BABYDUCK

DOCUMENTACIÓN DEL COMPILADOR

CHRISTOPHER GABRIEL PEDRAZA POHLENZ

DESARROLLO DE APLICACIONES AVANZADAS DE CIENCIAS COMPUTACIONALES (GPO. 503)

Monterrey, Nuevo León - 2 de junio, 2025

CONTENTS

REGISTRO DE CAMBIOS

Entrega	**Cambios realizados**

#3

- Cambiar regla de gramática de **exp** para permitir expresiones que combinen sumas y restas.
  - **Antes:**

```
?exp: term
      | term (PLUS term)+ -> exp_plus
      | term (MINUS term)+ -> exp_minus
```

  - **Después:**

```
?exp: term
      | exp PLUS term -> exp_plus
      | exp MINUS term -> exp_minus
```

- Cambiar regla de gramática de **term** para permitir expresiones que combinen multiplicaciones y divisiones.
  - **Antes:**

```
?term: factor
      | factor (MULT factor)+ -> term_mult
      | factor (DIV factor)+ -> term_div
```

  - **Después:**

```
?term: factor
      | term MULT factor -> term_mult
      | term DIV factor -> term_div
```

- Modificar clase interpreter para que se genere el código intermedio (cuádruplos) ahí en vez de interpretar las expresiones.

- Cambiar reglas de expresiones para considerar correctamente la precedencia de los operadores relacionales y lógicos (AND→OR→[>, <, ≤, ≥, ≠, ==])
  - **Antes:**

```
?expresion: exp
            | exp comparison_op exp -> expresion_comparison_op
            | exp logic_cond exp -> expresion_logic_cond

?logic_cond: AND | OR
```

  - **Después:**

```
?comparison: exp
            | exp comparison_op exp -> binary_comparison


?logical_and: comparison
            | logical_and AND comparison -> binary_logical_and
```

```
?logical_or: logical_and
            | logical_or OR logical_and -> binary_logical_or

?expresion: logical_or
```

- Cambiar función transformadora de ids negativos para que se transformen en restas de cero menos el id:
  - **Antes:**

```
def negative_factor_id(self, minus, id):
    return ("negative_factor_id", id)
```

  - **Después:**

```
def negative_factor_id(self, minus, id):
    return ("exp_minus", ("cte_num", 0), ("id", id))
```

- Cambiar estructura de árbol sintáctico transformado de tuplas a utilizar clases de datos. Esta decisión se tomó para facilitar el acceso de los datos guardados en cada nodo. Anteriormente era necesario acceder a los datos por medio de índices, sin embargo, siempre era necesario acudir a una referencia para saber qué estaba guardado en cada posición, mientras que ahora, pudiendo acceder por medio de los nombres de los atributos, es más fácil identificar qué dato se está accediendo.

#4

- Anteriormente guardaba los identificadores de las variables y las constantes string en los cuádruplos, en esta entrega migre a que los cuádruplos tengan puros valores enteros (excepto los elementos vacíos que tienen un None), usando las direcciones de memoria.
- Anteriormente cuando asignaba espacios de memoria, de una vez iba guardando en los espacios el valor. Sin embargo, en esta entrega lo cambié a que solo se asignen los espacios donde irán los valores, pero no se haga ningún proceso de almacenamiento, puesto que eso se hará en el proceso de traducción.

#5

- Anteriormente, dividía la memoria en 4 espacios: Global, local, temporal y constantes. Esto no era la mejor solución ya que cada registro de activación de las funciones requería sus propios temporales, por lo que el espacio de memoria de temporal solo sería usado para aspectos globales. Para mejorar esto, se creó una clase Memory para describir un espacio de memoria con tipos de datos arbitrarios (para que sea más modular), que se pondría en los espacios de global, y constantes, al igual que como memoria local cada que se crea un nuevo registro de activación. El espacio de memoria temporal se eliminó para mejor incluir los registros temporales dentro de local y los registros de activación.
- Modificar diagramas de gramática para incluir “return” en los estatutos y llamadas de función como factores.

#6

- Agregar operador de “!=” a los cuádruplos, ya que la gramática lo permitía, pero no existía un valor numérico que lo describiera en la lista de operadores. También agregar a la máquina virtual la interpretación del operador.
- Corregir el id positivo como factor, ya que no consideraba el símbolo de +, por lo que fallaba el proceso de parsing.
- Agregar pruebas unitarias para probar cada regla de la gramática.

- Actualizar diagramas con puntos neurálgicos para reflejar cambios realizados en las últimas entregas

- Al hacer la llamada de función como factor, por error estaba creando siempre un temporal en el espacio de memoria global para guardar lo que regresaba la función. Cambié que se usen las dos variables que tenía declaradas que dinámicamente iban referenciando al espacio de memoria y contenedor en donde se encontraba el analizador, y de esta manera ya se hacen los temporales correctamente: si se hace la llamada desde main, se guardan como temporal global, de lo contrario, se guardan como temporal local. Esto resuelve un problema que se tenía previamente que no dejaba hacer recursión doble (dos llamadas recursivas en la misma expresión).

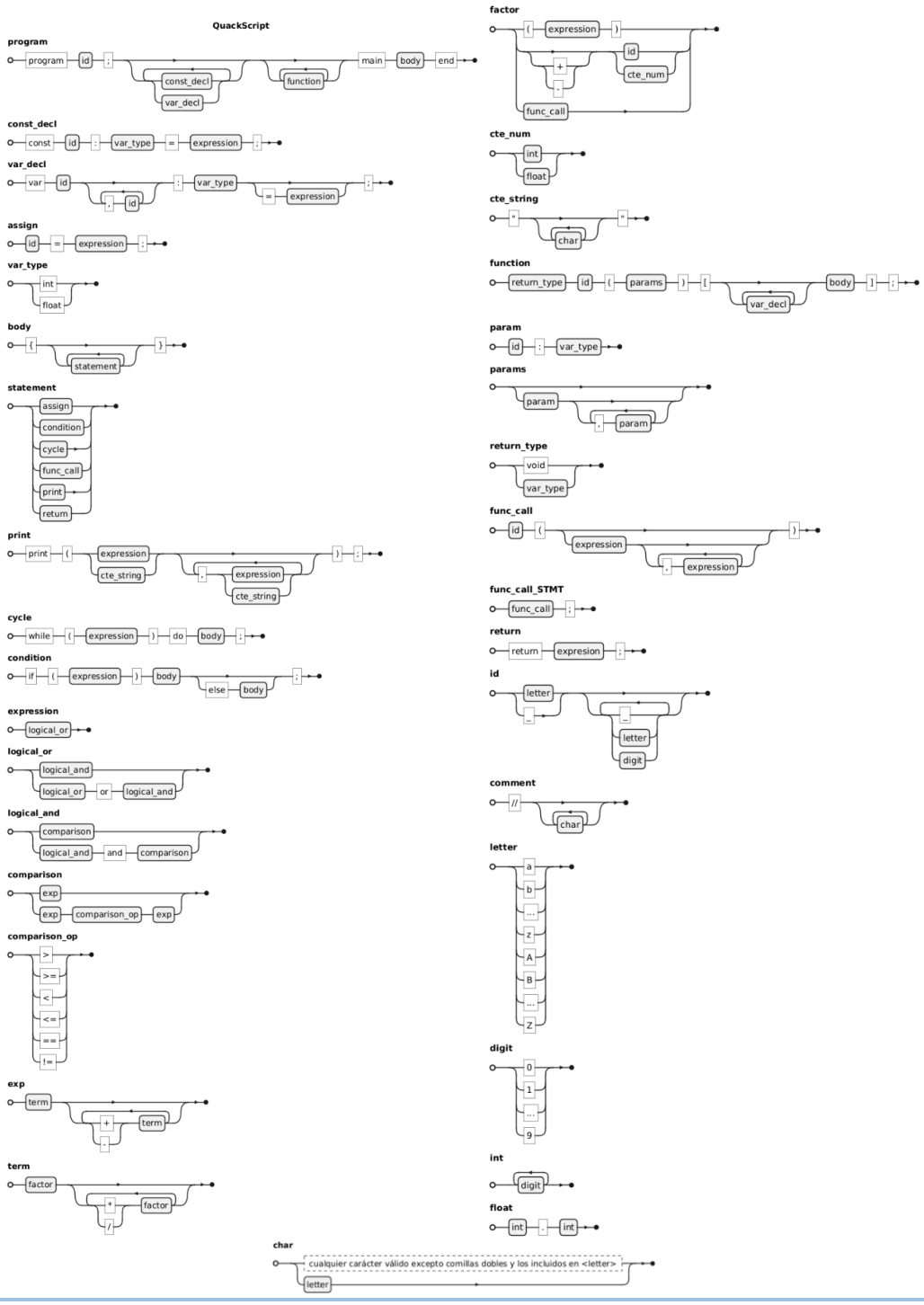
REPOSITORIO DEL COMPILADOR

https://github.com/christopher-pedraza/compilador-quack-script

CAPACIDADES DEL COMPILADOR

- Operaciones de suma, resta, multiplicación y división.
- Comparaciones con los operadores <, >, ≤, ≥, ≠, ==, and, or.
- Constantes numéricas enteras, flotantes, en notación científica.
- Constantes de cadenas de caracteres para ser usadas en impresiones de pantalla.
- Almacenamiento en variables y constantes de tipo entero y flotante.
- Asignar y actualizar valores a las variables posterior a su declaración.
- Impresiones de pantalla y uso de caracteres de escape en las impresiones.
- Ciclos while
- Condicionales if e if-else
- Funciones
    - Tanto void, como que regresen valores enteros y flotantes.
    - Permitir recursividad por medio de memorias locales específicas por llamada de función.
- Validaciones de errores de léxico, sintaxis y semántica.

DIAGRAMAS DE LA GRAMÁTICA




```
@startebnf
title QuackScript
program = "program", id, ";", {const_decl | var_decl}, {function}, "main", body, "end";

const_decl = "const", id, ":", var_type, "=", expression, ";";

var_decl = "var", id, {",", id}, ":", var_type, ["=", expression], ";";
```

```
assign = id, "=", expression, ";";

var_type = "int" | "float";

body = "{", {statement}, "}";

statement = assign | condition | cycle | func_call | print | return;

print = "print", "(", (expression | cte_string), {",", (expression | cte_string)}, ")", ";";

cycle = "while", "(", expression, ")", "do", body, ";";

condition = "if", "(", expression, ")", body, ["else", body], ";";

expression = logical_or;

logical_or = logical_and | logical_or, "or", logical_and;

logical_and = comparison | logical_and, "and", comparison;

comparison = exp | exp, comparison_op, exp;

comparison_op = ">" | ">=" | "<" | "<=" | "==" | "!=";

exp = term, {("+" | "-"), term};

term = factor, {("*" | "/"), factor};

factor = ("(", expression, ")") | (["+" | "-"], (id | cte_num)) | func_call;

cte_num = int | float;

cte_string = '"', {char}, '"';

function = return_type, id, "(", params, ")", "[", {var_decl}, body, "]", ";";

param = id, ":", var_type;

params = [param, {",", param}];

return_type = "void" | var_type;

func_call = id, "(", [expression, {",", expression}], ")";

func_call_STMT = func_call, ";";

return = "return", expresion, ";";

id = (letter|"_"), {"_" | letter | digit};

comment = "//", {char};

letter = "a" | "b" | "..." | "z" | "A" | "B" | "..." | "Z";
digit = "0" | "1" | "..." | "9";
int = {digit}-;
float = int, ".", int;
char = ? cualquier carácter válido excepto comillas dobles y los incluidos en <letter> ? | letter;

@endebnf
```

Código y diagramas en: https://n9.cl/13zqa

TOKENS Y EXPRESIONES REGULARES

PALABRAS RESERVADAS

Nombre de Token	Patrón	Regex
PROGRAM	program	program
MAIN	main	main
END	end	end
CONST	const	const
VAR	var	var
VOID	void	void
WHILE	while	while
DO	do	do
IF	if	if
ELSE	else	else
PRINT	print	print
TYPE	float, int	float|int
AND	and	and
OR	or	or
RETURN	return	return

SÍMBOLOS

Nombre de Token	Patrón	Regex
ASSIGN	=	=
COLON	:	:
SEMICOLON	;	;
COMMA	,	,
LPAREN	(	\(
RPAREN	)	\)
LBRACE	{	\{
RBRACE	}	\}
LBRACKET	[	\[
RBRACKET	]	\]
PLUS	+	\+
MINUS	-	\-
MULT	*	*
DIV	/	/
GT	>	>
GTE	>=	>=
LT	<	<

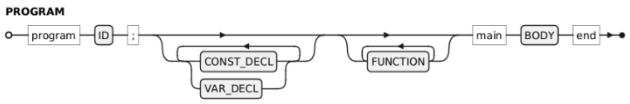
LTE	<	<=
NE	!=	!=
EE	==	==
DOT	.	\.

IDENTIFICADORES Y LITERALES

Nombre de Token	Patrón	Regex
ID	num_A	[a-zA-Z_]([a-zA-Z0-9_])*
INT	23	[0-9]+
FLOAT	15.44	[0-9]+\.[0-9]+
CTE_STRING	“Tec de Monterrey”	"[^"\n]*"
COMMENT	// Comentario	//[^\n]*

REGLAS GRAMATICALES EQUIVALENTES A LOS DIAGRAMAS

Program



PROGRAM → program <ID> ; <PROGRAM’> <PROGRAM’’> main <BODY> end

PROGRAM’ → ε

PROGRAM’ → <CONST_DECL> <PROGRAM’>

PROGRAM’ → <VAR_DECL> <PROGRAM’>

PROGRAM’’ → ε

PROGRAM’’ → <CONST_DECL> <PROGRAM’’>

CONST_DECL

CONST_DECL → const <ID> : <TYPE> = <EXPRESION> ;

VAR_DECL

VAR_DECL → var <ID> <MULT_ID> : <TYPE> <DECL_ASSIGN> ;

MULT_ID → ε

MULT_ID → , <ID> <MULT_ID>

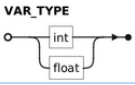
DECL_ASSIGN → ε

DECL_ASSIGN → = <EXPRESION>

ASSIGN

ASSIGN → <ID> = <EXPRESION> ;

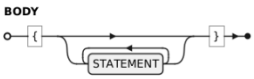
VAR_TYPE



VAR_TYPE → int

VAR_TYPE → float

BODY

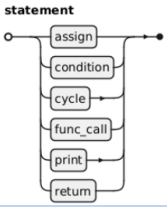


BODY → { <BODY'> }

BODY' → ε

BODY' → <STATEMENT> <BODY'>

STATEMENT



STATEMENT → <ASSIGN>

STATEMENT → <CONDITION>

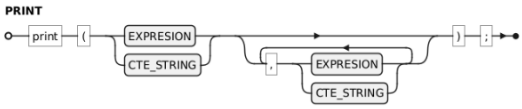
STATEMENT → <CYCLE>

STATEMENT → <FUNC_CALL>

STATEMENT → <PRINT>

STATEMENT → <RETURN>

PRINT



PRINT → print (<PRINT'>) ;

PRINT' → <EXPRESION> <PRINT''>

PRINT' → <CTE_STRING> <PRINT''>

PRINT'' → ε

PRINT'' → , <EXPRESION> <PRINT''>

PRINT'' → , <CTE_STRING> <PRINT''>

CYCLE

CYCLE → while (<EXPRESION>) do <BODY> ;

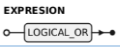
CONDITION

CONDITION → if (<EXPRESION>) <BODY> <ELSE_COND> ;

ELSE_COND → ε

ELSE_COND → else <BODY>

EXPRESION



EXPRESION → <LOGICAL_OR>

LOGICAL_OR

LOGICAL_OR → <LOGICAL_AND>

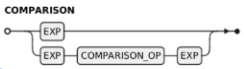
LOGICAL_OR → <LOGICAL_OR> or <LOGICAL_AND>

LOGICAL_AND

LOGICAL_AND → <COMPARISON>

LOGICAL_AND → <LOGICAL_AND> and <COMPARISON>

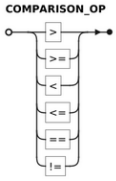
COMPARISON



COMPARISON → <EXP>

COMPARISON → <EXP> <COMPARISON_OP> <EXP>

COMPARISON_OP

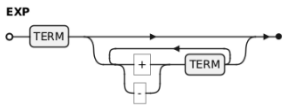


COMPARISON_OP → >

COMPARISON_OP → >=

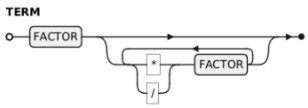
COMPARISON_OP → <
COMPARISON_OP → <=
COMPARISON_OP → ==
COMPARISON_OP → !=

EXP



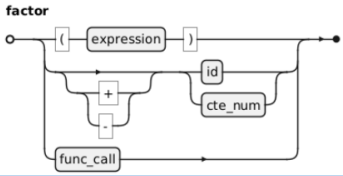
EXP → <TERM> <EXP'>
EXP' → ε
EXP' → + <TERM> <EXP'>
EXP' → - <TERM> <EXP'>

TERM



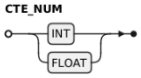
TERM → <FACTOR> <TERM'>
TERM' → ε
TERM' → * <FACTOR> <TERM'>
TERM' → / <FACTOR> <TERM'>

FACTOR



FACTOR → (<EXPRESION>)
FACTOR → <FUNC_CALLl>
FACTOR → <FACTOR'> <ID>
FACTOR → <FACTOR'> <CTE_NUM>
FACTOR' → ε
FACTOR' → +
FACTOR' → -

CTE_NUM



CTE_NUM → <INT>
CTE_NUM → <FLOAT>

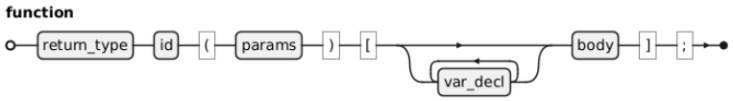
CTE_STRING

CTE_STRING → “ <CTE_STRING’> “

CTE_STRING’ → ε

CTE_STRING’ → <CHAR> <CTE_STRING’>

FUNCTION



FUNCTION → <RETURN_TYPE> <ID> (<PARAMS>) [<FUNCTION’> <BODY>] ;

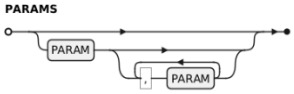
FUNCTION’ → ε

FUNCTION’ → <VAR_DECL> <FUNCTION’>

PARAM

PARAM → <ID> : <TYPE>

PARAMS



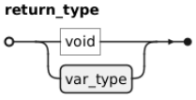
PARAMS → ε

PARAMS → PARAM <PARAMS’>

PARAMS’ → ε

PARAMS’ → , PARAM <PARAMS’>

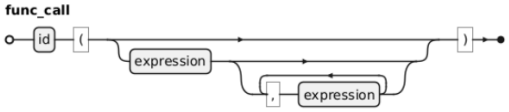
RETURN_TYPE



RETURN_TYPE → void

RETURN_TYPE → <VAR_TYPE>

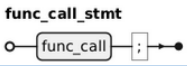
FUNC_CALL



FUNC_CALL → <ID> (<FUNC_CALL’>)

FUNC_CALL’ → ε
FUNC_CALL’ → EXPRESION <FUNC_CALL’’>
FUNC_CALL’’ → ε
FUNC_CALL’’ → , <EXPRESION>

FUNC_CALL_STMT

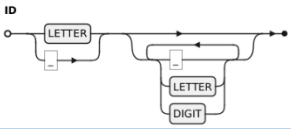


FUNC_CALL_STMT → <FUNC_CALL> ;

RETURN

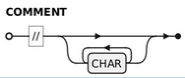
RETURN → return <EXPRESSION> ;

ID



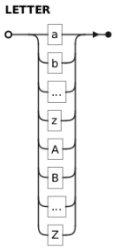
ID → LETTER <ID’>
ID → _ <ID’>
ID’ → ε
ID’ → _ <ID’>
ID’ → <LETTER> <ID’>
ID’ → <DIGIT> <ID’>

COMMENT

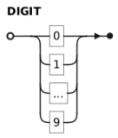


COMMENT → // <COMMENT’>
COMMENT’ → ε
COMMENT’ → <CHAR> <COMMENT’>

LETTER



LETTER → a | b | ... | z | A | B | ... | Z

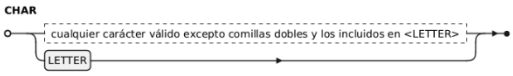
DIGIT

DIGIT → 0 | 1 | ... | 9

INT

INT → DIGIT <INT'> INT' → ε INT' → DIGIT <INT'>

FLOAT

FLOAT → <INT> . <INT>

CHAR

CHAR → *cualquier carácter válido excepto comillas dobles y los incluidos en <LETTER>* CHAR → <LETTER>

HERRAMIENTAS PARA EL DESARROLLO DE ANALIZADORES LÉXICOS Y SINTÁCTICOS

Para facilitar el desarrollo de parsers y lexers, existen múltiples herramientas consolidadas. A continuación, se describen brevemente algunas de ellas:

Herramienta	**Descripción**
Flex & Bison	Flex es un generador de analizadores léxicos y Bison es un generador de analizadores sintácticos, ambos diseñados principalmente para su uso con el lenguaje C. Flex describe las reglas léxicas utilizando expresiones regulares, mientras que Bison permite definir gramáticas en un formato similar a BNF. Bison genera parsers de tipo bottom-up, utilizando el algoritmo LALR(1). Las reglas en Bison se escriben especificando producciones y acciones en C que se ejecutan cuando las reglas se reducen.
PLY (Python Lex-Yacc)	Implementación de Lex y Yacc escrita en Python puro. Utiliza decoradores de funciones para definir las reglas léxicas y sintácticas.

