

1. Selección de Herramienta de Generación Automática

Análisis de Herramientas Evaluadas

Anteriormente, investigue tres herramientas para elaborar compiladores, para ello, considere las siguientes 3 herramientas:

Flex y Bison

Flex genera escaneres léxicos basados en autómatas finitos deterministas, y Bison crea analizadores de sintaxis a partir de gramáticas libres de contexto

- Conocimiento de C para el código de acciones
- Compilación separada de archivos intermedios
- Manejo de archivos `.l` y `.y` con sintaxis específica
- Proceso de compilación adicional con herramientas de C

Aunque son muy poderosas y eficientes, su integración con Python (el lenguaje seleccionado para este proyecto) requeriría bindings adicionales o comunicación entre procesos, lo cual añade complejidad innecesaria.

Flex y Bison

Flex y Bison son las herramientas más usadas en el desarrollo de compiladores. Flex genera escaneres léxicos basados en autómatas finitos deterministas, y Bison crea analizadores de sintaxis a partir de gramáticas libres de contexto.

Se requiere de conocimiento en C para la definición de las acciones de los puntos neurálgicos, compilación separada de archivos intermedios, manejo de archivos con extensión `.l` y `.y`, y un proceso de compilación adicional utilizando herramientas de C.

Lrparsing

Lrparsing ofrece un enfoque más moderno, usa expresiones de Python para definir gramáticas. Esta herramienta proporciona un parser LR con tokenizador integrado, pre-compilación de gramática para optimizar el rendimiento, y mecanismos de recuperación de errores que permiten continuar el parseo incluso cuando se detectan problemas.

PLY (Python Lex-Yacc)

PLY es una implementación completamente desarrollada en Python sin dependencias externas, lo que facilita su instalación y uso. Usa la sintaxis y filosofía de Lex/Yacc, usa parseo LALR (Look-Ahead LR), tiene validación automática de entrada con reportes de errores, y no requiere archivos de entrada especiales ni compilación separada como

Flex/Bison, implementa cache de tablas de parseo que solo regenera las tablas cuando detecta cambios en la gramatica, lo que optimiza el ciclo de desarrollo.

Decidi utilizar a PLY debido a que cuenta con bastante documentacion y cuenta con muchos ejemplos que facilitan la comprension de el proceso de elaboracion del parser.

Estructura del Compilador

[lex.py](#) - Analizador Lexico

Contiene la definicion de tokens y expresiones regulares para el lenguaje.

Dentro de lex se define a la lista de tokens y a las expresiones del lenguaje, donde para PLY, se definen primero a las palabras reservadas en formato de un diccionario, donde la llave es palabra escrita de la manera en la que sera reconocida en el codigo, y el valor sera nombre del token que esperaríamos. Luego creamos el arreglo de tokens que esperamos, y le concatenamos a los valores de el diccionario de palabras reservadas, despues de esto procedemos a definir las expresiones regulares para cada token, donde primero definimos a las expresiones regulares de los caracteres individuales como variables, luego definimos a las expresiones regulares mas complicadas, como CNT_FLOAT, CNT_INT, ID, y CNT_STRING, espues inclui una regla gramaticar para ignorar espacios y tabs para que el programa no se detenga cuando detecte saltos de linea, luego defini una regla que me de un conteo de lineas, y una regla para definir el manejo de errores del lexer, donde recibo e imprimo el error y luego salto al siguiente token.

yacc.py - Analizador Sintactico

Implementa las reglas gramaticales y construye el Arbol de Sintaxis Abstracta (AST).

Primero defino el simbolo de inicio, este sera la gramatica que empieza el programa, luego defino las producciones de la gramatica, esto lo logro mediante escribir a las gramaticas libres de contexto en forma de docstrings que yacc usa para la formacion de las tablas y definir a p[0] (indice del resultado) como el indice que representa al valor semantico que queremos que represente ese simbolo en el arbol de parseo. Luego agrego una regla para el manejo de errores.

Pruebas

Contiene casos de prueba exhaustivos para validar el funcionamiento del compilador, donde test_lexer.py contiene las pruebas del lexer y test_parse.py contiene las pruebas del parser, el cual emplea al lexer.

