmente, se proporcionan métodos auxiliares como `set_memory` y `get_memory` para manipular directamente el contenido de la memoria, y `dump_memory` para obtener un volcado ordenado de todos los valores almacenados, lo cual es útil para propósitos de depuración o análisis posterior. En conjunto, esta máquina virtual actúa como el motor de ejecución de un lenguaje intermedio, procesando de forma eficiente las instrucciones derivadas del análisis semántico del código fuente.

# ERRORES

## Primer Error:

- Mala implementación de integración de variables

```
def p_a_vars(p):
    '''a_vars : empty
        | vars'''
    # Manejar la declaración de variables
    variables_list = p[1]
    for var in variables_list:
        the_name, the_type = var.split(':')
        st.push_variable(the_name, the_type)

# → body: para manejar el cuerpo del programa principal o de funciones
def p_body(p):
    'body : LBRACE p_list_of_statements RBRACE'
    # Manejar el cuerpo de funciones o del programa principal
    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]}'
```

## Solución: Repetición de llamadas al archivo de texto

## Segundo Error:

- Print Variable Table ---->
- {}
- Print Quads ---->
- []
- []
- []
- []
- []
- []
- [SymTab] Inserté variable 'a' de tipo 'int' en dirección 0
- [SymTab] Inserté variable 'b' de tipo 'int' en dirección 1
- [SymTab] Inserté variable 'c' de tipo 'int' en dirección 2

- Error durante parseo/semántica:
- pop from empty list

```

Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while
Probando Do while

```

#### Razones:

- Desbordamiento UNDERFLOW de pilas semánticas
- Manejo incompleto del back\_patch en el esquema de do...while
  - Debido a que do\_while\_cycle : DO cycle\_start body WHILE LPAREN expression RPAREN go\_to\_true SEMICOLON

#### Genera problemas:

<b>cycle_start</b>	Coloca correctamente el índice de inicio en st.stack_jumps.
<b>body</b>	genera cuádruplos por cada sentencia interna (print, asignaciones, etc.).
<b>WHILE ( expression ) go_to_true ;</b>	evalúa la expresión y luego solo hace add_go_to_true(), lo cual genera en la tabla de cuádruplos algo como: ['go_to_t', <cond_addr>, None, <start_idx>]

- Logré solucionar dichos errores al mejorar la tabla de símbolos al agregar diferentes funciones, como: add\_goto\_true\_while, add\_goto\_loop y patch\_gotoif

## **Código En GitHub**

[https://github.com/Valenriquez/Compilers/tree/main/Python/Entrega\\_Final](https://github.com/Valenriquez/Compilers/tree/main/Python/Entrega_Final)