	El análisis léxico se realiza mediante expresiones regulares, mientras que las reglas sintácticas son especificadas en forma de funciones Python cuyo nombre y docstring definen la producción de la gramática. PLY genera parsers bottom-up basados en LALR(1). Tiene un enfoque cercano a la forma tradicional de Lex/Yacc, pero adaptado a las capacidades de Python.
SLY (Sly Lex-Yacc)	Sucesor de PLY, también desarrollado en Python. Mantiene el modelo de definir reglas usando funciones decoradas, pero introduce mejoras como el soporte para atributos más explícitos, mayor modularidad en la construcción de los parsers y un sistema interno de depuración más completo. El motor de parsing de SLY sigue siendo bottom-up utilizando LALR(1). Las gramáticas se declaran mediante clases con métodos nombrados siguiendo convenciones específicas para asociarlos a las reglas, lo cual mejora la estructura y claridad de los proyectos grandes.
Lark	Herramienta de parsing moderna escrita íntegramente en Python. A diferencia de las anteriores, permite definir la gramática en archivos de texto o cadenas separadas del código de procesamiento, usando una sintaxis basada en EBNF (Extended Backus-Naur Form). Lark soporta varios algoritmos de parsing: Earley (top-down, capaz de procesar cualquier gramática de contexto libre, incluyendo gramáticas ambiguas) y LALR(1) (bottom-up, optimizado para rendimiento en gramáticas no ambiguas). El usuario puede elegir entre ambos modos de parsing según los requisitos de su gramática y el rendimiento esperado. Además, Lark genera automáticamente árboles sintácticos a partir del análisis, y proporciona facilidades integradas para visualizarlos, recorrerlos y manipularlos. También ofrece características adicionales como resolución automática de conflictos, seguimiento detallado de posiciones (línea y columna), herramientas de depuración, y opciones para transformar o simplificar árboles de manera programática.

ELECCIÓN DE LA HERRAMIENTA DE ANÁLISIS

Dado que el compilador se desarrollará en Python, se optó por utilizar una herramienta implementada en este mismo lenguaje para maximizar la integración, legibilidad y mantenimiento del código. Entre las opciones evaluadas, se consideraron SLY y Lark, ambas escritas en Python y con soporte para análisis léxico y sintáctico completo. Sin embargo, finalmente se eligió Lark por las siguientes razones:

- **Principios de diseño:** Lark sigue seis principios fundamentales que resonaron con las necesidades del proyecto y mías:
  1. *"La legibilidad importa"*
  2. *"Mantener la gramática limpia y simple"*
  3. *"No forzar al usuario a tomar decisiones que el parser pueda deducir automáticamente"*
  4. *"La usabilidad es más importante que el rendimiento"*
  5. *"El rendimiento sigue siendo muy importante"*
  6. *"Seguir el Zen de Python siempre que sea posible"*

- **Separación entre gramática y código:**
  Lark permite definir la gramática de forma separada del código que la procesa, facilitando la organización y el mantenimiento del proyecto.

- **Simplicidad y rapidez en la definición de gramáticas:**
  La inspiración en la sintaxis EBNF facilita declarar reglas de manera clara y entendible desde las primeras etapas de desarrollo. Posteriormente, estas reglas fueron reescritas en una forma más cercana a BNF para facilitar su procesamiento, basándose en la estructura inicial.

- **Generación automática de árboles de análisis:**
  Lark construye árboles sintácticos de forma automática, que pueden ser visualizados para verificar el análisis, o recorridos mediante funciones específicas de la herramienta. Esto permite dividir el proceso de parsing en etapas más controladas y depurables.

- **Soporte de múltiples algoritmos de análisis:**
  Lark soporta tanto Earley como LALR(1). El algoritmo Earley resulta muy versátil, ya que permite analizar cualquier gramática expresada en EBNF, ideal para etapas iniciales. No obstante, para este proyecto se eligió utilizar LALR(1) debido a su mejor rendimiento.

- **Características adicionales:**
  - Resolución automática de colisiones de terminales, cuando es posible.
  - Seguimiento preciso de líneas y columnas en el código fuente.
  - Herramientas de depuración integradas que hacen el comportamiento del parser más transparente.

- **Documentación:**
  Aunque otras herramientas más consolidadas podrían contar con una comunidad más grande o con más ejemplos disponibles, Lark ofrece una documentación clara, detallada y bien estructurada. Incluye guías prácticas para la creación de gramáticas, ejemplos guiados y ejemplos más abiertos que permiten aprender de casos concretos.

REGLAS DE CONSTRUCCIÓN

Como se mencionó anteriormente, las reglas de construcción de la gramática han sido elaboradas siguiendo el formato de EBNF, aunque con algunas modificaciones para asemejarse más a BNF. Esta adaptación resulta útil para facilitar el procesamiento posterior de las reglas.

```
?start: program

id: CNAME

cte_num: INT -> int
       | FLOAT -> float

cte_string: ESCAPED_STRING

factor: id -> factor_id
      | PLUS id -> positive_factor_id
      | MINUS id -> negative_factor_id
      | cte_num -> factor_cte_num
```

```
      | PLUS cte_num -> positive_cte_num
      | MINUS cte_num -> negative_cte_num
      | LPAREN expresion RPAREN -> parenthesis_expresion
      | func_call -> factor_func_call

?term: factor
     | term MULT factor -> term_mult
     | term DIV factor -> term_div

?exp: term
    | exp PLUS term -> exp_plus
    | exp MINUS term -> exp_minus

?comparison: exp
           | exp comparison_op exp -> binary_comparison

?logical_and: comparison
            | logical_and AND comparison -> binary_logical_and

?logical_or: logical_and
           | logical_or OR logical_and -> binary_logical_or

?expresion: logical_or

comparison_op: GT | LT | NE | EE | GTE | LTE

?var_type: "float" -> float_type
         | "int" -> int_type

assign: id ASSIGN expresion SEMICOLON

print: PRINT LPAREN (expresion | cte_string) RPAREN SEMICOLON -> print_single
     | PRINT LPAREN (expresion | cte_string) (COMMA (expresion | cte_string))+ RPAREN SEMICOLON ->
print_multiple

cycle: WHILE LPAREN expresion RPAREN DO body SEMICOLON

condition: IF LPAREN expresion RPAREN body SEMICOLON -> condition_if
         | IF LPAREN expresion RPAREN body ELSE body SEMICOLON -> condition_if_else

const_decl: CONST id COLON var_type ASSIGN expresion SEMICOLON

var_decl: VAR id COLON var_type SEMICOLON -> var_single_decl_no_assign
        | VAR id COLON var_type ASSIGN expresion SEMICOLON -> var_single_decl_assign
        | VAR id (COMMA id)+ COLON var_type SEMICOLON -> var_multi_decl_no_assign
        | VAR id (COMMA id)+ COLON var_type ASSIGN expresion SEMICOLON -> var_multi_decl_assign

?body: LBRACE RBRACE -> empty_body
     | LBRACE statement+ RBRACE -> body_statements

?params: id COLON var_type -> param
       | id COLON var_type (COMMA id COLON var_type)+ -> params_list

?return_type: "void" -> void
            | var_type -> return_type

function: return_type id LPAREN RPAREN LBRACKET body RBRACKET SEMICOLON ->
function_no_params_no_var_decl
        | return_type id LPAREN params RPAREN LBRACKET body RBRACKET SEMICOLON -> function_no_var_decl
        | return_type id LPAREN params RPAREN LBRACKET var_decl+ body RBRACKET SEMICOLON ->
function_params_var_decl
        | return_type id LPAREN RPAREN LBRACKET var_decl+ body RBRACKET SEMICOLON -> function_no_params

func_call: id LPAREN RPAREN -> func_call_no_params
         | id LPAREN expresion RPAREN -> func_call_single_param
         | id LPAREN expresion (COMMA expresion)+ RPAREN -> func_call_multiple_params

func_call_stmt: func_call SEMICOLON
```

```
return: RETURN expresion SEMICOLON -> return_expresion

?statement: assign
          | condition
          | cycle
          | func_call_stmt
          | print
          | return

program: program_pt1 program_pt2 MAIN body END -> program_no_decl
       | program_pt1 program_pt2 (const_decl | var_decl)+ MAIN body END -> program_decl_no_func
       | program_pt1 program_pt2 function+ MAIN body END -> program_func_no_decl
       | program_pt1 program_pt2 (const_decl | var_decl)+ function+ MAIN body END -> program_decl_func

program_pt1: PROGRAM
program_pt2: id SEMICOLON

// Tokens
// PALABRAS RESERVADAS
PROGRAM: "program"
MAIN: "main"
END: "end"
CONST: "const"
VAR: "var"
VOID: "void"
WHILE: "while"
DO: "do"
IF: "if"
ELSE: "else"
PRINT: "print"
AND: "and"
OR: "or"
RETURN: "return"

// SIMBOLOS
ASSIGN: "="
COLON: ":"
SEMICOLON: ";"
COMMA: ","
LPAREN: "("
RPAREN: ")"
LBRACE: "{"
RBRACE: "}"
LBRACKET: "["
RBRACKET: "]"
PLUS: "+"
MINUS: "-"
MULT: "*"
DIV: "/"
GT: ">"
LT: "<"
GTE: ">="
LTE: "<="
NE: "!="
EE: "=="

%import common.WS
%import common.CNAME
%import common.INT
%import common.FLOAT
%import common.ESCAPED_STRING
%import common.CPP_COMMENT

%ignore WS
%ignore CPP_COMMENT
```

PRECEDENCIA Y ASOCIATIVIDAD

A continuación, se presenta la tabla que detalla la precedencia y asociatividad de los operadores definidos en QuackScript. Es importante destacar que tanto la precedencia como la asociatividad de los operadores no están definidas explícitamente mediante niveles numéricos ni se calculan a través del uso de pilas durante el procesamiento de las expresiones. En lugar de eso, estas propiedades están completamente determinadas por la estructura de las reglas gramaticales.

En términos generales, la precedencia de un operador se establece según el nivel de anidamiento de las reglas en la gramática: a mayor profundidad en la definición de las reglas, mayor precedencia tendrá el operador correspondiente. Esto significa que aquellos operadores definidos en las reglas más internas (más específicas) se evaluarán primero respecto a los que aparecen en reglas más externas.

Por otro lado, la asociatividad se define a través del uso de recursión izquierda en las producciones gramaticales. Cuando una regla hace referencia recursiva a sí misma del lado izquierdo, esto indica que el operador tiene una asociatividad izquierda, lo cual implica que, en expresiones con operadores de igual precedencia, la evaluación procederá de izquierda a derecha.

Este enfoque evita la necesidad de implementar mecanismos adicionales—como el uso de pilas o tablas de precedencia—para manejar el orden de evaluación de los operadores. Esto simplifica considerablemente la lógica del parser, ya que el orden de ejecución queda determinado directamente por cómo se estructuran las reglas sintácticas.

Regla	Operadores	Precedencia (1=Alta)	Asociatividad
factor	Constantes numéricas, ids, paréntesis, etc.	1	N/A
term	*, /	2	Izquierda
exp	+, -	3	Izquierda
comparison	>, <, ≥, ≤, ≠, ==	4	Izquierda
logical_and	and	5	Izquierda
logical_or	or	6	Izquierda

TRANSFORMAR EL ÁRBOL SINTÁCTICO A UNA ESTRUCTURA MÁS MANEJABLE

Una vez que Lark procesa la gramática, genera un árbol sintáctico que puede estar compuesto por varias estructuras, entre las cuales destacan *Tree* y *Token*. *Tree* se refiere a los nodos padre, es decir, aquellos que tienen al menos un hijo, mientras que *Token* representa las hojas del árbol. Además de estas, Lark puede generar otros tipos de nodos que pueden ser relevantes dependiendo de la gramática utilizada, como nodos intermedios que agrupan expresiones o estructuras específicas.

Aunque es posible interpretar este árbol sintáctico y validar su semántica directamente a partir de su estructura, esta opción no siempre resulta la más conveniente. Por esta razón, se optó por procesar el árbol sintáctico mediante una clase llamada *Transformer*. Esta clase permite asociar las reglas de la gramática con funciones específicas que las transforman. De este modo, cuando la herramienta reconoce una regla, invoca la función

transformadora correspondiente, y el resultado de esta función se convierte en la nueva representación del *Tree* o *Token*.

En QuackScript, decidí transformar las estructuras en clases específicas para cada tipo de nodo, lo que mejora considerablemente la claridad y el manejo de los datos. Por ejemplo, en lugar de usar tuplas con un identificador posicional, implemento clases como MultiplicativeOpNode, definidas con @dataclass, que encapsulan de forma clara los elementos relevantes de cada operación. Estas clases incluyen atributos explícitos que permiten un acceso directo y semánticamente significativo a los componentes del nodo, sin depender de índices posicionales.

Esto no solo facilita el acceso a los datos, sino que también me permite definir de manera más precisa los tipos de los valores que se almacenan en cada campo. Por ejemplo, el método term_mult devuelve ahora una instancia de MultiplicativeOpNode, donde el atributo op indica si se trata de una multiplicación o división, mientras que left y right representan los operandos involucrados. Esta aproximación mejora tanto la legibilidad del código como su seguridad de tipos, facilitando además su mantenimiento y procesamiento posterior.

Primero, estas son las clases de datos que se utilizan para reconstruir el árbol sintáctico:

```
from dataclasses import dataclass
from typing import Union, List, Optional, Literal

@dataclass
class CteNumNode:
    value: Union[int, float]

@dataclass
class IdNode:
    name: str

@dataclass
class CteStringNode:
    value: str

@dataclass
class UnaryOpNode:
    op: Literal["+", "-"]
    expr: Union[CteNumNode, IdNode]

@dataclass
class ExpMinusNode:
    left: CteNumNode
    right: IdNode

@dataclass
class MultiplicativeOpNode:
    op: Literal["*", "/"]
    left: "ExprNode"
    right: "ExprNode"

@dataclass
class ArithmeticOpNode:
    op: Literal["+", "-"]
    left: "ExprNode"
    right: "ExprNode"

@dataclass
class ComparisonNode:
    op: Literal[">", "<", "==", "!=", ">=", "<="]
    left: "ExprNode"
    right: "ExprNode"

@dataclass
class LogicalAndNode:
```

```
    op: Literal["and"]
    left: "ExprNode"
    right: "ExprNode"

@dataclass
class LogicalOrNode:
    op: Literal["or"]
    left: "ExprNode"
    right: "ExprNode"

@dataclass
class AssignNode:
    var_name: str
    expr: "ExprNode"

@dataclass
class BodyNode:
    statements: List["StmtNode"]

@dataclass
class PrintNode:
    values: List[Union["ExprNode", CteStringNode]]

@dataclass
class WhileNode:
    condition: "ExprNode"
    body: BodyNode

@dataclass
class IfNode:
    condition: "ExprNode"
    then_body: BodyNode

@dataclass
class IfElseNode:
    condition: "ExprNode"
    then_body: BodyNode
    else_body: BodyNode

TypeNode = Literal["int", "float"]

@dataclass
class VarDeclNode:
    names: List[str]
    var_type: TypeNode
    init_value: Optional["ExprNode"] = None
    isConstant: bool = False

@dataclass
class ParamNode:
    name: str
    param_type: TypeNode

@dataclass
class ParamsNode:
    params: List[ParamNode]

@dataclass
class FunctionDeclNode:
    name: str
    return_type: Union[TypeNode, Literal["void"]]
    params: ParamsNode
    body: BodyNode
    var_decls: List[VarDeclNode]

@dataclass
class ReturnNode:
    expresion: "ExprNode"
```

```
@dataclass
class FuncCallNode:
    name: str
    args: List["ExprNode"]

@dataclass
class ProgramNode:
    name: str
    global_decls: List[VarDeclNode]
    functions: List[FunctionDeclNode]
    main_body: BodyNode

# Para poder referenciar ExprNode y StmtNode antes de que estén completamente definidos

ExprNode = Union[
    CteNumNode,
    IdNode,
    CteStringNode,
    UnaryOpNode,
    MultiplicativeOpNode,
    ArithmeticOpNode,
    ComparisonNode,
    LogicalAndNode,
    LogicalOrNode,
    FuncCallNode,
]

StmtNode = Union[
    AssignNode,
    PrintNode,
    WhileNode,
    IfNode,
    IfElseNode,
    VarDeclNode,
    FunctionDeclNode,
]
```