Definicion de Expresiones Regulares

Palabras Reservadas

Defini a las palabras reservadas mediante un diccionario que mapea el texto de la palabra a su tipo de token:

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

Tokens Simples

Los operadores y delimitadores se definen mediante expresiones regulares sencillas que se asignan a variables:

Token	Expresion Regular	Descripcion
COMMA	,	Coma
COLON	:	Dos puntos
SEMI_COLON	;	Punto y coma
L_CBRACKET	\{	Llave izquierda
R_CBRACKET	\}	Llave derecha
L_SBRACKET	\[Corchete izquierdo
R_SBRACKET	\]	Corchete derecho
L_PARENTHESIS	\(Parentesis izquierdo
R_PARENTHESIS	\)	Parentesis derecho
EQUAL	=	Igual
PLUS	\+	Suma
MINUS	-	Resta
ASTERISK	*	Multiplicacion

FORWARD_SLASH	/	Division
NOT_EQUAL	!=	Diferente de
GREATER_THAN	>	Mayor que
LOWER_THAN	<	Menor que

Tokens Complejos

Los tokens con patrones complejos se definen mediante funciones:

CNT_FLOAT

```
def t_CNT_FLOAT(t):
    r'\d+\.\d+(e[+-]?\d+)?'
    t.value = float(t.value)
    return t
```

Patron: Uno o mas digitos, seguido de punto decimal, uno o mas digitos, opcionalmente seguido de notacion cientifica (e/E seguido de signo opcional y digitos).

Ejemplos validos: 3.14, 2.5e10, 1.0e-5

CNT_INT

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

Patron: Uno o mas digitos.

Ejemplos validos: 0, 42, 1000

CNT_STRING

```
def t_CNT_STRING(t):
    r'"([^"\\]|\\.)*"'
    t.value = t.value[1:-1] # Remover comillas
    return t
```

Patron: Comillas dobles, seguidas de cualquier caracter excepto comillas o backslash (o secuencias de escape), terminando en comillas dobles.

Ejemplos validos: "hola", "valor: 10", "con \"escape\""

ID

```
def t_ID(t):  
    r'[a-zA-Z_]\w*'  
    t.type = reserved_namespace.get(t.value, 'ID')  
    return t
```

Patron: Letra o guion bajo, seguido de cero o mas caracteres alfanumericos o guiones bajos.

Ejemplos validos: x, variable1, _temp, miVariable

Manejo de Espacios y Comentarios

```
# Ignorar espacios y tabs  
t_ignore = ' \t'  
  
# Contar lineas  
def t_newline(t):  
    r'\n+'  
    t.lexer.lineno += len(t.value)
```

Importante: PLY evalua las reglas revizando primero a las funciones definidas antes que otras funciones, luego a las expresiones regulares mas largas antes que a las cortas, y luego al orden de definicion en el codigo. Por eso, t_CNT_FLOAT debe definirse antes que t_CNT_INT, ya que 3.14 podria ser interpretado incorrectamente como 3, ., 14 y daria un error por el punto.

Reglas Gramaticales

Estructura del Programa

Primero se define a la gramatica que empieza al programa, esta es program

Implementacion

```
def p_program(p):  
    """program : PROGRAM ID SEMI_COLON vars func_list MAIN body END  
                | PROGRAM ID SEMI_COLON func_list MAIN body END  
                | PROGRAM ID SEMI_COLON vars MAIN body END  
                | PROGRAM ID SEMI_COLON MAIN body END"""
```

```

if len(p) == 9:
    p[0] = ('program', p[2], p[4], p[5], p[7])
elif len(p) == 8:
    if isinstance(p[4], tuple) and p[4][0] == 'vars':
        p[0] = ('program', p[2], p[4], [], p[6])
    elif isinstance(p[4], list):
        p[0] = ('program', p[2], None, p[4], p[6])
    else:
        p[0] = ('program', p[2], None, [], p[5])
else:
    p[0] = ('program', p[2], None, [], p[5])

```

Variables

```

<VARS> → var <var_list>
<var_list> → <id_list> : <type> ; <var_list'>
<var_list'> → <var_list> | ε
<id_list> → id <id_list'>
<id_list'> → , <id_list> | ε
<type> → int | float

```

Permite declarar multiples variables del mismo tipo y declaraciones consecuticas.