A continuación, se presenta la clase *Transformer* que se desarrolló para manejar todas las reglas de la gramática:

```
from lark import Transformer, v_args

from SymbolTable import SymbolTable
from TransformerClasses import (
    ArithmeticOpNode,
    AssignNode,
    BodyNode,
    ComparisonNode,
    CteNumNode,
    CteStringNode,
    FuncCallNode,
    FunctionDeclNode,
    IdNode,
    IfElseNode,
    IfNode,
    LogicalAndNode,
    LogicalOrNode,
    MultiplicativeOpNode,
    ParamNode,
    ParamsNode,
    PrintNode,
    ProgramNode,
    ReturnNode,
    VarDeclNode,
    WhileNode,
```

```
)


@v_args(inline=True)
class QuackTransformer(Transformer):
    def __init__(self):
        self.symbol_table = None

    """
    id: CNAME
    """

    def id(self, value):
        return IdNode(name=str(value))

    """
    cte_num: INT -> int
           | FLOAT -> float
    """

    def int(self, value):
        return int(value)

    def float(self, value):
        return float(value)

    """
    cte_string: ESCAPED_STRING
    """

    def cte_string(self, value):
        return CteStringNode(value=str(value)[1:-1])

    """
    factor: id -> factor_id
          | PLUS id -> positive_factor_id
          | MINUS id -> negative_factor_id
          | cte_num -> factor_cte_num
          | PLUS cte_num -> positive_cte_num
          | MINUS cte_num -> negative_cte_num
          | LPAREN expresion RPAREN -> parenthesis_expresion
          | func_call -> factor_func_call
    """

    def factor_id(self, id):
        return id

    def positive_factor_id(self, plus, id):
        return id

    def negative_factor_id(self, minus, id):
        return ArithmeticOpNode(op="-", left=CteNumNode(0), right=id)

    def factor_cte_num(self, cte_num):
        return CteNumNode(value=cte_num)

    def positive_cte_num(self, plus, cte_num):
        return CteNumNode(value=cte_num)

    def negative_cte_num(self, minus, cte_num):
        return ArithmeticOpNode(op="-", left=CteNumNode(0), right=CteNumNode(cte_num))

    def parenthesis_expresion(self, lpar, expresion, rpar):
        return expresion

    def factor_func_call(self, func_call):
        return FuncCallNode(name=func_call.name, args=func_call.args)
```

```
    """
    ?term: factor
        | term MULT factor -> term_mult
        | term DIV factor -> term_div
    """

    def term_mult(self, term, mult, factor):
        return MultiplicativeOpNode(op="*", left=term, right=factor)

    def term_div(self, term, div, factor):
        return MultiplicativeOpNode(op="/", left=term, right=factor)

    """
    ?exp: term
        | exp PLUS term -> exp_plus
        | exp MINUS term -> exp_minus
    """

    def exp_plus(self, exp, plus, term):
        return ArithmeticOpNode(op="+", left=exp, right=term)

    def exp_minus(self, exp, minus, term):
        return ArithmeticOpNode(op="-", left=exp, right=term)

    """
    ?comparison: exp
               | exp comparison_op exp -> binary_comparison
    """

    def binary_comparison(self, exp1, comparison_op, exp2):
        return ComparisonNode(op=comparison_op, left=exp1, right=exp2)

    """
    ?logical_and: comparison
                | logical_and AND comparison -> binary_logical_and
    """

    def binary_logical_and(self, logical_and, and_, comparison):
        return LogicalAndNode(op="and", left=logical_and, right=comparison)

    """
    ?logical_or: logical_and
               | logical_or OR logical_and -> binary_logical_or
    """

    def binary_logical_or(self, logical_or, or_, logical_and):
        return LogicalOrNode(op="or", left=logical_or, right=logical_and)

    """
    comparison_op: GT | LT | NE | EE | GTE | LTE
    """

    def comparison_op(self, value):
        return str(value)

    """
    ?var_type: "float" -> float_type
             | "int" -> int_type
    """

    def int_type(self):
        return "int"

    def float_type(self):
        return "float"

    """
    assign: id ASSIGN expresion SEMICOLON
```

```
    """

    def assign(self, id, assign, expresion, semicolon):
        return AssignNode(var_name=id.name, expr=expresion)

    """
    body: LBRACE RBRACE -> empty_body
        | LBRACE statement+ RBRACE -> body_statements
    """

    def empty_body(self, lbrace, rbrace):
        return BodyNode(statements=[])

    def body_statements(self, lbrace, *statements):
        return BodyNode(statements=list(statements[:-1]))

    """
    print: PRINT LPAREN (expresion | cte_string) RPAREN SEMICOLON -> print_single
         | PRINT LPAREN (expresion | cte_string) (COMMA (expresion | cte_string))+ RPAREN SEMICOLON ->
print_multiple
    """

    def print_single(self, print_, lpar, content, rpar, semicolon):
        return PrintNode(values=[content])

    def print_multiple(self, print_, lpar, content, *args):
        cont = [content]  # Start with the first content
        for i in range(1, len(args) - 2, 2):
            cont.append(args[i])
        return PrintNode(values=cont)

    """
    cycle: WHILE LPAREN expresion RPAREN DO body SEMICOLON
    """

    def cycle(self, while_, lpar, expresion, rpar, do, body, semicolon):
        return WhileNode(condition=expresion, body=body)

    """
    condition: IF LPAREN expresion RPAREN body SEMICOLON -> condition_if
             | IF LPAREN expresion RPAREN body ELSE body SEMICOLON -> condition_if_else
    """

    def condition_if(self, if_, lpar, expresion, rpar, body, semicolon):
        return IfNode(condition=expresion, then_body=body)

    def condition_if_else(self, if_, lpar, expresion, rpar, body1, else_, body2, semicolon):
        return IfElseNode(condition=expresion, then_body=body1, else_body=body2)

    """
    const_decl: CONST id COLON var_type ASSIGN expresion SEMICOLON
    """

    def const_decl(self, const, id, colon, var_type, assign, expresion, semicolon):
        return VarDeclNode(names=[id], var_type=var_type, init_value=expresion, isConstant=True)

    """
    var_decl: VAR id COLON var_type SEMICOLON -> var_single_decl_no_assign
            | VAR id COLON var_type ASSIGN expresion SEMICOLON -> var_single_decl_assign
            | VAR id (COMMA id)+ COLON var_type SEMICOLON -> var_multi_decl_no_assign
            | VAR id (COMMA id)+ COLON var_type ASSIGN expresion SEMICOLON -> var_multi_decl_assign
    """

    def var_single_decl_no_assign(self, var, id, colon, var_type, semicolon):
        return VarDeclNode(names=[id], var_type=var_type, isConstant=False)

    def var_single_decl_assign(self, var, id, colon, var_type, assign, expresion, semicolon):
        return VarDeclNode(names=[id], var_type=var_type, init_value=expresion, isConstant=False)
```

```
    def var_multi_decl_no_assign(self, var, id, *args):
        ids = [id]
        for i in range(1, len(args) - 3, 2):
            ids.append(args[i])
        return VarDeclNode(names=ids, var_type=args[-2], isConstant=False)

    def var_multi_decl_assign(self, var, id, *args):
        ids = [id]
        for i in range(1, len(args) - 5, 2):
            ids.append(args[i])
        return VarDeclNode(names=ids, var_type=args[-4], init_value=args[-2], isConstant=False)

    """
    ?params: id COLON var_type -> param
           | id COLON var_type (COMMA id COLON var_type)+ -> params_list
    """

    def param(self, id, colon, type_):
        return ParamsNode(params=[ParamNode(name=id, param_type=type_)])

    def params_list(self, id, colon, type_, comma, *args):
        params = [ParamNode(name=id, param_type=type_)]
        for i in range(0, len(args) - 1, 4):
            params.append(ParamNode(name=args[i], param_type=args[i + 2]))
        return ParamsNode(params=params)

    """
    ?return_type: "void" -> void
                | var_type -> return_type
    """

    def void(self):
        return "void"

    def return_type(self, var_type):
        return var_type

    """
    function: return_type id LPAREN RPAREN LBRACKET body RBRACKET SEMICOLON ->
function_no_params_no_var_decl
            | return_type id LPAREN params RPAREN LBRACKET body RBRACKET SEMICOLON ->
function_no_var_decl
            | return_type id LPAREN params RPAREN LBRACKET var_decl+ body RBRACKET SEMICOLON ->
function_params_var_decl
            | return_type id LPAREN RPAREN LBRACKET var_decl+ body RBRACKET SEMICOLON ->
function_no_params
    """

    def function_no_params_no_var_decl(self, return_type, id, lpar, rpar, lbracket, body, rbracket,
semicolon):
        return FunctionDeclNode(name=id, return_type=return_type, params=ParamsNode(params=[]),
body=body, var_decls=[])

    def function_no_var_decl(self, return_type, id, lpar, params, rpar, lbracket, body, rbracket,
semicolon):
        return FunctionDeclNode(name=id, return_type=return_type, params=params, body=body,
var_decls=[])

    def function_params_var_decl(self, return_type, id, lpar, params, rpar, lbracket, *args):
        body = args[-3]
        var_decls = list(args[:-3])
        return FunctionDeclNode(name=id, return_type=return_type, params=params, body=body,
var_decls=var_decls)

    def function_no_params(self, return_type, id, lpar, rpar, lbracket, *args):
        body = args[-3]
        var_decls = list(args[:-3])
```

```
        return FunctionDeclNode(
            name=id, return_type=return_type, params=ParamsNode(params=[]), body=body,
var_decls=var_decls
        )

    """
    return: RETURN expresion SEMICOLON -> return_expresion
    """

    def return_expresion(self, return_, expresion, semicolon):
        return ReturnNode(expresion=expresion)

    """
    func_call: id LPAREN RPAREN -> func_call_no_params
             | id LPAREN expresion RPAREN -> func_call_single_param
             | id LPAREN expresion (COMMA expresion)+ RPAREN -> func_call_multiple_params
    """

    def func_call_no_params(self, id, lpar, rpar):
        return FuncCallNode(name=id, args=[])

    def func_call_single_param(self, id, lpar, expresion, rpar):
        return FuncCallNode(name=id, args=[expresion])

    def func_call_multiple_params(self, id, lpar, expresion, *args):
        arguments = [expresion]
        for i in range(1, len(args) - 1, 2):
            arguments.append(args[i])
        return FuncCallNode(name=id, args=arguments)

    """
    func_call_stmt: func_call SEMICOLON
    """

    def func_call_stmt(self, func_call, semicolon):
        return func_call

    """
    program: program_pt1 program_pt2 MAIN body END -> program_no_decl
           | program_pt1 program_pt2 (const_decl | var_decl)+ MAIN body END -> program_decl_no_func
           | program_pt1 program_pt2 function+ MAIN body END -> program_func_no_decl
           | program_pt1 program_pt2 (const_decl | var_decl)+ function+ MAIN body END ->
program_decl_func
    """

    def program_no_decl(self, program_pt1, program_pt2, main, body, end):
        return ProgramNode(name=program_pt2, global_decls=[], functions=[], main_body=body)

    def program_decl_no_func(self, program_pt1, program_pt2, *args):
        body = args[-2]
        decls = []
        for i in range(0, len(args) - 3):
            decls.append(args[i])
        return ProgramNode(name=program_pt2, global_decls=decls, functions=[], main_body=body)

    def program_func_no_decl(self, program_pt1, program_pt2, *args):
        body = args[-2]
        funcs = []
        for i in range(0, len(args) - 3):
            funcs.append(args[i])
        return ProgramNode(name=program_pt2, global_decls=[], functions=funcs, main_body=body)

    def program_decl_func(self, program_pt1, program_pt2, *args):
        body = args[-2]
        decls = []
        funcs = []
        for i in range(0, len(args) - 3):
            item = args[i]
```

```
            if isinstance(item, VarDeclNode):
                decls.append(item)
            elif isinstance(item, FunctionDeclNode):
                funcs.append(item)
        return ProgramNode(name=program_pt2, global_decls=decls, functions=funcs, main_body=body)

    """
    ?program_pt1: PROGRAM
    ?program_pt2: id SEMICOLON
    """

    def program_pt1(self, program):
        self.symbol_table = SymbolTable()
        return program

    def program_pt2(self, id, semicolon):
        self.symbol_table.create_global_container(id.name)
        return id.name
```

A continuación, incluyo un ejemplo de la transformación del árbol sintáctico de una asignación de variable con identificador **i** con el valor **i+1**:

```
AssignNode(var_name='i',
           expr=ArithmeticOpNode(op='+', left=IdNode(name='i'), right=CteNumNode(value=1)))
```

TABLA DE CONSIDERACIONES SEMÁNTICAS

Para garantizar que todas las expresiones se evalúen correctamente y cumplan con las reglas semánticas del lenguaje, se desarrolló un cubo semántico. Este cubo permite validar en tiempo constante el tipo de dato resultante de la combinación de dos tipos de dato y un operador. Esta funcionalidad es esencial para asegurar que, al procesar una expresión, el tipo de dato del resultado coincida con el tipo esperado. Con el fin de optimizar estas consultas rápidas, se optó por crear un diccionario de diccionarios que se mapea a la siguiente tabla:

TIPO 1	TIPO 2	OPERADOR	RESULTADO
int	int	+	int
int	int	-	int
int	int	*	int
int	int	/	int
int	int	<	int
int	int	<=	int
int	int	>	int
int	int	>=	int
int	int	==	int
int	int	!=	int
int	int	and	int
int	int	or	int

TIPO 1	TIPO 2	OPERADOR	RESULTADO
float	float	+	float
float	float	-	float
float	float	*	float
float	float	/	float
float	float	<	int
float	float	<=	int
float	float	>	int
float	float	>=	int
float	float	==	int
float	float	!=	int
float	float	and	int
float	float	or	int

TIPO 1	TIPO 2	OPERADOR	RESULTADO
int	float	+	float
int	float	-	float
int	float	*	float
int	float	/	float
int	float	<	int

TIPO 1	TIPO 2	OPERADOR	RESULTADO
float	int	+	float
float	int	-	float
float	int	*	float
float	int	/	float
float	int	<	int

int	float	<=	int	float	int	<=	int
int	float	>	int	float	int	>	int
int	float	>=	int	float	int	>=	int
int	float	==	int	float	int	==	int
int	float	!=	int	float	int	!=	int
int	float	and	int	float	int	and	int
int	float	or	int	float	int	or	int

El código correspondiente a esta tabla, junto con la función para hacer las consultas sobre el tipo de dato esperado para una expresión, es el siguiente:

```
class SemanticCube:
    def __init__(self):
        self.cube = {
            "int": {
                "int": {
                    "+": "int",
                    "-": "int",
                    "*": "int",
                    "/": "int",
                    "<": "int",
                    "<=": "int",
                    ">": "int",
                    ">=": "int",
                    "==": "int",
                    "!=": "int",
                    "and": "int",
                    "or": "int",
                },
                "float": {
                    "+": "float",
                    "-": "float",
                    "*": "float",
                    "/": "float",
                    "<": "int",
                    "<=": "int",
                    ">": "int",
                    ">=": "int",
                    "==": "int",
                    "!=": "int",
                    "and": "int",
                    "or": "int",
                }
            },
            "float": {
                "int": {
                    "+": "float",
                    "-": "float",
                    "*": "float",
                    "/": "float",
                    "<": "int",
                    "<=": "int",
                    ">": "int",
                    ">=": "int",
                    "==": "int",
                    "!=": "int",
                    "and": "int",
                    "or": "int",
                },
                "float": {
                    "+": "float",
                    "-": "float",
                    "*": "float",
                    "/": "float",
                    "<": "int",
```

```
                    "<=": "int",
                    ">": "int",
                    ">=": "int",
                    "==": "int",
                    "!=": "int",
                    "and": "int",
                    "or": "int",
                }
            }
        }

def get_type(self, type1, type2, operation):
    # Get the resulting type of an operation between two types
    if type1 in self.cube and type2 in self.cube[type1]:
        if operation in self.cube[type1][type2]:
            return self.cube[type1][type2][operation]
    return None  # Invalid operation or types
```

Asimismo, se creó una estructura de datos que permite validar los tipos de datos para las declaraciones de variables, asegurando que se almacenen los tipos correctos según lo que permite cada variable. En este caso, se decidió solo permitir guardar en una variable el mismo tipo de dato. Por ejemplo, si la variable era entera, que solo se pudieran guardar valores enteros. Para ello, se implementó un diccionario de diccionarios donde la combinación de tipos de datos resulta en un booleano verdadero en caso de que sea válida. La razón de utilizar un diccionario de diccionarios es que las consultas se pueden hacer en tiempo constante, y es muy fácil agregar más combinaciones de declaraciones válidas si se desea en un futuro.

A continuación, se presenta el código que define estas reglas semánticas y la función para validar si una declaración es válida:

```
class SemanticCube:
    def __init__(self):
       self.valid_declarations = {
            "int": {
                "int": True,
            },
            "float": {
                "float": True,
            }
        }

def is_decl_valid(self, type1, type2):
    # Check if the types are valid for declaration
    return type1 in self.valid_declarations and type2 in self.valid_declarations[type1]
```

DIRECTORIO DE FUNCIONES Y TABLAS DE VARIABLES Y CONSTANTES

Para el directorio de funciones, se optó por crear cuatro clases: *SymbolTable*, *Container, Parameter* y *Symbol*. Asimismo, a esta tabla de símbolos se le agrega una tabla de constantes, la cual requirió de 2 clases: *ConstantsTable* y *Constant*.

La clase *Symbol* representa las variables y almacena información como el nombre, el tipo de dato, la dirección de memoria donde se ubica, y si es o no una constante. Cabe recalcar que debido a la segmentación de los espacios de memoria por medio de índices (se discutirá más adelante), técnicamente no sería necesario guardar el tipo de dato de las variables, ya que se podría simplemente checar en qué rango de memoria se encuentra y de esta

manera saber su tipo. Sin embargo, se optó por sí guardar el tipo, sacrificando espacio temporal en lo que se almacenan, a favor de reducir las validaciones necesarias cada vez que se ocupe utilizar el valor.

Además, se encuentra la clase *Container*, que representa los espacios de alcance de los símbolos. En esta clase se almacenan las funciones, así como el contenedor global, cuyos símbolos son accesibles desde cualquier otro contenedor. Cada contenedor tiene un nombre que lo identifica, el tipo de dato que regresa, la dirección de memoria global en donde se regresan los valores, el índice del cuádruplo donde comienza y termina el procesamiento de la función, un diccionario de los símbolos que contiene, la firma paramétrica, y un diccionario para cada tipo de variable que se utiliza dentro de la función, con valor de su cantidad usada. La dirección de memoria donde se regresan los valores permite que cuando se llame al estatuto *return*, se pueda almacenar en algún espacio de memoria global reservado para la función. De esta manera, cuando se regrese a la posición donde se hizo la llamada de la función, se puede recuperar de ese espacio lo que regresó la función. El índice de inicio permite a la máquina virtual saber a qué cuádruplo necesita brincar cuando hace una llamada de función. La firma paramétrica se trata de una lista de tipos de datos en el orden de los parámetros de la función. De esta se puede obtener el orden, cantidad y tipo de datos de los parámetros, lo que permite hacer validaciones semánticas al momento de hacer llamadas de funciones. Por último, se tiene el diccionario de espacio requerido, el cual dice cuánto se necesita reservar de memoria cuando se llame en la máquina virtual.

Finalmente, la clase *SymbolTable* es la estructura que contiene todas las funciones y el contenedor global. Esta clase tiene tres atributos: un diccionario que almacena todos los contenedores (funciones) declarados, el nombre del contenedor global para referencia posterior, y la referencia a la tabla de constantes. Durante la inicialización de la clase, se agrega el contenedor global, que siempre debe existir; los demás contenedores, correspondientes a las funciones, se añadirán a medida que se declaren. Nuevamente, se eligió un diccionario para almacenar estos contenedores debido a las ventajas que ofrece en términos de acceso eficiente.

Tanto la clase *Container* como la clase *SymbolTable* incluyen funciones que facilitan las operaciones de alta, modificación y consulta. Esto no solo permite agregar, modificar y consultar tanto los símbolos como los contenedores de manera eficiente, sino que también permite implementar validaciones y controlar el acceso a los atributos de la clase.

Además de las clases mencionadas anteriormente, se han creado dos clases adicionales para gestionar las constantes literales. Una de ellas es la clase *Constant*, que permite definir los tipos de datos de los atributos que caracterizan una constante. Los dos atributos considerados son el valor de la constante literal y su tipo de dato. Aquí, nuevamente no sería necesario guardar el tipo de dato, pero se siguió la misma lógica para reducir la complejidad temporal: una constante se declara solo una vez, por lo que guardar el tipo de dato no aporta mucho al espacio ocupado, mientras que se puede consultar múltiples veces, lo que ocasionaría que, si se requiriera validar usando los rangos de memoria cada vez, las consultas podrían contribuir a mayor tiempo de procesamiento. Por otro lado, se ha implementado la clase *ConstantsTable*, que actúa como un diccionario para almacenar las constantes, utilizando como clave la dirección de memoria donde se encuentran. Esta clase cuenta únicamente con una función para agregar nuevas constantes, ya que no es necesario modificar ni eliminar las constantes previamente creadas, y una para obtener la dirección de memoria dado una constante.