Expresiones Aritmeticas

```

<EXP> → <EXP> + <TERMINO>
      | <EXP> - <TERMINO>
      | <TERMINO>

<TERMINO> → <TERMINO> * <FACTOR>
          | <TERMINO> / <FACTOR>
          | <FACTOR>

<FACTOR> → ( <EXPRESION> )
          | + id | + <CTE>
          | - id | - <CTE>
          | id
          | <CTE>

<CTE> → cte_int | cte_float

```

Precedencia de operadores

1. +, − (suma, resta)
2. *, / (multiplicacion, division)
3. Operadores unarios +, −
4. Parentesis ()

Expresiones

```
<EXPRESION> → <EXP> <EXPRESION'>  
<EXPRESION'> → > <EXP>  
                | < <EXP>  
                | != <EXP>  
                | ε
```

Condiciones y ciclos

Condicional

```
<CONDITION> → if ( <EXPRESION> ) <Body> ;  
              | if ( <EXPRESION> ) <Body> else <Body> ;
```

Ciclo

```
<CYCLE> → while ( <EXPRESION> ) do <Body> ;
```

Funciones

```
<FUNCS> → void id ( <param_list> ) [ [<VAR>] <Body> ] ; <func_list'  
<func_list'> → <FUNCS> | ε  
  
<param_list> → id : <type> <param_list'>  
<param_list'> → , <param_list> | ε
```

Se permiten definiciones de variables locales

Statements

```
<Body> → { <statement_list> }  
        | { }
```

```

<statement_list> → <statement> <statement_list'>
<statement_list'> → <statement_list> | ε

<statement> → <assign>
              | <condition>
              | <cycle>
              | <f_call>
              | <print_stmt>

<assign> → id = <EXPRESSION> ;

<f_call> → id ( [<expression_list>] ) ;

<print_stmt> → print ( <print_expression_list> ) ;
<print_expression_list> → (<expression> | cte_string) <print_list'>
<print_list'> → , <print_expression_list> | ε

```

Plan de Pruebas y Casos de Prueba

Analisis Lexivo

Se creo un archivo de python donde se incluyo un string y un print que arroja a los tokens detectados, la linea donde se encontro y su valor, corrio de manera exitosa y sin errores, todos los tokens fueron detectadis correctamente.

Casos de Prueba del Analisis Sintactico

Test 1: Programa Minimo

Se debe de verificar la estructura basica de un programa.

Entrada:

```

program test1;
main {
}
end

```

AST Esperado:

```

('program', 'test1', None, [], ('body', []))

```

Test 2: Programa con Variables

Se debe de validar declaracion de variables globales.

Entrada:

```
program test2;
var
    x : int;
    y, z : float;
main {
}
end
```

AST Esperado:

```
( 'program', 'test2',
  ( 'vars', [
    ( 'var_declaration', ['x'], 'int'),
    ( 'var_declaration', ['y', 'z'], 'float')
  ]),
  [],
  ( 'body', [] )
)
```

Test 3: Expresiones Aritmeticas

Se debe de verificar precedencia y asociatividad de operadores.

Entrada:

```
program test3;
var
    x, y, z : int;
main {
    x = 5 + 3;
    y = x * 2;
    z = (x + y) / 2;
}
end
```

AST Esperado (fragmento de z):

```
( 'assign', 'z',  
  ('/', ('+', 'x', 'y'), 2)  
)
```

Test 4: Condicional If Simple

Se debe de validar estructura if sin else.

Entrada:

```
program test4;  
var  
    x : int;  
main {  
    x = 10;  
    if (x > 5) {  
        x = x + 1;  
    };  
}  
end
```

AST Esperado:

```
( 'if',  
  ('>', 'x', 5),          # condicion  
  ('body', [              # then  
    ('assign', 'x', ('+', 'x', 1))  
  ]),  
  None                   # sin else  
)
```

Test 5: Condicional If-Else

Se debe de validar estructura if-else completa.

Entrada:

```
program test5;  
var  
    x : int;  
main {  
    x = 3;
```

```
    if (x > 5) {
        x = 10;
    } else {
        x = 0;
    };
}
end
```

Test 6: Ciclo While

Se debe de verificar estructura de ciclos.

Entrada:

```
program test6;
var
    x : int;
main {
    x = 0;
    while (x < 10) do {
        x = x + 1;
    };
}
end
```

AST Esperado:

```
( 'while',
  (<, 'x', 10),           # condicion
  ('body', [              # cuerpo del ciclo
    ('assign', 'x', ('+', 'x', 1))
  ])
)
```

Test 7: Funcion Print

Se debe de validar impresion de expresiones y strings.