A continuación, se incluye el código del directorio de funciones, tablas de variables y tablas de constantes:

```
from dataclasses import dataclass
from typing import Literal, Union

from Exceptions import (
    ContainerRedeclarationError,
```

```
    NameNotFoundError,
    ReservedWordError,
    SymbolRedeclarationError,
)

RESERVED_WORDS = [
    "if",
    "else",
    "while",
    "do",
    "int",
    "float",
    "program",
    "main",
    "void",
    "end",
    "const",
    "var",
    "print",
    "and",
    "or",
    "return",
]


@dataclass
class Symbol:
    name: str
    var_type: Literal["int", "float", "str", "bool"]
    isConstant: bool = False
    address: int = None

    def __post_init__(self):
        if not isinstance(self.name, str):
            raise TypeError(f"Invalid type for symbol name: {type(self.name)}. Must be str.")
        if self.var_type not in ["int", "float", "str", "bool"]:
            raise ValueError(f"Invalid type: {self.var_type}. Must be 'int', 'float', 'str', or
'bool'.")


@dataclass
class Parameter:
    name: str
    var_type: Literal["int", "float", "str", "bool"]


class Container:
    def __init__(self, name, return_type: Literal["int", "float", None]):
        self.name = name
        self.return_type = return_type
        self.return_address = None
        self.initial_position = None
        self.final_position = None
        self.symbols = {}
        self.param_signature = []
        self.required_space = {}

    def add_symbol(self, symbol: Symbol) -> None:
        """Add a symbol to the container."""
        if symbol.name in RESERVED_WORDS:
            raise ReservedWordError(f"Symbol '{symbol.name}' is a reserved word and cannot be used as
an identifier.")
        if symbol.name in self.symbols:
            raise SymbolRedeclarationError(f"Symbol '{symbol.name}' already exists in '{self.name}'.")
        self.symbols[symbol.name] = symbol

    def get_symbol(self, name: str) -> Symbol:
        """Get a symbol from the container."""
```

```
        if name in self.symbols:
            return self.symbols.get(name)
        else:
            raise NameNotFoundError(f"Symbol '{name}' not found in '{self.name}'.")

    def is_symbol_declared(self, name: str) -> bool:
        """Check if a symbol is declared in the container."""
        return name in self.symbols

    def set_initial_position(self, position: int) -> None:
        """Set the initial position of the container."""
        self.initial_position = position

    def get_param_signature(self) -> list:
        """Get the parameter signature of the container."""
        return self.param_signature

    def clear(self) -> None:
        """Clear the container's symbols."""
        self.symbols.clear()


@dataclass
class Constant:
    value: Union[int, float, str, bool]
    var_type: Literal["int", "float", "str", "bool"]

    def __post_init__(self):
        if not isinstance(self.value, (int, float, str, bool)):
            raise TypeError(f"Invalid type for constant value: {type(self.value)}. Must be int, float,
str, or bool.")


class ConstantsTable:
    def __init__(self):
        self.constants = {}
        self.required_space = {"int": 0, "float": 0, "str": 0}

    def add_constant(
        self, address: int, value: Union[int, float, str, bool], value_type: Literal["int", "float",
"str", "bool"]
    ) -> None:
        """Add a constant to the table."""
        if address not in self.constants:
            self.constants[address] = Constant(value=value, var_type=value_type)
            self.required_space[value_type] += 1

    def check_and_get_address(self, value: Union[int, float, str, bool]) -> int:
        """Check if a constant exists and retrieve its address."""
        for address, constant in self.constants.items():
            if constant.value == value:
                return address
        return None


class SymbolTable:
    def __init__(self):
        self.containers = {}
        self.global_container_name = "global"
        self.constants_table = ConstantsTable()

    def get_variable(self, name: str, containerName: str) -> Symbol:
        """Get a variable from the specified container."""
        container = self.get_function(containerName)
        # Check if the symbol is declared in the specified container
        if container.is_symbol_declared(name):
            return container.get_symbol(name)
        # If not, check in the global container
```

```
        else:
            container = self.get_function(self.global_container_name)
            if container.is_symbol_declared(name):
                return container.get_symbol(name)
            else:
                raise NameNotFoundError(f"Symbol '{name}' not found in '{containerName}' or global
container.")

    def add_variable(
        self,
        name: str,
        var_type: str,
        containerName: str,
        isConstant: bool = False,
        address: int = None,
    ) -> None:
        """Add a variable to the specified container."""
        variable = Symbol(name=name, var_type=var_type, isConstant=isConstant, address=address)
        container = self.get_function(containerName)
        container.add_symbol(variable)
        container.required_space[var_type] = container.required_space.get(var_type, 0) + 1

    def add_parameter(self, name: str, var_type: str, containerName: str, address: int) -> None:
        """Add a parameter to the specified container."""
        self.add_variable(
            name=name,
            var_type=var_type,
            isConstant=False,
            containerName=containerName,
            address=address,
        )
        container = self.get_function(containerName)
        container.param_signature.append(var_type)

    def add_temp(self, var_type: str, containerName: str):
        container = self.get_function(containerName)
        container.required_space[var_type] = container.required_space.get(var_type, 0) + 1

    def add_function(self, name: str, return_type: str) -> None:
        """Add a function as a container"""
        if name in self.containers:
            raise ContainerRedeclarationError(f"Container '{name}' already exists.")
        self.containers[name] = Container(name=name, return_type=return_type)

    def create_global_container(self, id):
        """Create a global container with the given id."""
        self.add_function(id, None)
        self.global_container_name = id

    def get_function(self, name: str) -> Container:
        """Get a container by name."""
        if name not in self.containers:
            raise NameNotFoundError(f"Container {name} not found.")
        return self.containers.get(name)

    def is_function_declared(self, name: str) -> bool:
        """Check if a function is declared."""
        return name in self.containers

    def add_constant(
        self, address: int, value: Union[int, float, str, bool], value_type: Literal["int", "float",
"str", "bool"]
    ) -> None:
        """Add a constant to the constants table."""
        self.constants_table.add_constant(address=address, value=value, value_type=value_type)

    def get_return_type(self, containerName: str) -> str:
        """Get the return type of a container."""
```

```
        container = self.get_function(containerName)
        if container.return_type is None:
            raise ValueError(f"Container '{containerName}' has no return type.")
        return container.return_type

    def get_str_representation(self) -> str:
        """Return a table-like string representation of the symbol table."""
        lines = []

        # Global container
        gcontainer = self.containers.get(self.global_container_name)
        lines.append(f"Global Container: {self.global_container_name}")
        lines.append(f"Initial Position: {gcontainer.initial_position if gcontainer else 'N/A'}")
        if gcontainer and gcontainer.symbols:
            lines.append(f"{'Name':<15} {'Type':<10} {'Const':<6} {'Address':<10}")
            for symbol in gcontainer.symbols.values():
                lines.append(
                    f"{symbol.name:<15} {symbol.var_type:<10} {str(symbol.isConstant):<6}
{str(symbol.address):<10}"
                )
            # Show required space for global container
            if gcontainer.required_space:
                lines.append("Required Space:")
                for vtype, amount in gcontainer.required_space.items():
                    lines.append(f"  {vtype}: {amount}")
        else:
            lines.append("  (No symbols)")
        lines.append("")

        # Other containers (functions)
        for cname, container in self.containers.items():
            if cname == self.global_container_name:
                continue
            lines.append(f"Container: {cname}")
            lines.append(
                f"Initial Position: {container.initial_position if container.initial_position is not
None else 'N/A'}"
            )
            lines.append(f"Return Type: {container.return_type}")
            # Symbols
            lines.append(f"{'Name':<15} {'Type':<10} {'Const':<6} {'Address':<10}")
            if not container.symbols:
                lines.append("  (No symbols)")
            for symbol in container.symbols.values():
                lines.append(
                    f"{symbol.name:<15} {symbol.var_type:<10} {str(symbol.isConstant):<6}
{str(symbol.address):<10}"
                )
            # Parameters
            if container.param_signature:
                lines.append("Parameters:")
                for idx, param_type in enumerate(container.param_signature):
                    lines.append(f"  {idx + 1}. {param_type}")
            else:
                lines.append("Parameters: None")
            # Show required space for this container
            if container.required_space:
                lines.append("Required Space:")
                for vtype, amount in container.required_space.items():
                    lines.append(f"  {vtype}: {amount}")
            lines.append("")

        # Constants table
        lines.append("Constants Table:")
        if self.constants_table.constants:
            lines.append(f"{'Address':<10} {'Value':<20} {'Type':<10}")
            for address, constant in self.constants_table.constants.items():
                lines.append(f"{str(address):<10} {str(constant.value):<20} {constant.var_type:<10}")
```

```
        else:
            lines.append("  (No constants)")
        lines.append("")

        return "\n".join(lines)

    def __str__(self):
        """String representation of the symbol table."""
        return self.get_str_representation()
```

CÓDIGO INTERMEDIO: CUÁDRUPLOS

Para la generación del código intermedio, se eligió una representación basada en cuádruplos, estructuras compuestas por cuatro elementos: operación, primer argumento, segundo argumento y resultado. Para implementar cada cuádruplo, se utilizaron tuplas, ya que permiten definir esta estructura de manera sencilla y eficiente.

Durante el proceso de compilación, los cuádruplos se van generando conforme se procesan las instrucciones del programa y se almacenan en una cola. La elección de una cola como estructura de datos obedece a que los cuádruplos deben ser procesados siguiendo un orden FIFO (First-In-First-Out), es decir, los primeros cuádruplos generados serán los primeros en ser ejecutados por la máquina virtual. Esto asegura que las operaciones se lleven a cabo en el orden correcto durante la ejecución del código intermedio.

Además de la cola de cuádruplos, se utilizan dos estructuras auxiliares adicionales para gestionar información temporal durante la generación del código intermedio. Estas son pilas, que siguen el principio LIFO (Last-In-First-Out).

La primera pila se utiliza para almacenar las posiciones de salto dentro de la cola de cuádruplos. Esto resulta necesario cuando se genera una instrucción de salto cuyo destino aún no es conocido, como ocurre en sentencias condicionales o bucles. Un salto permite modificar el flujo de ejecución del programa intermedio, ya sea de forma incondicional o dependiendo del resultado de una condición (por ejemplo, goto verdadero o goto falso). Esta funcionalidad es clave para implementar estructuras de control como while.

Por ejemplo, en un ciclo while, existe una condición inicial que determina si se ejecuta el cuerpo del ciclo. Si la condición es falsa, se debe realizar un salto hacia el final del ciclo. Por otro lado, una vez que se completa la ejecución del cuerpo del ciclo, se requiere un salto incondicional para regresar y volver a evaluar la condición. Estos dos tipos de saltos se representan mediante cuádruplos específicos: uno que depende de una condición y otro que se ejecuta siempre.

El segundo uso de las pilas está relacionado con el registro de los puntos de retorno. En ciertas estructuras, como los bucles, es necesario recordar desde qué posición del código intermedio se debe reanudar la ejecución después de realizar un salto. Por ello, se almacena temporalmente dicha posición en una pila, garantizando que, al cerrarse una estructura anidada, se retome correctamente el flujo de ejecución.

La decisión de usar pilas para estas tareas se fundamenta en la necesidad de resolver primero las estructuras de control más internas en caso de que estén anidadas, lo cual se logra naturalmente con el comportamiento LIFO de las pilas.

Finalmente, aunque no es estrictamente necesario, se implementó una tabla de mapeo que asigna un valor numérico a cada operación disponible. Esta tabla permite almacenar únicamente números en los cuádruplos en lugar de cadenas de texto, reduciendo la posibilidad de errores y optimizando el manejo interno del código intermedio.

Identificador	Operador	Identificador	Operador
1	+	**13**	goto
2	-	**14**	gotoF
3	*	**15**	gotoT
4	/	**16**	=
5	<	**17**	print
6	≤	**18**	era
7	>	**19**	param
8	≥	**20**	gosub
9	==	**21**	return
10	!=	**22**	endFunc
11	and	**23**	end
12	or		

A continuación, se muestra el código desarrollado para la generación de cuádruplos:

```
from collections import deque


class OperatorsInterface:
    def __init__(self):
        self.operators = {
            "+": 1,
            "-": 2,
            "*": 3,
            "/": 4,
            "<": 5,
            "<=": 6,
            ">": 7,
            ">=": 8,
            "==": 9,
            "!=": 10,
            "and": 11,
            "or": 12,
            "goto": 13,
            "gotoF": 14,
            "gotoT": 15,
            "=": 16,
            "print": 17,
            "era": 18,
            "param": 19,
            "gosub": 20,
            "return": 21,
            "endFunc": 22,
            "end": 23,
        }
```

```
    def get_operator(self, op: str):
        """Get the operator."""
        return self.operators.get(op, None)


class QuackQuadruple:
    def __init__(self):
        self.jumps_stack = []
        self.returns_stack = []
        self.quadruples = deque()
        self.current_index = 0
        self.operators = OperatorsInterface()

    def get_current_index(self):
        """Get the current index."""
        return self.current_index

    def add_quadruple(
        self, op: str, arg1: str, arg2: str, result: str = None, memory_space: str = None, result_type:
str = None
    ):
        """Add a quadruple to the list."""
        op = self.operators.get_operator(op)
        self.quadruples.append((op, arg1, arg2, result))
        self.current_index += 1
        return result

    def add_return(self, return_value=None):
        """Add a return to the list."""
        if return_value is None:
            return_value = self.current_index

        self.returns_stack.append(return_value)

    def add_jump(self, type: str = "goto", condition: str = None, target: str = None):
        """Add a jump to the list."""
        type = self.operators.get_operator(type)
        self.quadruples.append((type, condition, None, target))
        self.current_index += 1

    def get_quadruples(self):
        """Get the list of quadruples."""
        return list(self.quadruples)

    def push_jump(self, jump: int = None):
        """Push a jump onto the stack."""
        if jump is None:
            jump = self.current_index
        self.jumps_stack.append(jump)

    def pop_jump(self):
        """Pop the last jump from the stack."""
        if self.jumps_stack:
            return self.jumps_stack.pop()
        return None

    def pop_return(self):
        """Pop the last return from the stack."""
        if self.returns_stack:
            return self.returns_stack.pop()
        return None

    def update_jump(self, index: int, target: int):
        """Update the jump at the given index."""
        if 0 <= index < len(self.quadruples):
            op, arg1, arg2, _ = self.quadruples[index]
            self.quadruples[index] = (op, arg1, arg2, target)
        else:
```

```
            raise IndexError("Jump index out of range")

    def get_str_representation(self, pretty: bool = False):
        """Get a string representation of the quadruples."""
        lines = []
        for i, quadruple in enumerate(self.quadruples):
            op = quadruple[0]
            op_str = next((k for k, v in self.operators.operators.items() if v == op), op) if pretty
else op
            lines.append(f"{i}: ({op_str}, {quadruple[1]}, {quadruple[2]}, {quadruple[3]})")
        return "\n".join(lines)

    def __str__(self):
        """Get a string representation of the quadruples."""
        return "\n".join([f"{i}: {quadruple}" for i, quadruple in enumerate(self.quadruples)])

    def __repr__(self):
        """Get a string representation of the quadruples."""
        return self.__str__()
```

PUNTOS NEURÁLGICOS

No se implementaron puntos neurálgicos directamente en el parser de Lark debido a las limitaciones inherentes al análisis sintáctico Bottom-Up que utiliza LALR(1). En este tipo de análisis, el parser reconoce los elementos básicos (como los tokens de tipo ID) desde las hojas del árbol sintáctico hacia arriba, sin tener aún visibilidad del contexto global o las reglas no terminales que engloban esos elementos. El Transformer de Lark opera en modo post-order (procesa hijos antes que padres), lo que refuerza su inadecuación para tareas que requieren conocimiento ascendente (como el alcance de variables). Por ejemplo, un ID podría corresponder a una variable declarada, una llamada a función, un parámetro, o incluso parte de una expresión. Dado que LALR(1) no puede resolver estos matices durante el proceso de reducción, cualquier intento por asociar semántica específica (como registrar una variable en el directorio de funciones) en tiempo de parsing resultaría en ambigüedades o errores. Además, Lark tiene la particularidad que busca siempre generar un árbol sintáctico usando las reglas gramaticales provistas, por lo que, entre las capacidades de la herramienta, no existe una manera de inyectar código.

Para abordar esto, se adoptó un enfoque de dos etapas:

- **Transformación del árbol sintáctico:**
  La clase Transformer en Lark se utilizó exclusivamente para reestructurar el árbol de análisis en una estructura abstracta (AST) más manejable.
- **Interpretación con un recorrido Top-Down:**
  Una vez construido el AST, se pasó a una clase Interpreter que procesa el árbol de manera recursiva desde la raíz hacia las hojas (enfoque Top-Down). Esto permite establecer el contexto necesario (como el ámbito actual o la función en ejecución) antes de analizar los identificadores. Aquí es donde se encuentran los puntos neurálgicos para las declaraciones de variables y funciones. Por ejemplo, al encontrar una declaración de función, el intérprete crea una entrada en el Directorio de Funciones y, al procesar sus parámetros o variables locales, llena las tablas de variables asociadas al contexto actual. Este diseño asegura que cada ID se interprete con base en su entorno semántico, evitando conflictos y respetando el alcance de las declaraciones.

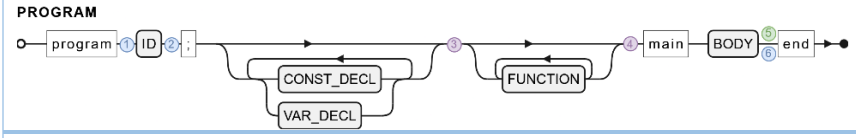
Este enfoque trae consigo algunas ventajas, entre las cuales puedo resaltar:

- **Separación de responsabilidades:** El parser/transfomer se enfoca en la estructura sintáctica, mientras que el intérprete maneja la semántica contextual.
- **Flexibilidad semántica:** Al posponer la interpretación, se evitan decisiones prematuras sobre el rol de un identificador.
- **Escalabilidad:** El modelo permite añadir nuevas reglas de contexto (como manejo de clases o módulos) sin modificar la gramática o el parser.

A continuación, se incluyen los diagramas de la gramática con los puntos neurálgicos resaltados. Asimismo, se incluye el fragmento de código que procesa la regla:

Referencia del color de los puntos neurálgicos

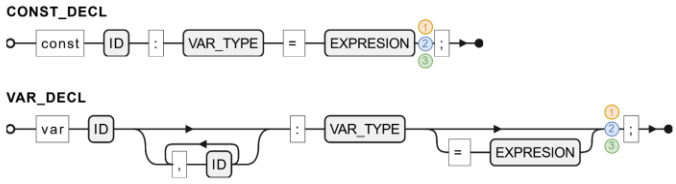
Tabla de símbolos ● ● Cuádruplos: Saltos
Cuádruplos ● ● Memoria




```
"""
program: program_pt1 program_pt2 MAIN body END → program_no_decl
    | program_pt1 program_pt2 (const_decl | var_decl)+ MAIN body END → program_decl_no_func
    | program_pt1 program_pt2 function+ MAIN body END → program_func_no_decl
    | program_pt1 program_pt2 (const_decl | var_decl)+ function+ MAIN body END → program_decl_func
?program_pt1: PROGRAM
?program_pt2: id SEMICOLON
"""

def program_pt1(self, program):
    self.symbol_table = SymbolTable()  (1)
    return program

def program_pt2(self, id, semicolon):
    self.symbol_table.create_global_container(id.name)  (2)
    return id.name
```

```
elif isinstance(ir, ProgramNode):
    for decl in ir.global_decls:
        self.execute(decl)

    self.quack_quadruple.push_jump()  (3)
    self.quack_quadruple.add_jump(type="goto")

    for func in ir.functions:
        self.execute(func)

    self.quack_quadruple.update_jump(  (4)
        index=self.quack_quadruple.pop_jump(), target=self.quack_quadruple.get_current_index()
    )

    self.execute(ir.main_body)

    self.quack_quadruple.add_quadruple("end", None, None, None)  (5)

    self.symbol_table.get_function(self.global_container_name).clear()  (6)
else:
    raise UnknownIRTypeError(f"Unknown IR type: {type(ir)}")
```

1. Inicializar la tabla de símbolos
2. Agregar el contenedor global a la tabla de símbolos
3. Agregar salto al inicio de “main”. Como todavía no se sabe dónde estará este inicio, se agrega también la posición de este cuádruplo de salto para regresar a actualizar su destino.
4. Posterior a declarar todas las funciones, ya se conoce la posición de inicio de main, por lo que se actualiza el salto anterior.
5. Agregar cuádruplo de fin de programa.
6. Borrar datos innecesarios declarados globalmente.

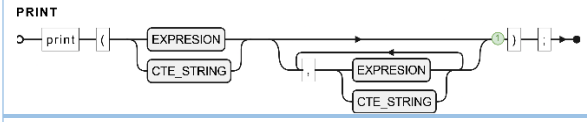



```
elif isinstance(ir, VarDeclNode):
    var_type = ir.var_type

    value, value_type = (None, None)
    if ir.init_value:
        value, value_type = self.__evaluate_expression(ir.init_value)
        if not self.semantic_cube.is_decl_valid(var_type, value_type):
            raise TypeMismatchError(
                f"Cannot assign value of type '{value_type}' to variable of type '{var_type}'"
            )

    for var_name in ir.names:
        address = self.memory_manager.get_first_available_address(  (1)
            var_type=var_type,
            space=self.current_memory_space,
        )
        self.symbol_table.add_variable(
            name=var_name.name,
            var_type=var_type,                                      (2)
            containerName=self.current_container,
            isConstant=ir.isConstant,
            address=address,
        )
        self.quack_quadruple.add_quadruple("=", value, None, address)  (3)
```

1. Obtener la primera dirección de memoria disponible en el espacio de memoria actual (global o local) y para el tipo de dato de la variable.
2. Agregar variable a la tabla de símbolos
3. Agregar cuádruplo de asignación


```
if isinstance(ir, AssignNode):
    var_name = ir.var_name
    variable = self.symbol_table.get_variable(name=var_name, containerName=self.current_container)  (1)

    if variable.isConstant:
        raise TypeMismatchError(f"Cannot reassign constant variable '{var_name}'")

    value, value_type = self.__evaluate_expression(ir.expr)

    if self.semantic_cube.is_decl_valid(variable.var_type, value_type):  (2)
        self.quack_quadruple.add_quadruple("=", value, None, variable.address)
    else:
        raise TypeMismatchError(
            f"Cannot assign type '{value_type}' to variable '{var_name}' of type '{variable.var_type}'"
        )
```

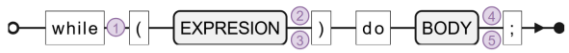
1. Obtener la variable a sobrescribir
2. Agregar cuádruplo de asignación actualizando valor asignado




```
elif isinstance(ir, PrintNode):
    for value in ir.values:
        value, value_type = self._resolve_operand(value)        (1)
        self.quack_quadruple.add_quadruple("print", None, None, value)
```

1. Agregar cuádruplo de impresión

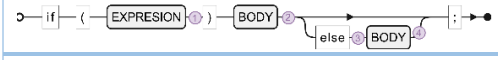
CYCLE




```
elif isinstance(ir, WhileNode):
    self.quack_quadruple.add_return()
    condition = self._resolve_operand(ir.condition)

    self.quack_quadruple.push_jump()
    self.quack_quadruple.add_jump(type="gotoF", condition=condition[0])

    self.execute(ir.body)

    self.quack_quadruple.add_jump(type="goto", target=self.quack_quadruple.pop_return())
    self.quack_quadruple.update_jump(
        index=self.quack_quadruple.pop_jump(), target=self.quack_quadruple.get_current_index()
    )
```

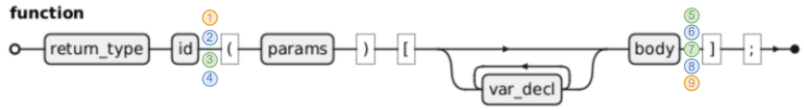
1. Agregar punto de regreso previo a las expresiones a la pila de regresos
2. Agregar posición de salto a la pila de saltos para poder regresar a actualizar el destino
3. Agregar salto con condición falsa para saltar al final del ciclo
4. Agregar el salto sin condición hacia la posición de regreso a los cuádruplos
5. Actualizar el destino del salto falso al inicio del ciclo