Entrada:

```
program test7;
var
    x : int;
```

```

main {
    x = 42;
    print(x);
    print("El valor es:", x);
}
end

```

AST Esperado:

```

('print', ['x'])
('print', ['El valor es:', 'x'])

```

Test 8: Declaracion de Funciones

Se debe de verificar sintaxis de funciones con parametros y variables locales.

Entrada:

```

program test8;
void myFunc(a : int, b : int) [
    var
        result : int;
    {
        result = a + b;
    }
];
main {
}
end

```

AST Esperado:

```

('func', 'myFunc',
 [(['a', 'int'), ('b', 'int')], # parametros
 ('vars', [                          # variables locales
    ('var_declaration', ['result'], 'int')
 ]),
 ('body', [                          # cuerpo
    ('assign', 'result', ('+', 'a', 'b'))
 ]))
)

```

Test 9: Programa Completo

Se debe de validar integracion de todas las características del lenguaje.

Entrada:

```
program completo;
var
    x, y : int;
    z : float;

void calcular(a : int, b : int) [
    var
        resultado : int;
    {
        resultado = a + b;
    }
];

main {
    x = 10;
    y = 20;
    z = 3.14;

    if (x < y) {
        print("x es menor que y");
    } else {
        print("x es mayor o igual que y");
    };

    while (x < 100) do {
        x = x + 1;
    };

    calcular(x, y);
}
end
```

Casos de Prueba de Errores Sintacticos

Error 1: Falta Punto y Coma

Entrada:

```
program test1
var
    x : int;
main {
}
end
```

Salida:

Syntax error at token VAR ('var') on line 2

Error 2: Falta Dos Puntos en Declaracion

Entrada:

```
program test2;
var
    x int;
main {
}
end
```

Salida:

Syntax error at token INT ('int') on line 3

Error 3: Falta Punto y Coma en Asignacion

Entrada:

```
program test3;
var
    x : int;
main {
    x = 5
}
end
```

Salida:

Syntax error at token R_CBRACKET ('}') on line 6

Error 4: Parentesis Sin Cerrar

Entrada:

```
program test4;  
var  
    x : int;  
main {  
    if (x > 5 {  
        x = 10;  
    };  
}  
end
```

Salida:

Syntax error at token L_CBRACKET ('{') on line 5

Error 5: Falta Palabra Clave 'end'

Entrada:

```
program test5;  
main {  
    x = 5;  
}
```

Salida:

Syntax error: Unexpected end of file (EOF)

Analisis semantico

semantic_cube.py - Cubo semantico

El cubo semantico es una estructura de datos que define las reglas de compatibilidad de tipos para todas las operaciones del lenguaje. Proporciona validacion de operaciones binarias mediante consultas de tipos.

En mi caso lo implemente con un diccionario de tres niveles donde se consulta mediante la estructura `cubo[operador][tipo_izquierdo][tipo_derecho]` para obtener el tipo resultante, los tipos soportados son `int` y `float`, y el resultado puede ser uno de estos tipos o `None` si la operacion es invalida.

Operadores y Reglas de Tipo

Los operadores se organizan en cuatro categorias: aritmeticos (suma, resta, multiplicacion), division especial, relacionales, y asignacion.

Para operadores aritmeticos de suma, resta y multiplicacion, las reglas son que `int` operado con `int` produce `int`, `int` con `float` produce `float`, `float` con `int` produce `float`, y `float` con `float` produce `float`. Esto nos permite realizar operaciones entre `ints` y `floats`, y realizar conversiones de `int` a `float` de ser necesario, para la division, todas las combinaciones producen `float`, los operadores relacionales mayor que, menor que y diferente a, todas las combinaciones de operandos producen `int` que representa un valor booleano (0 u 1), la asignacion, valida compatibilidad permitiendo asignacion de `int` a `float` con promocion implicita, pero `int` no puede recibir `float` directamente.

Metodos Principales

El cubo semantico proporciona dos metodos principales:

- El metodo `get_result_type` recibe un operador y dos tipos de operandos, y arroja el tipo resultante o `None` si la operacion es invalida
- El metodo `is_valid_operation` verifica si una operacion es valida sin la necesidad de consultar el tipo especifico

Analizador semantico

El analizador semantico usa al cubo semantico con las estructuras de datos para realizar validacion semantica del programa durante el parseo. Contiene referencias al directorio de funciones, la funcion actual siendo analizada, y una lista con los errores encontrados.