CONDITION



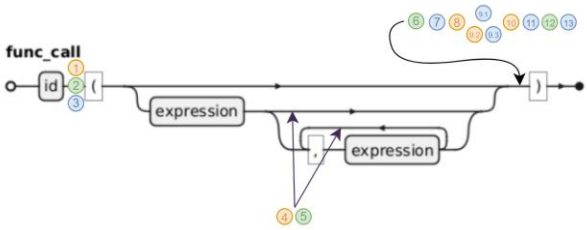

```
elif isinstance(ir, IfNode):
    value, value_type = self._resolve_operand(ir.condition)

    self.quack_quadruple.push_jump()
    self.quack_quadruple.add_jump(type="gotoF", condition=value)

    self.execute(ir.then_body)

    self.quack_quadruple.update_jump(
        index=self.quack_quadruple.pop_jump(), target=self.quack_quadruple.get_current_index()
    )
##########################################################################################
elif isinstance(ir, IfElseNode):
    value, value_type = self._resolve_operand(ir.condition)

    self.quack_quadruple.push_jump()
    self.quack_quadruple.add_jump(type="gotoF", condition=value, target=None)

    self.execute(ir.then_body)

    # Add 1 to the current index to skip the else block
    self.quack_quadruple.update_jump(
        index=self.quack_quadruple.pop_jump(), target=self.quack_quadruple.get_current_index() + 1
    )

    self.quack_quadruple.push_jump()
    self.quack_quadruple.add_jump()

    self.execute(ir.else_body)

    self.quack_quadruple.update_jump(
        index=self.quack_quadruple.pop_jump(), target=self.quack_quadruple.get_current_index()
    )
```

1. Agregar salto falso para el if y guardar posición de salto en pila de saltos
2. Actualizar posición de destino del salto falso al inicio del condicional if
3. Agregar salto sin condición para el else y guardar posición de salto en pila de saltos
4. Actualizar posición de destino del salto del else




```python
elif isinstance(ir, FunctionDeclNode):
    func_name = ir.name.name
    func_return_type = ir.return_type
    func_params = ir.params.params
    func_body = ir.body
    func_var_decls = ir.var_decls

    self.current_container = func_name
    self.current_memory_space = "local"

    old_memory = self.memory_manager.replace_memory_space(
        "local",
        Memory(
            mapping={
                "int": ((5000, 5999), 0),
                "float": ((6000, 6999), 0),
                "t_int": ((7000, 7999), 0),
                "t_float": ((8000, 8999), 0),
            }
        ),
    )

    self.symbol_table.add_function(name=func_name, return_type=func_return_type)

    starting_index = self.quack_quadruple.get_current_index()
    self.symbol_table.get_function(func_name).initial_position = starting_index

    for param in func_params:
        self.execute(param)

    for var_decl in func_var_decls:
        self.execute(var_decl)

    self.execute(func_body)

    final_index = self.quack_quadruple.get_current_index()
    self.symbol_table.get_function(func_name).final_position = final_index - 1

    self.quack_quadruple.add_quadruple("endFunc", None, None, func_name)

    self.current_container = self.global_container_name
    self.current_memory_space = "global"
    self.symbol_table.get_function(func_name).clear()

    self.memory_manager.replace_memory_space("local", old_memory)
```

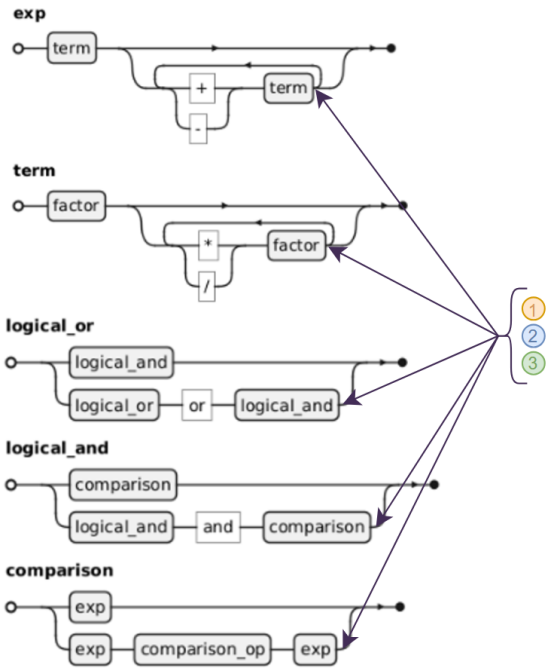
1. Reemplazar la memoria local actual con una nueva para poder llevar seguimiento de los temporales y variables locales.
2. Agregar al directorio de funciones la función con su nombre y tipo de dato de regreso.
3. Obtener el índice de los cuádruplos actual.
4. Asignar la posición de inicio de la función a la función en el directorio de funciones.
5. Obtener el índice de los cuádruplos actual.
6. Asignar la posición de fin de la función a la función en el directorio de funciones.
7. Agregar cuádruplo de fin de función.
8. Borrar la información sobre la función actual guardada en el directorio de funciones que ya no se va a necesitar.
9. Reemplazar la memoria local con la que se había quitado, dado que ya no se necesita conservar la de la función.




```
if isinstance(expr_tree, FuncCallNode):
    func_name = expr_tree.name.name
    return_type = self.__process_func_call(expr_tree)

    if return_type ≠ "void":
        if self.symbol_table.get_function(self.global_container_name).is_symbol_declared(name=func_name):
            func_address = self.symbol_table.get_variable(
                name=func_name, containerName=self.global_container_name
            ).address
        else:
            func_address = self.memory_manager.get_first_available_address(
                var_type=return_type,
                space="global",
            )
            self.symbol_table.add_variable(
                name=func_name,
                var_type=return_type,
                containerName=self.global_container_name,
                address=func_address,
            )
        temp_address = self.memory_manager.get_first_available_address(
            var_type=f"t_{return_type}",
            space=self.current_memory_space,
        )
        self.symbol_table.add_temp(
            var_type=f"t_{return_type}",
            containerName=self.current_container,
        )
        self.quack_quadruple.add_quadruple("=", func_address, None, temp_address)
        self.symbol_table.get_function(func_name).return_address = func_address

        return temp_address, return_type
    else:
        raise TypeMismatchError(
            f"Function '{func_name}' does not return a value, cannot be used in an expression."
        )
```

```
def __process_func_call(self, func_call):
    func_name = func_call.name.name
    func_args = func_call.args

    old_memory = self.memory_manager.replace_memory_space(
        "local",
        Memory(
            mapping={
                "int": ((5000, 5999), 0),
                "float": ((6000, 6999), 0),
                "t_int": ((7000, 7999), 0),
                "t_float": ((8000, 8999), 0),
            }
        ),
    )

    self.quack_quadruple.add_quadruple("era", None, None, func_name)

    param_signature = self.symbol_table.get_function(func_name).get_param_signature()

    if len(param_signature) ≠ len(func_args):
        raise TypeMismatchError(
            f"Function '{func_name}' expects {len(param_signature)} arguments, but got {len(func_args)}"
        )

    for i, arg in enumerate(func_args):
        arg_value, arg_type = self.__evaluate_expression(arg)

        if not self.semantic_cube.is_decl_valid(param_signature[i], arg_type):
            raise TypeMismatchError(
                f"Argument {i + 1} of function '{func_name}' expects type '{param_signature[i]}', but got '{arg_type}'"
            )

        address_in_function = self.memory_manager.get_first_available_address(
            var_type=arg_type,
            space="local",
        )

        self.quack_quadruple.add_quadruple("param", arg_value, None, address_in_function)

    self.quack_quadruple.add_quadruple("gosub", None, None, func_name)

    return_type = self.symbol_table.get_return_type(func_name)

    self.memory_manager.replace_memory_space("local", old_memory)

    return return_type
```

1. Reemplazar memoria local con una temporal para llevar seguimiento de los índices de los temporales y declaraciones locales.
2. Agregar cuádruplo de “era”.
3. Obtener la firma paramétrica.
4. Obtener la primera dirección de memoria para el tipo de argumento y en el espacio local.
5. Crear cuádruplo de “param” pasando la dirección local.
6. Agregar un cuádruplo de gosub hacia el nombre de la función.
7. Obtener el tipo de dato que se declaró que la función debería regresar.
8. Regresar la memoria local que se tenía anteriormente.
9. Si no es void:
   Si ya se había declarado como variable global:
   9.1. Obtener la dirección de memoria donde se declaró.
   Si no había sido declarda:
   9.2. Obtener un espacio de memoria en el espacio global.
   9.3. Agregar la función como variable global en ese espacio de memoria.
10. Obtener la primera dirección de memoria en el espacio de memoria actual (ya sea local o global, dependiendo desde donde se hizo la llamada de función).
11. Agregar un temporal en el contendor desde donde se hizo la llamada (ya sea una función o main).
12. Agregar un cuádruplo asignando lo guardado globalmente de la función, en la dirección temporal de memoria.
13. Guardar la dirección de memoria global donde se guardan los valores que se regresan en el directorio de funciones.




```
elif (
    isinstance(expr_tree, MultiplicativeOpNode)
    or isinstance(expr_tree, ArithmeticOpNode)
    or isinstance(expr_tree, ComparisonNode)
    or isinstance(expr_tree, LogicalAndNode)
    or isinstance(expr_tree, LogicalOrNode)
):
    left_value, left_type = self.__evaluate_expression(expr_tree.left)
    right_value, right_type = self.__evaluate_expression(expr_tree.right)

    result_type = self.semantic_cube.get_type(left_type, right_type, expr_tree.op)

    if result_type is None:
        raise UnsupportedOperationError(
            f"Unsupported operation '{expr_tree.op}' for types '{left_type}' and '{right_type}'"
        )

    func_address = self.memory_manager.get_first_available_address(  ①
        var_type=f"t_{result_type}",
        space=self.current_memory_space,
    )
    self.symbol_table.add_temp(
        var_type=f"t_{result_type}",          ②
        containerName=self.current_container,
    )

    result = self.quack_quadruple.add_quadruple(      ③
        op=expr_tree.op, arg1=left_value, arg2=right_value, result=func_address
    )
    return result, result_type
```

1. Obtener la primera dirección de memoria temporal.
2. Agregar el temporal al contenedor actual.
3. Agregar un cuádruplo con el operador, los dos argumentos y la dirección de memoria temporal donde se guardará el resultado.

param


```
elif isinstance(ir, ParamNode):
    param_name = ir.name.name
    param_type = ir.param_type

    address = self.memory_manager.get_first_available_address(
        var_type=param_type,                    (1)
        space=self.current_memory_space,
    )
    self.symbol_table.add_parameter(
        name=param_name,
        var_type=param_type,                    (2)
        containerName=self.current_container,
        address=address,
    )
```

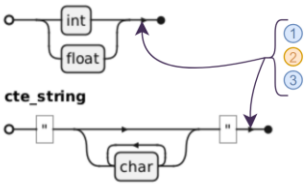
1. Obtener la primera dirección de memoria para guardar un parámetro (como variable local) de cierto tipo de dato, y en el espacio de memoria local.
2. Agregar a la tabla de símbolos el parámetro para tener la referencia como variable local, al igual que agregarlo a la firma paramétrica.

return


```
elif isinstance(ir, ReturnNode):
    return_value = self.__evaluate_expression(ir.expresion)      (1)
    self.quack_quadruple.add_quadruple("return", self.current_container, None, return_value[0])
```

1. Agregar cuádruplo de “return” con el nombre del contenedor actual como argumento 1 y la dirección de memoria del resultado de la expresión que se está regresando.

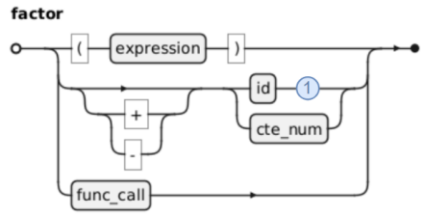
cte_num




```
elif isinstance(expr_tree, CteNumNode) or isinstance(expr_tree, CteStringNode):
    var_type = type(expr_tree.value).__name__
    value = expr_tree.value
                                                                    (1)
    constant_address = self.symbol_table.constants_table.check_and_get_address(value)

    if constant_address is not None:
        result = constant_address
    else:
        result = self.memory_manager.get_first_available_address(
            var_type=var_type,
            space="constant",                                       (2)
        )                                                                   (3)
        self.symbol_table.add_constant(address=result, value=value, value_type=var_type)

    return result, var_type
```

1. En caso de que la constante ya esté declarada, se obtiene su dirección de memoria.
   Si la constante no ha sido declarada:
2. Obtener la primera dirección de memoria en el espacio de memoria de las constantes para el tipo de dato que se va a guardar.

3. Guardar la constante en la tabla de constantes.




```
if isinstance(expr_tree, IdNode):
    var_name = expr_tree.name
    variable = self.symbol_table.get_variable(name=var_name, containerName=self.current_container)
    value = variable.address
    var_type = variable.var_type
    return value, var_type
```

1. Obtener de la tabla de símbolos la dirección en memoria del id y su tipo de dato.

INTERPRETE

Para interpretar el código procesado por el parser, se desarrolló una clase llamada *Interpreter*. Esta clase permite recorrer el árbol sintáctico transformado mediante un enfoque recursivo. Utiliza las clases de datos anteriormente definidas para determinar cómo procesar cada elemento del árbol. En este punto, las representaciones de las acciones del lenguaje se convierten en operaciones concretas: declarar variables, asignarles valores, ejecutar ciclos, imprimir resultados, realizar acciones condicionadas basadas en expresiones, así como declarar y llamar a funciones y evaluar expresiones.

Dentro de esta clase, se emplea el cubo semántico para garantizar que las acciones realizadas se ajusten a la semántica definida para el lenguaje. Como se mencionó en la sección previa, es en esta clase donde se encuentran los puntos neurálgicos para las declaraciones de variables y funciones. Mientras se va recorriendo el AST, se hacen pausas en el análisis para hacer las acciones correspondientes. Lo anterior se logra por medio de la integración de una tabla de símbolos que permite llevar un seguimiento de los símbolos y contenedores que se van declarando, un manejador de memoria que gestiona los espacios en memoria disponibles, y una clase que permite agregar cuádruplos de manera sencilla. Además, para asegurar un manejo adecuado del alcance de los símbolos, se utiliza una variable global que especifica el contenedor actual (ya sea global o dentro de una función). De este modo, se pueden declarar y consultar las variables correspondientes al alcance en el que se encuentra el intérprete. También, aquí ya se utiliza el manejador de memoria que se definirá más adelante para llevar seguimiento de los espacios de memoria que se ocupará reservar en la máquina virtual, con respecto a los espacios que se están usando en esta etapa.

De esta clase se pueden resaltar 3 funciones, las cuales permiten procesar estatutos, expresiones y llamadas de funciones. La decisión de qué proceso seguir con base en la línea de código que se está procesando dependerá de la clase que se utilizó durante el proceso de transformación.

A continuación, se presenta el código del intérprete desarrollado:

```
from Exceptions import (
    TypeMismatchError,
    UnknownIRTypeError,
    UnsupportedOperationError,
)
from MemoryManager import Memory
```

```
from SemanticCube import SemanticCube
from TransformerClasses import (
    ArithmeticOpNode,
    AssignNode,
    BodyNode,
    ComparisonNode,
    CteNumNode,
    CteStringNode,
    FuncCallNode,
    FunctionDeclNode,
    IdNode,
    IfElseNode,
    IfNode,
    LogicalAndNode,
    LogicalOrNode,
    MultiplicativeOpNode,
    ParamNode,
    PrintNode,
    ProgramNode,
    ReturnNode,
    VarDeclNode,
    WhileNode,
)


class QuackInterpreter:
    def __init__(self, symbol_table, quack_quadruple, memory_manager):
        self.memory_manager = memory_manager
        self.symbol_table = symbol_table
        self.global_container_name = self.symbol_table.global_container_name
        self.current_container = self.global_container_name
        self.semantic_cube = SemanticCube()
        self.quack_quadruple = quack_quadruple
        self.current_memory_space = "global"

    def __process_func_call(self, func_call):
        func_name = func_call.name.name
        func_args = func_call.args

        old_memory = self.memory_manager.replace_memory_space(
            "local",
            Memory(
                mapping={
                    "int": ((5000, 5999), 0),
                    "float": ((6000, 6999), 0),
                    "t_int": ((7000, 7999), 0),
                    "t_float": ((8000, 8999), 0),
                }
            ),
        )

        self.quack_quadruple.add_quadruple("era", None, None, func_name)

        param_signature = self.symbol_table.get_function(func_name).get_param_signature()

        if len(param_signature) != len(func_args):
            raise TypeMismatchError(
                f"Function '{func_name}' expects {len(param_signature)} arguments, but got
{len(func_args)}"
            )

        for i, arg in enumerate(func_args):
            arg_value, arg_type = self.__evaluate_expression(arg)

            if not self.semantic_cube.is_decl_valid(param_signature[i], arg_type):
                raise TypeMismatchError(
                    f"Argument {i + 1} of function '{func_name}' expects type '{param_signature[i]}',
but got '{arg_type}'"
```

```
                )

            address_in_function = self.memory_manager.get_first_available_address(
                var_type=arg_type,
                space="local",
            )

            self.quack_quadruple.add_quadruple("param", arg_value, None, address_in_function)

        self.quack_quadruple.add_quadruple("gosub", None, None, func_name)

        return_type = self.symbol_table.get_return_type(func_name)

        self.memory_manager.replace_memory_space("local", old_memory)

        return return_type

    def __evaluate_expression(self, expr_tree):
        if isinstance(expr_tree, IdNode):
            var_name = expr_tree.name
            variable = self.symbol_table.get_variable(name=var_name,
containerName=self.current_container)
            value = variable.address
            var_type = variable.var_type
            return value, var_type

        if isinstance(expr_tree, FuncCallNode):
            func_name = expr_tree.name.name
            return_type = self.__process_func_call(expr_tree)

            if return_type != "void":
                if
self.symbol_table.get_function(self.global_container_name).is_symbol_declared(name=func_name):
                    func_address = self.symbol_table.get_variable(
                        name=func_name, containerName=self.global_container_name
                    ).address
                else:
                    func_address = self.memory_manager.get_first_available_address(
                        var_type=return_type,
                        space="global",
                    )
                    self.symbol_table.add_variable(
                        name=func_name,
                        var_type=return_type,
                        containerName=self.global_container_name,
                        address=func_address,
                    )
                temp_address = self.memory_manager.get_first_available_address(
                    var_type=f"t_{return_type}",
                    space=self.current_memory_space,
                )
                self.symbol_table.add_temp(
                    var_type=f"t_{return_type}",
                    containerName=self.current_container,
                )
                self.quack_quadruple.add_quadruple("=", func_address, None, temp_address)
                self.symbol_table.get_function(func_name).return_address = func_address

                return temp_address, return_type
            else:
                raise TypeMismatchError(
                    f"Function '{func_name}' does not return a value, cannot be used in an expression."
                )

        elif isinstance(expr_tree, CteNumNode) or isinstance(expr_tree, CteStringNode):
            var_type = type(expr_tree.value).__name__
            value = expr_tree.value
```

```
            constant_address = self.symbol_table.constants_table.check_and_get_address(value)

            if constant_address is not None:
                result = constant_address
            else:
                result = self.memory_manager.get_first_available_address(
                    var_type=var_type,
                    space="constant",
                )
                self.symbol_table.add_constant(address=result, value=value, value_type=var_type)

            return result, var_type

        elif (
            isinstance(expr_tree, MultiplicativeOpNode)
            or isinstance(expr_tree, ArithmeticOpNode)
            or isinstance(expr_tree, ComparisonNode)
            or isinstance(expr_tree, LogicalAndNode)
            or isinstance(expr_tree, LogicalOrNode)
        ):
            left_value, left_type = self.__evaluate_expression(expr_tree.left)
            right_value, right_type = self.__evaluate_expression(expr_tree.right)

            result_type = self.semantic_cube.get_type(left_type, right_type, expr_tree.op)

            if result_type is None:
                raise UnsupportedOperationError(
                    f"Unsupported operation '{expr_tree.op}' for types '{left_type}' and
'{right_type}'"
                )

            func_address = self.memory_manager.get_first_available_address(
                var_type=f"t_{result_type}",
                space=self.current_memory_space,
            )
            self.symbol_table.add_temp(
                var_type=f"t_{result_type}",
                containerName=self.current_container,
            )

            result = self.quack_quadruple.add_quadruple(
                op=expr_tree.op, arg1=left_value, arg2=right_value, result=func_address
            )
            return result, result_type

        else:
            raise UnsupportedOperationError(f"Unsupported expression type: {type(expr_tree)}")
            # return expr_tree

    def execute(self, ir):
        if isinstance(ir, AssignNode):
            var_name = ir.var_name
            variable = self.symbol_table.get_variable(name=var_name,
containerName=self.current_container)

            if variable.isConstant:
                raise TypeMismatchError(f"Cannot reassign constant variable '{var_name}'")

            value, value_type = self.__evaluate_expression(ir.expr)

            if self.semantic_cube.is_decl_valid(variable.var_type, value_type):
                self.quack_quadruple.add_quadruple("=", value, None, variable.address)
            else:
                raise TypeMismatchError(
                    f"Cannot assign type '{value_type}' to variable '{var_name}' of type
'{variable.var_type}'"
                )
```

```
        elif isinstance(ir, VarDeclNode):
            var_type = ir.var_type

            value, value_type = (None, None)
            if ir.init_value:
                value, value_type = self.__evaluate_expression(ir.init_value)
                if not self.semantic_cube.is_decl_valid(var_type, value_type):
                    raise TypeMismatchError(
                        f"Cannot assign value of type '{value_type}' to variable of type '{var_type}'"
                    )

            for var_name in ir.names:
                address = self.memory_manager.get_first_available_address(
                    var_type=var_type,
                    space=self.current_memory_space,
                )
                self.symbol_table.add_variable(
                    name=var_name.name,
                    var_type=var_type,
                    containerName=self.current_container,
                    isConstant=ir.isConstant,
                    address=address,
                )
                self.quack_quadruple.add_quadruple("=", value, None, address)

        elif isinstance(ir, BodyNode):
            for statement in ir.statements:
                self.execute(statement)

        elif isinstance(ir, WhileNode):
            self.quack_quadruple.add_return()
            condition = self.__evaluate_expression(ir.condition)

            self.quack_quadruple.push_jump()
            self.quack_quadruple.add_jump(type="gotoF", condition=condition[0])

            self.execute(ir.body)

            self.quack_quadruple.add_jump(type="goto", target=self.quack_quadruple.pop_return())
            self.quack_quadruple.update_jump(
                index=self.quack_quadruple.pop_jump(), target=self.quack_quadruple.get_current_index()
            )

        elif isinstance(ir, PrintNode):
            for value in ir.values:
                value, value_type = self.__evaluate_expression(value)
                self.quack_quadruple.add_quadruple("print", None, None, value)

        elif isinstance(ir, IfNode):
            value, value_type = self.__evaluate_expression(ir.condition)

            self.quack_quadruple.push_jump()
            self.quack_quadruple.add_jump(type="gotoF", condition=value)

            self.execute(ir.then_body)

            self.quack_quadruple.update_jump(
                index=self.quack_quadruple.pop_jump(), target=self.quack_quadruple.get_current_index()
            )

        elif isinstance(ir, IfElseNode):
            value, value_type = self.__evaluate_expression(ir.condition)

            self.quack_quadruple.push_jump()
            self.quack_quadruple.add_jump(type="gotoF", condition=value, target=None)

            self.execute(ir.then_body)
```

```
            # Add 1 to the current index to skip the else block
            self.quack_quadruple.update_jump(
                index=self.quack_quadruple.pop_jump(), target=self.quack_quadruple.get_current_index()
+ 1
            )

            self.quack_quadruple.push_jump()
            self.quack_quadruple.add_jump()

            self.execute(ir.else_body)

            self.quack_quadruple.update_jump(
                index=self.quack_quadruple.pop_jump(), target=self.quack_quadruple.get_current_index()
            )

        elif isinstance(ir, FunctionDeclNode):
            func_name = ir.name.name
            func_return_type = ir.return_type
            func_params = ir.params.params
            func_body = ir.body
            func_var_decls = ir.var_decls

            self.current_container = func_name
            self.current_memory_space = "local"

            old_memory = self.memory_manager.replace_memory_space(
                "local",
                Memory(
                    mapping={
                        "int": ((5000, 5999), 0),
                        "float": ((6000, 6999), 0),
                        "t_int": ((7000, 7999), 0),
                        "t_float": ((8000, 8999), 0),
                    }
                ),
            )

            self.symbol_table.add_function(name=func_name, return_type=func_return_type)

            starting_index = self.quack_quadruple.get_current_index()
            self.symbol_table.get_function(func_name).initial_position = starting_index

            for param in func_params:
                self.execute(param)

            for var_decl in func_var_decls:
                self.execute(var_decl)

            self.execute(func_body)

            final_index = self.quack_quadruple.get_current_index()
            self.symbol_table.get_function(func_name).final_position = final_index - 1

            self.quack_quadruple.add_quadruple("endFunc", None, None, func_name)

            self.current_container = self.global_container_name
            self.current_memory_space = "global"
            self.symbol_table.get_function(func_name).clear()

            self.memory_manager.replace_memory_space("local", old_memory)

        elif isinstance(ir, ParamNode):
            param_name = ir.name.name
            param_type = ir.param_type

            address = self.memory_manager.get_first_available_address(
                var_type=param_type,
                space=self.current_memory_space,
```

```
            )
            self.symbol_table.add_parameter(
                name=param_name,
                var_type=param_type,
                containerName=self.current_container,
                address=address,
            )

        elif isinstance(ir, FuncCallNode):
            self.__process_func_call(ir)

        elif isinstance(ir, ReturnNode):
            return_value = self.__evaluate_expression(ir.expresion)
            self.quack_quadruple.add_quadruple("return", self.current_container, None, return_value[0])

        elif isinstance(ir, ProgramNode):
            for decl in ir.global_decls:
                self.execute(decl)

            self.quack_quadruple.push_jump()
            self.quack_quadruple.add_jump(type="goto")

            for func in ir.functions:
                self.execute(func)

            self.quack_quadruple.update_jump(
                index=self.quack_quadruple.pop_jump(), target=self.quack_quadruple.get_current_index()
            )

            self.execute(ir.main_body)

            self.quack_quadruple.add_quadruple("end", None, None, None)

            self.symbol_table.get_function(self.global_container_name).clear()
        else:
            raise UnknownIRTypeError(f"Unknown IR type: {type(ir)}")
```

EXCEPCIONES

Para proporcionar al usuario del lenguaje QuackScript mensajes de error descriptivos, se desarrolló un archivo que contiene todas las clases de excepciones que pueden ser generadas por el programa. Estas excepciones abarcan una variedad de situaciones, incluyendo la redeclaración de variables, la falta de coincidencia de tipos de datos, errores en las operaciones, errores de tiempo de ejecución, errores de consulta, entre otras. A continuación, se presentan las clases de excepciones definidas en este archivo:

```
class InterpreterError(Exception):
    """Base class for all interpreter-related errors."""
    pass

# Redeclaration Errors
class SymbolRedeclarationError(InterpreterError):
    """Raised when a symbol is redeclared in the same scope."""
    pass
class ParameterRedeclarationError(SymbolRedeclarationError):
    """Raised when a function parameter is redeclared."""
    pass
class ContainerRedeclarationError(SymbolRedeclarationError):
    """Raised when a container (e.g., function, module) is redeclared."""
    pass

# Type Mismatch Errors
class TypeMismatchError(InterpreterError):
```

```
    """Raised when there is a type mismatch during assignment or operation."""
    pass

# Operation Errors
class UnsupportedOperationError(InterpreterError):
    """Raised when an operation is not supported between given types."""
    pass
class UnsupportedExpressionError(InterpreterError):
    """Raised when an expression is not supported."""
    pass
class UnknownIRTypeError(InterpreterError):
    """Raised when an unknown IR type is encountered."""
    pass

# Runtime Errors
class DivisionByZeroError(InterpreterError):
    """Raised when attempting to divide by zero."""
    pass
class CannotModifyConstantError(InterpreterError):
    """Raised when trying to update a constant."""
    pass

# Lookup Errors
class NameNotFoundError(InterpreterError):
    """Raised when a variable, function, or container name is not found."""
    pass
class UnknownOperatorError(InterpreterError):
    """Raised when encountering an unknown operator."""
    pass
class ParameterMismatchError(InterpreterError):
    """Raised when number of passed values doesn't match expected parameters."""
    pass
class InvalidParameterIndexError(ParameterMismatchError):
    """Raised when a parameter does not have a valid index."""
    pass
```

MANEJO DE MEMORIA

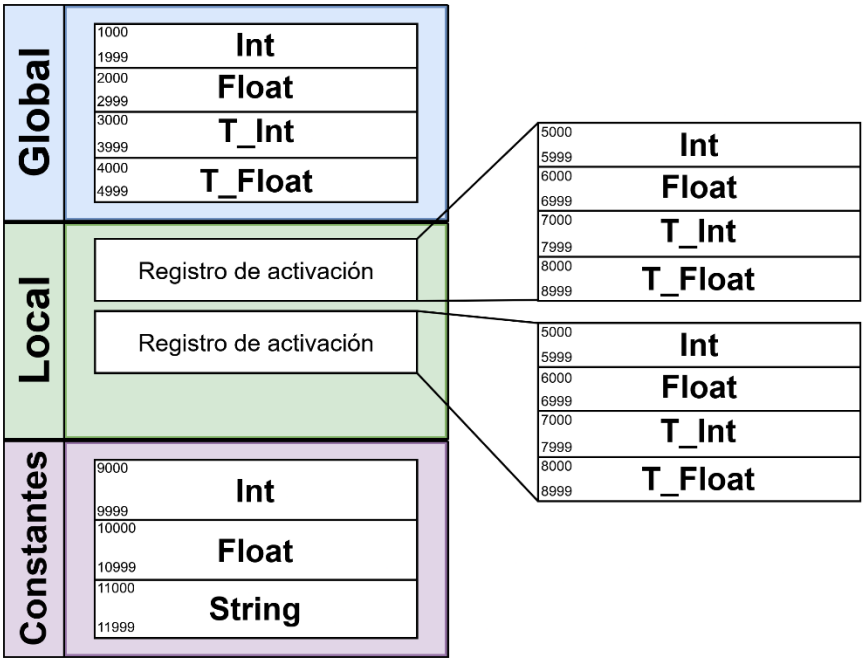
Con el objetivo de simular un espacio de memoria que el programa pueda utilizar para almacenar los valores durante su ejecución, se implementaron dos clases llamadas MemoryManager y Memory. La primera clase define tres espacios de memoria principales: global, local y constantes:

- El espacio global se utiliza para almacenar variables y temporales definidos a nivel global.
- El espacio local contiene las variables y temporales declarados dentro de funciones o bloques específicos.
- Finalmente, el espacio constante almacena valores fijos que no cambian durante la ejecución del programa.

Cada uno de estos espacios está dividido según el tipo de dato que almacena. Para garantizar que cada tipo de dato tenga su propia región de memoria y evitar solapamientos, se establecieron rangos específicos de direcciones de memoria:

- Para el espacio global, los tipos **int** y **float** tienen asignados los rangos (1000, 1999) y (2000, 2999), respectivamente. Además, se incluyen los temporales **t_int** y **t_float** en los rangos (3000, 3999) y (4000, 4999).
- En el caso del espacio local, los rangos son (5000, 5999) para **int**, (6000, 6999) para **float**, (7000, 7999) para **t_int** y (8000, 8999) para **t_float**.

- Por último, en el espacio constante, los tipos **int**, **float** y **string** están mapeados a los rangos (9000, 9999), (10000, 10999) y (11000, 11999), respectivamente.



Estos rangos permiten gestionar eficientemente la asignación de direcciones de memoria y facilitan la identificación del tipo y ámbito de cada variable durante la ejecución del programa.

Asimismo, esta clase de MemoryManager contiene funciones para agregar espacios de memoria (global, local, constantes), que se utiliza cuando se reconstruye la memoria en la máquina virtual, para reemplazar un espacio de memoria con otro provisto, lo cual es útil para gestionar los registros de activación durante las llamadas de funciones, y otras funciones que hacen referencia a métodos de la clase Memory.

Por otro lado, la clase Memory permite representar cada uno de los espacios de memoria, o registros de activación dentro del espacio de memoria local. Para almacenar la memoria se optó por un diccionario de arreglos, donde las llaves son los tipos de datos, y los valores son arreglos. De esta manera, se puede acceder al rango de memoria de un tipo de dato rápidamente, y de ahí, utilizar la posición en memoria y el offset de inicio de cada bloque, para obtener los elementos del arreglo. Es una solución similar a lo que se haría en otro lenguaje de programación con una lista enlazada. Además, como no se tiene una única estructura monolítica, no se está desperdiciando memoria en registros que nunca almacenarán nada: los arreglos permanecen vacíos a menos de que se les asigne un tamaño, o se vayan agregando valores al espacio de memoria. Sobre la parte de asignar memoria, esto sucede durante uno de los procesos de la máquina virtual, que reserva espacios de memoria, preparando los arreglos con el tamaño que ocuparán. Esta clase cuenta con funciones para acceder y guardar en espacios de memoria.

Cuando se encuentra en el proceso de compilación, en vez de almacenar y recuperar valores de los espacios de memoria, solo se reservan los espacios. Esto se hace por medio de la función de “get_first_available_address”:

```
class Memory:
    def get_first_available_address(self, var_type: str) -> int:
        """
```

```
        Returns the first available address for the given var_type, and increments the counter.
        Does not store anything in memory.
        """
        if var_type not in self.next_available:
            raise ValueError(f"No such var_type: {var_type}")
        addr = self.next_available[var_type]
        end_addr = self.memory[var_type]["address_range"][1]
        if addr > end_addr:
            raise MemoryError(f"No available addresses left for type {var_type}.")
        self.next_available[var_type] += 1
        return addr

@dataclass
class MemoryManager:
    def get_first_available_address(self, space: str, var_type: str) -> int:
        """
        Returns the first available address for the given space and var_type, and increments the
counter.
        Does not store anything in memory.
        """
        if space not in self.memory_spaces:
            raise ValueError(f"No such space: {space}")
        return self.memory_spaces[space].get_first_available_address(var_type)
```

Durante el procesesamiento de los cuádruplos en la máquina virtual, es necesario acceder y guardar en los espacios de memoria, para ello se crearon 3 funciones principales en MemoryManager que a su vez llaman a sus contrapartes en la clase Memory dependiendo del espacio de memoria en el que se encuentra. Comenzando con "get_memory" para obtener lo que está almacenado en la dirección de memoria.

```
@dataclass
class Memory:
    def get_memory(
        self,
        var_type: str,
        index: int,
    ):
        """Get the memory for a specific var type."""
        if var_type not in self.memory:
            raise KeyError(f"Memory for type {var_type} not found.")
        start = self.memory[var_type]["address_range"][0]
        return self.memory[var_type]["allocated"][index - start]

@dataclass
class MemoryManager:
    def get_memory(self, index: int) -> Union[int, float, str, bool]:
        """Get the memory for a specific index across all memory spaces."""
        for space_name, memory in self.memory_spaces.items():
            var_type = memory.get_var_type_from_address(index)
            if var_type:
                return memory.get_memory(var_type=var_type, index=index)
        raise ValueError(f"Address {index} not found in any memory space.")
```

También está la función "set_memory" y "add_memory" para guardar valores en direcciones de memoria. La diferencia entre ambas es que "add_memory" agrega al siguiente espacio de memoria disponible el valor, mientras que "set_memory" tiene que recibir una dirección de memoria para guardar en ese lugar el valor. Por lo anterior, "set_memory" se puede utilizar solamente una vez que se hayan reservado espacios. A continuación pongo el fragmento donde están las funciones:

```
@dataclass
class Memory:
```

```
    def set_memory(
        self,
        index: int,
        value: Union[int, float, str, bool],
    ):
        """Set the memory for a specific index, automatically determining var type."""
        for var_type, config in self.memory.items():
            start, end = config["address_range"]
            if start <= index <= end:
                if index - start >= len(self.memory[var_type]["allocated"]):
                    # Extend the allocated list if the index is greater than the current size
                    self.memory[var_type]["allocated"].extend(
                        [None] * (index - start - len(self.memory[var_type]["allocated"]) + 1)
                    )
                self.memory[var_type]["allocated"][index - start] = value
                return
        raise ValueError(f"Address {index} not found in any var type.")

    def add_memory(
        self,
        var_type: str,
        value: Union[int, float, str, bool],
    ):
        """Add a value to the memory for a specific var type in the next available slot."""
        if var_type not in self.memory:
            raise KeyError(f"Memory for type {var_type} not found.")
        start, end = self.memory[var_type]["address_range"]
        allocated = self.memory[var_type]["allocated"]
        for i in range(len(allocated)):
            if allocated[i] is None:
                allocated[i] = value
                return start + i
        if len(allocated) < end - start:
            allocated.append(value)
            return start + len(allocated) - 1
        raise MemoryError(f"No available space in memory for type {var_type}.")

@dataclass
class MemoryManager:
    def add_memory(self, space_name: str, var_type: str, value: Union[int, float, str, bool]):
        """Add a value to the memory for a specific space and var type."""
        if space_name not in self.memory_spaces:
            raise KeyError(f"Memory space {space_name} not found.")
        return self.memory_spaces[space_name].add_memory(var_type=var_type, value=value)

    def set_memory(self, index: int, value: Union[int, float, str, bool]):
        """Set the memory for a specific index across all memory spaces."""
        for space_name, memory in self.memory_spaces.items():
            var_type = memory.get_var_type_from_address(index)
            if var_type:
                memory.set_memory(index=index, value=value)
                return
        raise ValueError(f"Address {index} not found in any memory space.")
```

Asimismo, para hacerlo más modular, existen funciones para agregar los espacios de memoria (global, constantes, local), y para intercambiar un espacio de memoria por otro provisto como argumento. Además, estas funciones permiten pasar como argumento un *mapping* que no solo especifica los tipos de datos y sus rangos en la memoria, sino que también cuánto se tiene que reservar para cada uno. Esto último es opcional, pero es útil para el proceso de la máquina virtual para que se puedan crear espacios reservados.

```
@dataclass
class MemoryManager:
    memory_spaces: Dict[str, Memory]
```

```
    def add_memory_space(self, space_name: str, mapping: Dict[str, Tuple[Tuple[int, int],
Optional[int]]]):
        """
        Add a new memory space with the specified mapping.
        space_name: Name of the memory space (e.g., "global", "local", "constant").
        mapping: dict where keys are var types ("int", "float", "bool", "str")
        and values are the amount to allocate (int) or None (do not allocate).
        """
        self.memory_spaces[space_name] = Memory(mapping=mapping)

    def replace_memory_space(self, space_name: str, new_memory: Memory) -> Memory:
        """Replace an existing memory space with a new Memory object."""
        if space_name not in self.memory_spaces:
            raise KeyError(f"Memory space {space_name} not found.")
        if not isinstance(new_memory, Memory):
            raise TypeError("Provided object is not of type Memory.")
        current_space = self.memory_spaces[space_name]
        self.memory_spaces[space_name] = new_memory
        return current_space
```