Helpers

El metodo `get_literal_type` determina si un valor es entero o flotante basandose en el tipo de Python, el metodo `print_symbol_tables` imprime todas las tablas de simbolos tanto globales como de cada funcion para debugging, por ultimo, el metodo `reset` reinicia el analizador para evitar interferencias con programas anteriores.

Estructuras de Datos para Analisis semantico

FunctionDirectory

Lo implemente usando un diccionario ya que permite acceso rapido a las funciones por nombre y acceso directo a la tabla global, el diccionario se compone de funciones mapeadas por nombre, una referencia a la funcion siendo analizada, y una tabla de variables globales.

Las operaciones principales son `add_function` para agregar una funcion verificando no este duplicada, `get_function` para obtener una funcion por nombre, `function_exists` para verificacion rapida, `set_current_function` para cambiar el ambito actual, `add_global_variable` y `add_local_variable` para declarar variables en el ambito apropiado, y `lookup_variable` para buscar variables implementando por scope, verificando primero a las locales y luego a las globales.

VariableTable

La implemente con un diccionario debio a que permite busquedas or nombre rapidas, insercion rapida e iteracion eficiente, este diccionario mapea nombres de variables a objetos Variable.

Las operaciones principales son `add_variable` que agrega una variable verificando no exista dos veces, `get_variable` que busca una variable por nombre, `variable_exists` para verificacion rapida, y `get_all_variables` para obtener la lista completa cuando se necesita.

Variable

Contiene el nombre de la variable, su tipo, y el scope donde fue declarada

Function

Contiene el nombre de la funcion, su tipo de retorno , una lista de parametros con sus tipos, una tabla de variables para las variables locales, y un campo `start_address` para generacion de codigo.

Las funciones principales son `add_parameter` que agrega un parametro como variable local, `add_local_variable` para agregar variables locales, `get_variable` para busqueda de variables locales, y `get_parameter_types` para obtener la lista de tipos de parametros en orden.

Integracion de Analisis semantico

El analisis semantico se integra en el parser [yacc.py](#) donde cada regla gramatical invoca los puntos neuralgicos del analizador semantico, cuando el parser reconoce un token o completa una regla, ejecuta el punto neuralgico correspondiente que valida la semantica y actualiza las estructuras de datos, por ejemplo, cuando se parsea una declaracion de variable, [yacc.py](#) llama a `np_declare_variable` para validar y registrar la variable.

El analizador junta todos los errores encontrados sin parar el parseo, lo que permite mostrar multiples errores en una sola ejecucion, lo que facilita el debugging.

Las tablas de simbolos se construyen durante el analisis, cuando se termina el parseo completo, se puede consultar el directorio de funciones para obtener informacion sobre todas las funciones y variables del programa, asi como la lista de errores encontrados.

Generacion de Cuadрупlos

Genere una clase llamada IntermediateCodeGenerator con el fin de separar la logica de la generacion de cuadрупlos, dentro de esta clase inclui un singleton con el fin de poder utilizar al mismo objeto de manera global, la clase se inicializa de la siguiente manera:

```
class IntermediateCodeGenerator:
    def __init__(self):
        self.quads = []

        # Pilas
        self.operator_stack = []
        self.operand_stack = []

        self.type_stack = []

        # Aqui guardamos el salto pendiente previo a evaluar una cond
        self.jump_stack = []

        self.temp_var_counter = 0 # Contador de variables temporales
```

Donde se inicializan las pias para los cuadрупlos, operadores, operandos, tipos y saltos (cuadрупlos que tienen que ser completados despues de que se define el resultado de la condicion del if o while).

Es importante aclarar que las variables temporales no cuentan con una pila propia ya que estas se almacenan dentro de la pila de operandos y su tipo se almacena dentro de la pila de operadores (una variable temporal representa el resultado de una operacion, este tipo es asignado por el cubo semantico), por ello, tenemos un contador que facilita la generacion de las mismas.