A continuación, se muestra el código completo para facilitar ver la integración de las funciones, al igual que demostrar la inicialización de las clases y funciones adicionales:

```
from dataclasses import dataclass
from typing import Dict, Optional, Tuple, Union

@dataclass
class Memory:
    memory: Dict
    next_available: Dict[str, int]

    def __init__(self, mapping: Dict[str, Tuple[Tuple[int, int], Optional[int]]]):
        """
        Initialize memory for specified var types and allocation sizes.
        mapping: dict where keys are var types ("int", "float", "bool", "str")
        and values are the amount to allocate (int) or None (do not allocate).
        """
        self.memory = {}
        self.next_available = {}
        for var_type, config in mapping.items():
            addres_range, amount = config

            if var_type not in self.memory:
                self.memory[var_type] = {}

            self.memory[var_type]["address_range"] = addres_range
            self.next_available[var_type] = addres_range[0]

            if amount is not None:
                if not isinstance(amount, int) or amount < 0:
                    raise ValueError(f"Allocation for {var_type} must be a non-negative integer or
None.")

                self.memory[var_type]["allocated"] = [None] * amount
            else:
                self.memory[var_type]["allocated"] = []

    def get_memory(
        self,
        var_type: str,
        index: int,
    ):
        """Get the memory for a specific var type."""
        if var_type not in self.memory:
```

```
            raise KeyError(f"Memory for type {var_type} not found.")
        start = self.memory[var_type]["address_range"][0]
        return self.memory[var_type]["allocated"][index - start]

    def set_memory(
        self,
        index: int,
        value: Union[int, float, str, bool],
    ):
        """Set the memory for a specific index, automatically determining var type."""
        for var_type, config in self.memory.items():
            start, end = config["address_range"]
            if start <= index <= end:
                if index - start >= len(self.memory[var_type]["allocated"]):
                    # Extend the allocated list if the index is greater than the current size
                    self.memory[var_type]["allocated"].extend(
                        [None] * (index - start - len(self.memory[var_type]["allocated"]) + 1)
                    )
                self.memory[var_type]["allocated"][index - start] = value
                return
        raise ValueError(f"Address {index} not found in any var type.")

    def add_memory(
        self,
        var_type: str,
        value: Union[int, float, str, bool],
    ):
        """Add a value to the memory for a specific var type in the next available slot."""
        if var_type not in self.memory:
            raise KeyError(f"Memory for type {var_type} not found.")
        start, end = self.memory[var_type]["address_range"]
        allocated = self.memory[var_type]["allocated"]
        for i in range(len(allocated)):
            if allocated[i] is None:
                allocated[i] = value
                return start + i
        if len(allocated) < end - start:
            allocated.append(value)
            return start + len(allocated) - 1
        raise MemoryError(f"No available space in memory for type {var_type}.")

    def get_var_type_from_address(self, address: int) -> str:
        """Get the variable type from the address."""
        for var_type, config in self.memory.items():
            start, end = config["address_range"]
            if start <= address <= end:
                return var_type

    def get_first_available_address(self, var_type: str) -> int:
        """
        Returns the first available address for the given var_type, and increments the counter.
        Does not store anything in memory.
        """
        if var_type not in self.next_available:
            raise ValueError(f"No such var_type: {var_type}")
        addr = self.next_available[var_type]
        end_addr = self.memory[var_type]["address_range"][1]
        if addr > end_addr:
            raise MemoryError(f"No available addresses left for type {var_type}.")
        self.next_available[var_type] += 1
        return addr

@dataclass
class MemoryManager:
    memory_spaces: Dict[str, Memory]

    def __init__(self, mappings: Dict[str, Dict[str, Tuple[Tuple[int, int], Optional[int]]]] = None):
        self.memory_spaces = {}
```

```
        if mappings is None:
            mappings = {
                "global": {
                    "int": ((1000, 1999), None),
                    "float": ((2000, 2999), None),
                    "t_int": ((3000, 3999), None),
                    "t_float": ((4000, 4999), None),
                },
                "local": {
                    "int": ((5000, 5999), None),
                    "float": ((6000, 6999), None),
                    "t_int": ((7000, 7999), None),
                    "t_float": ((8000, 8999), None),
                },
                "constant": {
                    "int": ((9000, 9999), None),
                    "float": ((10000, 10999), None),
                    "str": ((11000, 11999), None),
                },
            }
        for space_name, mapping in mappings.items():
            self.add_memory_space(space_name=space_name, mapping=mapping)

    def add_memory_space(self, space_name: str, mapping: Dict[str, Tuple[Tuple[int, int],
Optional[int]]]):
        """
        Add a new memory space with the specified mapping.
        space_name: Name of the memory space (e.g., "global", "local", "constant").
        mapping: dict where keys are var types ("int", "float", "bool", "str")
        and values are the amount to allocate (int) or None (do not allocate).
        """
        self.memory_spaces[space_name] = Memory(mapping=mapping)

    def get_first_available_address(self, space: str, var_type: str) -> int:
        """
        Returns the first available address for the given space and var_type, and increments the
counter.
        Does not store anything in memory.
        """
        if space not in self.memory_spaces:
            raise ValueError(f"No such space: {space}")
        return self.memory_spaces[space].get_first_available_address(var_type)

    def get_memory(self, index: int) -> Union[int, float, str, bool]:
        """Get the memory for a specific index across all memory spaces."""
        for space_name, memory in self.memory_spaces.items():
            var_type = memory.get_var_type_from_address(index)
            if var_type:
                return memory.get_memory(var_type=var_type, index=index)
        raise ValueError(f"Address {index} not found in any memory space.")

    def add_memory(self, space_name: str, var_type: str, value: Union[int, float, str, bool]):
        """Add a value to the memory for a specific space and var type."""
        if space_name not in self.memory_spaces:
            raise KeyError(f"Memory space {space_name} not found.")
        return self.memory_spaces[space_name].add_memory(var_type=var_type, value=value)

    def set_memory(self, index: int, value: Union[int, float, str, bool]):
        """Set the memory for a specific index across all memory spaces."""
        for space_name, memory in self.memory_spaces.items():
            var_type = memory.get_var_type_from_address(index)
            if var_type:
                memory.set_memory(index=index, value=value)
                return
        raise ValueError(f"Address {index} not found in any memory space.")

    def replace_memory_space(self, space_name: str, new_memory: Memory) -> Memory:
        """Replace an existing memory space with a new Memory object."""
```

```
        if space_name not in self.memory_spaces:
            raise KeyError(f"Memory space {space_name} not found.")
        if not isinstance(new_memory, Memory):
            raise TypeError("Provided object is not of type Memory.")
        current_space = self.memory_spaces[space_name]
        self.memory_spaces[space_name] = new_memory
        return current_space

    def get_var_type_from_address(self, address: int) -> str:
        """Get the variable type from the address across all memory spaces."""
        for space_name, memory in self.memory_spaces.items():
            var_type = memory.get_var_type_from_address(address)
            if var_type:
                return var_type
        raise ValueError(f"Address {address} not found in any memory space.")

    def get_str_representation(self) -> str:
        """Return a table-like string representation of the memory manager."""
        lines = []
        for space, memory in self.memory_spaces.items():
            lines.append(f"Memory Space: {space}")
            lines.append(f"{'Type':<10} {'Address Range':<20} {'Allocated':<10} {'Values'}")
            for var_type, info in memory.memory.items():
                addr_range = f"{info['address_range'][0]}-{info['address_range'][1]}"
                allocated = len(info["allocated"])
                values = info["allocated"]
                lines.append(f"{var_type:<10} {addr_range:<20} {allocated:<10} {values}")
            lines.append("")
        return "\n".join(lines)

    def reset_local_temp_memory(self):
        """Reset the temporary memory spaces."""
        if "local" in self.memory_spaces:
            for var_type in self.memory_spaces["local"].memory:
                start = self.memory_spaces["local"].memory[var_type]["address_range"][0]
                self.memory_spaces["local"].next_available[var_type] = start
```

TRADUCCIÓN

COMPILACIÓN DE PROGRAMA

Antes de comenzar el proceso de traducción, es necesario compilar el programa, es decir, realizar todos los pasos que se describieron en las secciones previas. Para esto, se utilizan dos funciones que mandan a llamar a las clases correspondientes y regresan todo lo necesario para continuar con el proceso de traducción:

```
def parse_program(program):
    try:
        # Parse the input program
        tree = quack(program)

        # Transform the parse tree using QuackTransformer
        quack_transformer = QuackTransformer()
        ir = quack_transformer.transform(tree)

        # Get the symbol table from the transformer
        symbol_table = quack_transformer.symbol_table

        # Execute the IR
        # Initialize the memory manager
        memory_manager = MemoryManager()
```

```
        quack_quadruple = QuackQuadruple()
        quack_interpreter = QuackInterpreter(symbol_table, quack_quadruple, memory_manager)
        quack_interpreter.execute(ir)

        return (tree.pretty(), ir, symbol_table, quack_quadruple, memory_manager)
    except UnexpectedInput as e:
        print(f"Parsing failed: {e}")


def compile_program(input_file, output_file):
    """
    Compiles a QuackScript program from an input file and generates an object file.
    """
    try:
        with open(input_file, "r", encoding="utf-8") as file:
            program = file.read()
        tree, ir, symbol_table, quadruples, memory = parse_program(program)
        generate_obj_file(quadruples, symbol_table, output_file)
    except FileNotFoundError:
        print(f"File {input_file} not found.")
    except Exception as e:
        print(f"An error occurred: {e}")
```

GENERACIÓN DE ARCHIVO BINARIO

Para crear el archivo binario .obj, se utiliza la biblioteca *pickle*. En este archivo generado, se incluye la fila de cuádruplos, la referencia de los operadores de los cuádruplos, los datos de las funciones (sin sus tablas de variables que se eliminaron tras su procesamiento), la tabla de constantes y la referencia del nombre del contenedor global. Posteriormente, este archivo será leído por la máquina virtual, que se encargará de traducir el código intermedio (los cuádruplos) y ejecutar las operaciones correspondientes. A continuación, se presenta la función para generar el archivo binario:

```
import pickle

def generate_obj_file(quadruples, symbol_table, output_file):
    """
    Generates a binary object file from the quadruple and symbol table.
    """
    data = {
        "quadruples": quadruples.quadruples,
        "operators": quadruples.operators.operators,
        "functions": symbol_table.containers,
        "constants_table": symbol_table.constants_table,
        "global_container_name": symbol_table.global_container_name,
    }
    with open(output_file, "wb") as f:
        pickle.dump(data, f)
```

LECTURA DEL ARCHIVO BINARIO EN LA MÁQUINA VIRTUAL

Una vez que se genera el archivo binario, se comienza con el proceso de traducción. En este, primero se lee el archivo binario para recuperar todos los datos que se incluyeron y se elimina dado que ya no se necesita ese archivo. Posteriormente, se procede a reconstruir la memoria y finalmente, a procesar los cuádruplos que se generaron en la compilación.

```
def read_and_delete_object_files(self, file_name):
    # print(f"Reading object file: {file_name}")
    with open(file_name, "rb") as f:
        data = pickle.load(f)
    os.remove(file_name)
    return data


def translate_program(self, file_name):
    """
    Translates a QuackScript program from an object file
    """
    if not os.path.exists(file_name):
        print(f"File {file_name} does not exist.")
        return

    data = self.read_and_delete_object_files(file_name)

    self.quadruples = data["quadruples"]
    self.operators = data["operators"]
    self.functions = data["functions"]
    self.constant_table = data["constants_table"]
    self.global_container_name = data["global_container_name"]

    self.reconstruct_memory()

    self.process_quadruples()
```

RECONSTRUCCIÓN DE LA MEMORIA

Ya que se tienen todos los datos del archivo .obj, se puede conocer cuánta memoria se ocupará en el programa, específicamente, para el espacio global y de constantes. Ahorita se deja de lado la memoria necesitada por las funciones ya que eso se considerará una vez que se procese un cuádruplo de llamada de función. Aparte de reservar los espacios de memoria, se almacenan las constantes que se encontraron durante la compilación de una vez. Esto solo se puede hacer para las constantes debido a que el espacio global se llenará durante el procesamiento de los cuádruplos:

```
def reconstruct_memory(self):
    """
    Reconstructs the memory manager and constant table.
    """
    constants_required_space = self.constant_table.required_space
    global_required_space = self.functions[self.global_container_name].required_space

    self.memory_manager = MemoryManager(
        {
            "global": {
                "int": ((1000, 1999), global_required_space.get("int", 0)),
                "float": ((2000, 2999), global_required_space.get("float", 0)),
                "t_int": ((3000, 3999), global_required_space.get("t_int", 0)),
                "t_float": ((4000, 4999), global_required_space.get("t_float", 0)),
            },
            "local": {
                "int": ((5000, 5999), 0),
                "float": ((6000, 6999), 0),
                "t_int": ((7000, 7999), 0),
                "t_float": ((8000, 8999), 0),
            },
            "constant": {
                "int": ((9000, 9999), constants_required_space.get("int", 0)),
                "float": ((10000, 10999), constants_required_space.get("float", 0)),
```

```
                "str": ((11000, 11999), constants_required_space.get("str", 0)),
            },
        }
    )

    # Reconstruct constants
    constants = self.constant_table.constants
    if constants:
        for address, constant in constants.items():
            self.memory_manager.set_memory(index=address, value=constant.value)
```

PROCESAMIENTO DE LOS CUÁDRUPLOS

Ya con todos los datos y la memoria reconstruida, se procede a empezar a iterar sobre los cuádruplos. Para ir iterando se utiliza un índice de la posición actual, la cual en la mayoría de los casos simplemente aumenta 1 posición después de procesar el cuádruplo, excepto para las operaciones de goto, gotoT, gotoF y gosub, en las cuales se reemplaza el índice con la posición del salto para procesar otra sección. En el siguiente ejemplo se aprecia estos saltos del índice con un gotoT. Aquí nada más es relevante aclarar que se está casteando a booleano el argumento debido a que en mi lenguaje utilizo una lógica entera:

```
quadruple = self.quadruples[current_pos]
op, arg1, arg2, result = quadruple

if arg1 is not None and isinstance(arg1, int):
    arg1 = self.memory_manager.get_memory(arg1)
if arg2 is not None and isinstance(arg2, int):
    arg2 = self.memory_manager.get_memory(arg2)

match op:
   case _ if op == op_gotoT:
      if bool(arg1):
        current_pos = result
        continue
```

Y nada más para demostrar el manejo de la lógica entera, en el siguiente fragmento se ve la comparación de argumento 1 es mayor que argumento 2, comparación que luego se castea a entero. Esto resulta en 1 en caso de ser verdadera la comparación, o 0 si es falsa:

```
case _ if op == op_gt:
    result_value = int(arg1 > arg2)
    self.memory_manager.set_memory(index=result, value=result_value)
```

Para el procesamiento de las expresiones simplemente se toman las dos direcciones de memoria de los argumentos, se consulta el espacio de memoria para obtener los valores, y se guarda el resultado en el espacio de memoria especificado en campo de resultado como se puede ver en este ejemplo de una suma:

```
case _ if op == op_add:
    result_value = arg1 + arg2
    self.memory_manager.set_memory(index=result, value=result_value)
```

Otra de las instrucciones claves que se tienen que procesar son las impresiones de pantalla. En este caso, se hace una validación si lo que se va a imprimir es una cadena de caracteres o no, antes de imprimirlos. Se hace esta validación para poder interpretar caracteres de escape que el usuario podría llegar a introducir:

```
case _ if op == op_print:
    value = self.memory_manager.get_memory(result)
    if isinstance(value, str):
        print(value.encode().decode("unicode_escape"), end="")
    else:
        print(value, end="")
```

FUNCIONES

Para el procesamiento de los cuádruplos de las funciones, se necesita hacer unos pasos más que solo leer los dos argumentos y hacer una operación. Esto se debe a que ahora se tiene que introducir el manejo de la memoria local.

Comenzando con la llamada a una función, primero es necesario consultar del directorio de funciones, cuánto espacio se requiere reservar para guardar todas las variables locales y temporales. Una vez con este dato, se crea una nueva memoria local que posteriormente reemplazará a la actual como memoria activa. Por el momento solo se guarda la memoria nueva en una variable aparte para que todavía se puedan usar los valores de la memoria local actual al momento de pasar los argumentos.

```
case _ if op == op_era:
    required_space = self.functions[result].required_space

    new_local_memory = Memory(
        mapping={
            "int": ((5000, 5999), required_space.get("int", 0)),
            "float": ((6000, 6999), required_space.get("float", 0)),
            "t_int": ((7000, 7999), required_space.get("t_int", 0)),
            "t_float": ((8000, 8999), required_space.get("t_float", 0)),
        }
    )
    self.next_local_memory = new_local_memory
```

Posteriormente, si la función lo requiere, se pasan los argumentos. En este caso, se están tratando los parámetros de las funciones como variables locales, por lo que esta acción no es más que asignar un valor en el espacio de memoria asignado. Cabe recalcar que en este caso se están asignando las variables locales en el nuevo espacio de memoria que se generó cuando se recibió la instrucción “era”. Esto se debe a que las variables locales que se declaran cuando se pasan argumentos pertenecen a la función que se va a cargar en la memoria activa posteriormente, y no a la actual. La actual solo se conserva, como ya se mencionó, para poder seguir usando sus valores en los argumentos (por ejemplo, si se pasa como argumento una variable local. Si se reemplazara la memoria con la de la nueva función, esa variable local ya no existiría).

```
case _ if op == op_param:
   self.next_local_memory.set_memory(index=result, value=arg1)
```

Finalmente, cuando se recibe la instrucción de “gosub” se sabe que ya no se pasaran más argumentos y, por ende, ya no es necesario seguir teniendo la memoria local actual cargada. Por ende, se realiza el intercambio de la memoria actual por la nueva que se creó. La memoria actual se irá a una pila de memorias durmiendo para que pueda ser recuperada posteriormente. La decisión de usar una pila se debe a que en caso de tener llamadas recursivas, se quiere que se vayan agregando las memorias dormidas por encima, y que luego, cuando se vayan concluyendo las llamadas recursivas, se vayan sacando una por una de lo más profundo a lo más superficial (que termina siendo lo de más debajo de la pila). Posteriormente, se procede a saltar al inicio de la función. Para esto se usa la posición inicial que se guardó en el directorio de funciones. Sin embargo, antes de hacer el salto, se guarda la posición actual para que, tras finalizar la función, se pueda regresar. Esta posición se guarda en una pila de regresos, para que, en caso de tener llamadas anidadas, se pueda llevar un orden correcto de irlas sacando cuando se concluyan las llamadas.

```
case _ if op == op_gosub:
    prev_local_memory = self.swap_local_memory(self.next_local_memory)
    if prev_local_memory:
        self.sleeping_stack.append(prev_local_memory)

    # Save the next position to return to it later
    go_back_stack.append(current_pos + 1)
    # Set the new position to the function's start
    current_pos = self.functions[result].initial_position
    continue
```

En la declaración de una función, en caso de que la función regrese un valor, se creará un cuádruplo de tipo “return” cuando se encuentre este estatuto. Para procesar este cuádruplo, es necesario consultar el espacio de memoria que se guardó en el directorio de funciones para saber dónde, en la memoria global, se va a guardar el valor que se va a retornar. Una vez que se guarda el valor a retornar, se ajusta la posición actual al índice del cuádruplo de fin de función. Esto se realiza debido a que llegar a un estatuto de “return” debería terminar la ejecución de la función.

```
case _ if op == op_return:
    return_address = self.functions[arg1].return_address
    return_value = self.memory_manager.get_memory(result)
    return_type = self.memory_manager.get_var_type_from_address(result)
    if return_address is not None:
        # If the return address is specified, set the return value in the global memory
        self.memory_manager.set_memory(index=return_address, value=return_value)
    else:
        self.memory_manager.add_memory(space_name="global", var_type=return_type, value=return_value)
    current_pos = self.functions[arg1].final_position
```

El último cuádruplo de la declaración de la función es el que indica que se acabó la función. Primero que nada, se valida que, en caso de que la función debiera regresar un valor, lo regrese. Esto se logra validando que se haya declarado una variable global con el nombre de la función, lo cual solamente se hace cuando se procesa un “return”. Una vez validado esto, se restaura la memoria que estaba durmiendo y se coloca como la memoria local activa. La memoria local con la que se reemplaza se deshecha ya que ya no se utilizará. Finalmente, se actualiza el apuntador que está recorriendo los cuádruplos a la posición que se había guardado en el stack de regreso para continuar desde la posición posterior a la llamada de la función.

```
case _ if op == op_endFunc:
    func_name = result
```

```
    # Validate if the function has a return value
    return_type = self.functions[func_name].return_type
    if return_type in ["int", "float"]:
        return_value = self.memory_manager.get_memory(index=self.functions[func_name].return_address)
        if return_value is None:
            raise ValueError(f"Function {func_name} has no return value set.")

    # Restore the previous local memory
    if self.sleeping_stack:
        previous_local_memory = self.sleeping_stack.pop()
        self.swap_local_memory(previous_local_memory)
    else:
        mem = Memory(
            mapping={
                "int": ((5000, 5999), 0),
                "float": ((6000, 6999), 0),
                "t_int": ((7000, 7999), 0),
                "t_float": ((8000, 8999), 0),
            }
        )
        self.swap_local_memory(mem)

    current_pos = go_back_stack.pop() if go_back_stack else 0
    continue
```

MÁQUINA VIRTUAL COMPLETA

Por último, a continuación, se incluye el programa completo de la máquina virtual para que sea más fácil visualizar las interacciones de todas las funciones:

```
import os
import pickle

from MemoryManager import Memory, MemoryManager

class QuackVirtualMachine:
    """
    A simple virtual machine to execute QuackScript programs.
    It reads object files generated by the QuackCompiler
    and translates them into a format that can be executed.
    """

    def __init__(self):
        """
        Initializes the Quack Virtual Machine.
        """
        # self.symbol_table = None
        self.quadruples = None
        self.memory_manager = None
        self.operators = None
        self.constant_table = None
        self.functions = None
        self.global_container_name = None
        self.sleeping_stack = []
        self.next_local_memory = None

    def read_and_delete_object_files(self, file_name):
        # print(f"Reading object file: {file_name}")
        with open(file_name, "rb") as f:
            data = pickle.load(f)
        os.remove(file_name)
        return data
```

```
    def display_quads(self):
        """
        Displays the quadruples in a readable format.
        """
        ops = {v: k for k, v in self.operators.items()}

        for i, quad in enumerate(self.quadruples):
            op, arg1, arg2, result = quad
            op_str = ops.get(op, op)
            print(f"{i}: ({op_str}, {arg1}, {arg2}, {result})")

    def swap_local_memory(self, local_memory: Memory):
        if "local" in self.memory_manager.memory_spaces:
            previous_local = self.memory_manager.replace_memory_space(space_name="local",
new_memory=local_memory)
            return previous_local
        else:
            (
                self.memory_manager.add_memory_space(
                    space_name="local",
                    mapping={
                        "int": ((5000, 5999), 0),
                        "float": ((6000, 6999), 0),
                        "t_int": ((7000, 7999), 0),
                        "t_float": ((8000, 8999), 0),
                    },
                ),
            )

            return None

    def process_quadruples(self):
        """
        Processes the quadruples and executes them.
        This method should be implemented to handle the execution logic.
        """
        current_pos = 0
        go_back_stack = []
        quadruple = (None, None, None, None)

        # Assign operator values to local variables for match-case
        op_add = self.operators["+"]
        op_sub = self.operators["-"]
        op_mul = self.operators["*"]
        op_div = self.operators["/"]
        op_lt = self.operators["<"]
        op_lte = self.operators["<="]
        op_gt = self.operators[">"]
        op_gte = self.operators[">="]
        op_eq = self.operators["=="]
        op_ne = self.operators["!="]
        op_and = self.operators["and"]
        op_or = self.operators["or"]
        op_goto = self.operators["goto"]
        op_gotoF = self.operators["gotoF"]
        op_gotoT = self.operators["gotoT"]
        op_assign = self.operators["="]
        op_print = self.operators["print"]
        op_era = self.operators["era"]
        op_param = self.operators["param"]
        op_gosub = self.operators["gosub"]
        op_return = self.operators["return"]
        op_endFunc = self.operators["endFunc"]
        op_end = self.operators["end"]

        while quadruple[0] != self.operators["end"]:
            quadruple = self.quadruples[current_pos]
```

```
            op, arg1, arg2, result = quadruple

            if arg1 is not None and isinstance(arg1, int):
                arg1 = self.memory_manager.get_memory(arg1)
            if arg2 is not None and isinstance(arg2, int):
                arg2 = self.memory_manager.get_memory(arg2)

            match op:
                case _ if op == op_add:
                    result_value = arg1 + arg2
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_sub:
                    result_value = arg1 - arg2
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_mul:
                    result_value = arg1 * arg2
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_div:
                    if arg2 == 0:
                        print("Error: Division by zero.")
                        break
                    result_value = arg1 / arg2
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_lt:
                    result_value = int(arg1 < arg2)
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_lte:
                    result_value = int(arg1 <= arg2)
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_gt:
                    result_value = int(arg1 > arg2)
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_gte:
                    result_value = int(arg1 >= arg2)
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_eq:
                    result_value = int(arg1 == arg2)
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_ne:
                    result_value = int(arg1 != arg2)
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_and:
                    result_value = int(arg1 and arg2)
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_or:
                    result_value = int(arg1 or arg2)
                    self.memory_manager.set_memory(index=result, value=result_value)

                case _ if op == op_goto:
                    current_pos = result
                    continue

                case _ if op == op_gotoF:
                    if not bool(arg1):
                        current_pos = result
                        continue
```

```
                case _ if op == op_gotoT:
                    if bool(arg1):
                        current_pos = result
                        continue

                case _ if op == op_assign:
                    self.memory_manager.set_memory(index=result, value=arg1)

                case _ if op == op_print:
                    value = self.memory_manager.get_memory(result)
                    if isinstance(value, str):
                        print(value.encode().decode("unicode_escape"), end="")
                    else:
                        print(value, end="")

                case _ if op == op_era:
                    required_space = self.functions[result].required_space

                    new_local_memory = Memory(
                        mapping={
                            "int": ((5000, 5999), required_space.get("int", 0)),
                            "float": ((6000, 6999), required_space.get("float", 0)),
                            "t_int": ((7000, 7999), required_space.get("t_int", 0)),
                            "t_float": ((8000, 8999), required_space.get("t_float", 0)),
                        }
                    )
                    self.next_local_memory = new_local_memory

                case _ if op == op_param:
                    self.next_local_memory.set_memory(index=result, value=arg1)

                case _ if op == op_gosub:
                    prev_local_memory = self.swap_local_memory(self.next_local_memory)
                    if prev_local_memory:
                        self.sleeping_stack.append(prev_local_memory)

                    # Save the next position to return to it later
                    go_back_stack.append(current_pos + 1)
                    # Set the new position to the function's start
                    current_pos = self.functions[result].initial_position
                    continue

                case _ if op == op_return:
                    return_address = self.functions[arg1].return_address
                    return_value = self.memory_manager.get_memory(result)
                    return_type = self.memory_manager.get_var_type_from_address(result)
                    if return_address is not None:
                        # If the return address is specified, set the return value in the global memory
                        self.memory_manager.set_memory(index=return_address, value=return_value)
                    else:
                        self.memory_manager.add_memory(space_name="global", var_type=return_type,
value=return_value)
                    current_pos = self.functions[arg1].final_position

                case _ if op == op_endFunc:
                    func_name = result

                    # Validate if the function has a return value
                    return_type = self.functions[func_name].return_type
                    if return_type in ["int", "float"]:
                        return_value =
self.memory_manager.get_memory(index=self.functions[func_name].return_address)
                        if return_value is None:
                            raise ValueError(f"Function {func_name} has no return value set.")

                    # Restore the previous local memory
                    if self.sleeping_stack:
                        previous_local_memory = self.sleeping_stack.pop()
```

```
                        self.swap_local_memory(previous_local_memory)
                    else:
                        mem = Memory(
                            mapping={
                                "int": ((5000, 5999), 0),
                                "float": ((6000, 6999), 0),
                                "t_int": ((7000, 7999), 0),
                                "t_float": ((8000, 8999), 0),
                            }
                        )
                        self.swap_local_memory(mem)

                    current_pos = go_back_stack.pop() if go_back_stack else 0
                    continue

                case _ if op == op_end:
                    break

            # Update current index
            current_pos += 1

    def reconstruct_memory(self):
        """
        Reconstructs the memory manager and constant table.
        """
        constants_required_space = self.constant_table.required_space
        global_required_space = self.functions[self.global_container_name].required_space

        self.memory_manager = MemoryManager(
            {
                "global": {
                    "int": ((1000, 1999), global_required_space.get("int", 0)),
                    "float": ((2000, 2999), global_required_space.get("float", 0)),
                    "t_int": ((3000, 3999), global_required_space.get("t_int", 0)),
                    "t_float": ((4000, 4999), global_required_space.get("t_float", 0)),
                },
                "local": {
                    "int": ((5000, 5999), 0),
                    "float": ((6000, 6999), 0),
                    "t_int": ((7000, 7999), 0),
                    "t_float": ((8000, 8999), 0),
                },
                "constant": {
                    "int": ((9000, 9999), constants_required_space.get("int", 0)),
                    "float": ((10000, 10999), constants_required_space.get("float", 0)),
                    "str": ((11000, 11999), constants_required_space.get("str", 0)),
                },
            }
        )

        # Reconstruct constants
        constants = self.constant_table.constants
        if constants:
            for address, constant in constants.items():
                self.memory_manager.set_memory(index=address, value=constant.value)

    def translate_program(self, file_name):
        """
        Translates a QuackScript program from an object file
        """
        if not os.path.exists(file_name):
            print(f"File {file_name} does not exist.")
            return

        data = self.read_and_delete_object_files(file_name)

        self.quadruples = data["quadruples"]
        self.operators = data["operators"]
```

```
        self.functions = data["functions"]
        self.constant_table = data["constants_table"]
        self.global_container_name = data["global_container_name"]

        self.reconstruct_memory()

        self.process_quadruples()
```

PLAN DE PRUEBAS

Se desarrollaron varios casos de prueba para evaluar la funcionalidad de la gramática en su totalidad. Cada caso se centra en examinar un aspecto específico; sin embargo, debido a la naturaleza de los programas, es posible que un mismo caso incluya múltiples aspectos a evaluar.

PRUEBAS UNITARIAS

Las siguientes pruebas se desarrollaron para probar cada una de las reglas gramaticales en profundidad, explorando todas las variantes posibles.

BODY

Se valida la regla:

```
?body: LBRACE RBRACE -> empty_body
     | LBRACE statement+ RBRACE -> body_statements
```

Programa	Salida
// Regla de la gramática a probar: "body" program TestPrint; var num1: int; void display(a: int, b: float, num3: int) [var c: int; var d: float; { }]; main { } end	*<No se imprime nada>*

COMPARISON

Se valida la regla:

```
?comparison: exp
           | exp comparison_op exp -> binary_comparison
```

Programa	Salida
// Regla de la gramática a probar: "comparison" program TestPrint; main { // GT print("10 > 20: ", 10 > 20); // 0 print("\n20 > 10: ", 20 > 10); // 1 // LT print("\n\n10 < 20: ", 10 < 20); // 1 print("\n20 < 10: ", 20 < 10); // 0 // NE print("\n\n10 != 20: ", 10 != 20); // 1 print("\n20 != 20: ", 20 != 20); // 0 // EE print("\n\n10 == 20: ", 10 == 20); // 0 print("\n20 == 20: ", 20 == 20); // 1 // GTE print("\n\n10 >= 20: ", 10 >= 20); // 0 print("\n20 >= 10: ", 20 >= 10); // 1 // LTE print("\n\n10 <= 20: ", 10 <= 20); // 1 print("\n20 <= 10: ", 20 <= 10); // 0 } end	10 > 20: 0 20 > 10: 1 10 < 20: 1 20 < 10: 0 10 != 20: 1 20 != 20: 0 10 == 20: 0 20 == 20: 1 10 >= 20: 0 20 >= 10: 1 10 <= 20: 1 20 <= 10: 0

CONDITION

Se valida la regla:

```
condition: IF LPAREN expresion RPAREN body SEMICOLON -> condition_if
         | IF LPAREN expresion RPAREN body ELSE body SEMICOLON -> condition_if_else
```

Programa	Salida
// Regla de la gramática a probar: "condition" program TestPrint; var n: int = 1; const TOTAL: int = 10; main { while (n <= TOTAL) do { print(n); if (n == TOTAL/2) { print(" - La mitad?"); }; print("\n"); n = n + 1; }; } end	1 2 3 4 5 - La mitad? 6 7 8 9 10

CYCLE

Se valida la regla:

```
cycle: WHILE LPAREN expresion RPAREN DO body SEMICOLON
```

Programa	Salida
// Regla de la gramática a probar: "cycle" program TestPrint; var n: int = 0; main { while (n < 5) do { print(n+1, "\n"); n = n + 1; }; } end	1 2 3 4 5

EXP

Se valida la regla:

```
?exp: term
    | exp PLUS term -> exp_plus
    | exp MINUS term -> exp_minus
```

Programa	Salida
// Regla de la gramática a probar: "exp" program TestPrint; var num1: int = 25; int suma(a: int, b: int) [{ return a+b; }]; main { print("exp_plus: ", num1 + suma(2, 3)); print("\nexp_minus: ", (1E2 / 10) - 25); } end	exp_plus: 30 exp_minus: -15.0

FACTOR

Se valida la regla:

```
factor: id -> factor_id
      | PLUS id -> positive_factor_id
      | MINUS id -> negative_factor_id
```

```
        | cte_num -> factor_cte_num
        | PLUS cte_num -> positive_cte_num
        | MINUS cte_num -> negative_cte_num
        | LPAREN expresion RPAREN -> parenthesis_expresion
        | func_call -> factor_func_call
```

Programa	Salida
// Regla de la gramática a probar: "factor" program TestPrint; var num1: int = 25; int suma(a: int, b: int) [{ return a+b; }]; main { print("factor_id: ", num1); print("\npositive_factor_id: ", +num1); print("\nnegative_factor_id: ", -num1); print("\nfactor_cte_num: ", 25); print("\npositive_cte_num: ", +25); print("\nnegative_cte_num: ", -25); print("\nparenthesis_expresion: ", (2*5+15)); print("\nfactor_func_call: ", suma(10, 15)); } end	factor_id: 25 positive_factor_id: 25 negative_factor_id: -25 factor_cte_num: 25 positive_cte_num: 25 negative_cte_num: -25 parenthesis_expresion: 25 factor_func_call: 25

FUNCTIONS

Se valida la regla:

```
function: return_type id LPAREN RPAREN LBRACKET body RBRACKET SEMICOLON ->
function_no_params_no_var_decl
        | return_type id LPAREN params RPAREN LBRACKET body RBRACKET SEMICOLON -> function_no_var_decl
        | return_type id LPAREN params RPAREN LBRACKET var_decl+ body RBRACKET SEMICOLON ->
function_params_var_decl
        | return_type id LPAREN RPAREN LBRACKET var_decl+ body RBRACKET SEMICOLON -> function_no_params

func_call: id LPAREN RPAREN -> func_call_no_params
         | id LPAREN expresion RPAREN -> func_call_single_param
         | id LPAREN expresion (COMMA expresion)+ RPAREN -> func_call_multiple_params

func_call_stmt: func_call SEMICOLON
```

Programa	Salida
// Regla de la gramática a probar: "functions" program TestPrint; var num1: int = 25; const PI: float = 3.14; void func_0_param() [{ print("Cero parametros!\n"); }]; void func_1_param(a: int) [	Cero parametros! Un parametro: 1 Dos parametros: 1, 2 Sin parametros pero var local: 2 Parametro: 5.0 Var local: 2 Area del circulo: 12.56

```
    {
        print("Un parametro: ", a, "\n");
    }
];

void func_2_param(a: int, b: int) [
    {
        print("Dos parametros: ", a, ", ", b, "\n");
    }
];

void func_param_vars(a: int) [
    var b: int = 2;
    {
        print("Parametro: ", a, "\nVar local: ", b, "\n");
    }
];

void func_no_param_vars() [
    var a: int = 2;
    {
        print("Sin parametros pero var local: ", a, "\n");
    }
];

float get_area(r: float) [

    {
        return r * r * PI;
    }
];

main {
    func_0_param();
    print("\n");
    func_1_param(1);
    print("\n");
    func_2_param(1, 2);
    print("\n");
    func_no_param_vars();
    print("\n");
    func_param_vars(25/5);
    print("\n");
    print("Area del circulo: ", get_area(2.0));
}
end
```

TERM

Se valida la regla:

```
?term: factor
     | term MULT factor -> term_mult
     | term DIV factor -> term_div
```

Programa	Salida
// Regla de la gramática a probar: "term" program TestPrint; var num1: int = 25;	term_mult: 625 term_mult: 30 term_div: 1.0 term_div: 10.0

```
main {
   print("term_mult: ", num1*25);
   print("\nterm_mult: ", (num1 - 10)*2);
   print("\nterm_div: ", 25/num1);
   print("\nterm_div: ", (2E2 + 50) / 25);
}
end
```

VAR DECLARATION

Se valida la regla:

```
const_decl: CONST id COLON var_type ASSIGN expresion SEMICOLON

var_decl: VAR id COLON var_type SEMICOLON -> var_single_decl_no_assign
        | VAR id COLON var_type ASSIGN expresion SEMICOLON -> var_single_decl_assign
        | VAR id (COMMA id)+ COLON var_type SEMICOLON -> var_multi_decl_no_assign
        | VAR id (COMMA id)+ COLON var_type ASSIGN expresion SEMICOLON -> var_multi_decl_assign

assign: id ASSIGN expresion SEMICOLON
```

Programa	Salida
// Regla de la gramática a probar: "var_decl" program TestPrint; var num1: int; var num2: float; var num3: int; const num4: int = 100; const num5: float = 50.50; void display(a: int, b: float, num3: int) [var c: int; var d: float; { a = 1; b = 2.0; c = 3; d = 4.0; num1 = 5; num2 = 6.0; num3 = 7; print("\na:\t\t", a); print("\nb:\t\t", b); print("\nc:\t\t", c); print("\nd:\t\t", d); print("\nnum1 (global):\t", num1); print("\nnum2 (global):\t", num2); print("\nnum3 (local):\t", num3); }]; main { print("Variables globales:"); num1 = 101; num2 = 102.0; num3 = 103; print("\nnum1:\t", num1); print("\nnum2:\t", num2); print("\nnum3:\t", num3);	Variables globales: num1: 101 num2: 102.0 num3: 103 Actualizacion en funcion a: 1 b: 2.0 c: 3 d: 4.0 num1 (global): 5 num2 (global): 6.0 num3 (local): 7 Variables globales nuevamente num1: 5 num2: 6.0 num3: 103 num4 (const): 100 num5 (const): 50.5

print("\n\nActualizacion en funcion"); display(0, 0.0, 0); print("\n\nVariables globales nuevamente"); print("\nnum1:\t", num1); print("\nnum2:\t", num2); print("\nnum3:\t", num3); print("\nnum4 (const):\t", num4); print("\nnum5 (const):\t", num5); } end	

PRUEBAS DE INTEGRACIÓN

Las siguientes pruebas integran las capacidades del lenguaje para validar el funcionamiento en conjunto de las reglas.

ARITMÉTICA BÁSICA

Programa	Salida
// Prueba: Expresiones aritméticas básicas program TestExpr; var x: int = 5; var y: int; var z: int = -3; main { y = ((x + 3) * 2 - 4 / 2) * z; print(y); } end	-42.0

EXPRESIONES LÓGICAS

Programa	Salida
// Prueba: Expresiones lógicas y relacionales program TestLogic; var x: int = 10; var flag: int; main { flag = ((x > 5) and (x < 20)) or (x == 0); print(flag); } end	1

INTEGRACIÓN DE TODAS LAS REGLAS

Programa	Salida
// Programa ejemplo combinando todas las reglas program AllInOne; const PI: float = 3.14159; var radius: float = 2.5; var area: float; var isPositive: int;	Bienvenido al programa All-In-One! Área del círculo:19.6349375 Área calculada:28.274309999999996 isPositive no es cero. No es positivo No es positivo

```
void greet() [
   {
      print("\nBienvenido al programa All-In-One!\n\n");
   }
];

void calculateArea(r: float) [
   var temp_area: float;
   {
      temp_area = PI * r * r;
      print("Área calculada:", temp_area, "\n");
   }
];

void checkSign(value: int) [
   {
      if (value > 0) {
         print(value, " es positivo.\n");
      } else {
         print("No es positivo\n");
      };
   }
];

void countDown(start: int) [
   {
      while (start >= 1) do {
         print(start, "\n");
         start = start - 1;
      };
   }
];

main {
   greet();

   // Asignar y mostrar área
   area = PI * radius * radius;
   print("Área del círculo:", area, "\n");

   // Usar función con parámetro
   calculateArea(3.0);

   // Condicionales
   isPositive = 10;
   if (isPositive != 0) {
      print("isPositive no es cero.\n");
   };

   // Más condicionales anidados
   checkSign(-5);
   checkSign(0);
   checkSign(7);

   // Ciclo while
   countDown(5);

   // Print múltiple
   print("Fin ", "del ", "programa.\n");
}
end
```

```
7 es positivo.
5
4
3
2
1
Fin del programa.
```

FACTORIA

Programa	Salida
// Prueba: Factorial iterativo y recursivo program Factorial; int factorial_iterativo(n: int) [var result, i: int; { if (n < 0) { return -1; }; result = 1; i = 2; while (i <= n) do { result = result * i; i = i + 1; }; return result; }]; int factorial_recursivo(n: int) [{ if (n < 0) { return -1; }; if (n == 0 or n == 1) { return 1; }; if (n > 1) { return n * factorial_recursivo(n - 1); }; }]; main { print("Factorial iterativo: ", factorial_iterativo(5), "\n"); print("Factorial recursivo: ", factorial_recursivo(5), "\n"); } end	Factorial iterativo: 120 Factorial recursivo: 120

FIBONACCI

Programa	Salida
// Prueba: Fibonacci iterativo y recursivo program Fibonacci; void fibonacci_iterativo(n: int) [var a, b, tmp, count: int; { if (n <= 0) { print("Introduce un numero entero positivo!"); return -1; }; a = 0; b = 1;	Fibonacci iterativo:->0->1->1->2->3 Fibonacci recursivo:->0->1->1->2->3

```
    count = 0;

    while (count < n) do {
      print("->", a);
      tmp = a;
      a = b;
      b = tmp + b;
      count = count + 1;
    };

  }
];

int fibonacci_recursivo(n: int) [
  var fib1, fib2: int;
  {
    if (n < 0) {
      return -1;
    };
    if (n == 0) {
      return 0;
    };
    if (n == 1) {
      return 1;
    };
    if (n > 1) {
      return fibonacci_recursivo(n - 1) + fibonacci_recursivo(n - 2);
    };

  }
];

void fibonacci_recursivo_secuencia(n: int) [
 var i: int = 0;
 {
  if (n <= 0) {
    print("Introduce un numero entero positivo!");
  }
  else {
    while (i < n) do {
      print("->", fibonacci_recursivo(i));
      i = i + 1;
    };
  };
 }
];

main {
  print("Fibonacci iterativo:");
  fibonacci_iterativo(5);
  print("\n");
  print("Fibonacci recursivo:");
  fibonacci_recursivo_secuencia(5);
  print("\n");
}
end
```

¿CÓMO CORRER LAS PRUEBAS?

Existen dos tipos de pruebas, las pruebas de solo el léxico y la sintáxis, y las pruebas que integran todo el proceso desde la compilación, hasta la ejecución.

PRUEBAS DE LÉXICO Y SINTÁXIS

Las pruebas se encuentran en el archivo “ParseTests.py” y se pueden correr usando la librería de pytest por medio del comando:

```
> pytest -v Tests.py
```

PRUEBAS DE COMPILACIÓN Y EJECUCIÓN

Para correr todas las prueba, se puede usar el programa “RunAllTests.py” que compila y ejecuta cada una de las pruebas en el directorio tests/ con el comando:

```
> python .\RunAllTests.py
```

COMPILAR Y EJECUTAR UN PROGRAMA DE QUACKSCRIPT

Para poder compilar y ejecutar un programa .quack se puede utilizar el programa “Quackify.py”. Solo es necesario pasarle como argumento en la terminal el programa que se desea compilar y este hará automáticamente el proceso de compilación y ejecución del programa.

A manera de ejemplo, si se tuviera un archivo de programa llamado “programa.quack”, se podría ejecutar de la siguiente manera:

```
> python .\Quackify.py programa.quack
```