La clase cuenta con metodos que permiten realizar acciones CRUD con los atributos de la clase:

```
# Agregar un operador a la pila
def push_operator(self, operator):
    self.operator_stack.append(operator)
```

```

# Eliminamos uno de los operadores de la fila de operadores, pri
# si existe, eliminamos el elemento de la pila y lo arrojamos
# si no existe, arrojamos un nulo
def pop_operator(self):
    if self.operator_stack:
        return self.operator_stack.pop()

    return None

# Retornamos el elemento mas nuevo de la pila, usamos el indice
def peak_operator(self):
    if self.operator_stack:
        return self.operator_stack[-1]
    return None

def push_jump(self, position):
    self.jump_stack.append(position)

def pop_jump(self):
    if self.jump_stack:
        return self.jump_stack.pop()

    return None

def add_temp(self):
    self.temp_var_counter += 1
    return f"t{self.temp_var_counter}"

def add_quad(self, operator, arg1, arg2, result):
    quad = (operator, arg1, arg2, result)
    self.quads.append(quad)

    # Indice en el cual se almaceno el cuadruplo agregado
    return len(self.quads) - 1

```

Separare algunos metodos que tienen un comportamiento distinto

```

def push_operand(self, operand, operand_type):
    self.operand_stack.append(operand)
    self.type_stack.append(operand_type)

def pop_operand(self):
    if self.operand_stack and self.type_stack:

```

```
        operand = self.operand_stack.pop()
        operand_type = self.type_stack.pop()

    return operand, operand_type

return None, None
```

Las operaciones realizadas en la pila de operandos se aplican al mismo tiempo en la pila de tipos, esto debido a que un operador va ligado a su tipo, esto nos es útil particularmente al momento de generar cuádruplos para las operaciones aritméticas, ya que es necesario evaluar el tipo de dato que tendrá la variable temporal resultante después de evaluar una expresión.

Tenemos las siguientes estructuras:

Pila de Operadores - operator_stack

Almacena operadores binarios (+, -, *, /) durante el análisis de expresiones aritméticas. Se empuja un operador cuando se encuentra en la expresión y se saca cuando debe generarse un cuádruplo para una expresión aritmética. Permite manejar la precedencia y asociatividad de operadores.

Pila de Operandos - operand_stack

Almacena operandos (variables, constantes, resultados temporales) de las expresiones. Se empuja cada operando antes de procesarlo y se sacan dos operandos cuando se genera un cuádruplo aritmético, funciona en paralelo con la pila de tipos.

Pila de Tipos - type_stack

Mantiene sincronizado el tipo de dato de cada operando en la pila de operandos. Se empuja y se popa junto con cada operando para validar operaciones con el cubo semántico y determinar el tipo resultante de variables temporales.

Pila de Saltos - jump_stack

Almacena posiciones de cuádruplos GOTO y GOTO pendientes de llenar, se empuja la posición cuando se genera un salto condicional y se saca cuando se conoce el destino final, sirve para implementar estructuras de control (if-else, while).

Fila de Cuádruplos - quads

Almacena secuencialmente todos los cuádruplos generados como tuplas (operador, arg1, arg2, resultado), representa el código intermedio que será ejecutado. Se agrega con `add_quad()` que retorna el índice, permitiendo llenar cuádruplos pendientes por los saltos usando `fill_quad()`.

Memoria de Ejecucion

Se realizo una memoria de ejecucion la cual se divide en las siguientes partes, primero tenemos a las direcciones de memoria, utilice dos diccionarios para manejarlas, un contador para incrementar las direcciones de memoria, cuyos valores incremento conforme asigno direcciones de memoria, y un diccionario cuyos valores son la el indice maximo que puede alcanzar cada variable.

```
# Contadores para asignar a la siguiente direccion disponible
self.memory_counters = {
    'global_int' : 1000,
    'global_float' : 2000,
    'local_int' : 3000,
    'local_float' : 4000,
    'temp_int' : 5000,
    'temp_float' : 6000,
    'const_int' : 7000,
    'const_float' : 8000,
    'const_string' : 9000
}

# Diccionario con los limites superiores para cada grupo
self.assignment_limits = {
    'global_int' : 1999,
    'global_float' : 2999,
    'local_int' : 3999,
    'local_float' : 4999,
    'temp_int' : 5999,
    'temp_float' : 6999,
    'const_int' : 7999,
    'const_float' : 8999,
    'const_string' : 9999
}
```

Luego defini otros tres diccionarios, debug_memory para traducir a las direcciones de memoria a valores "humanos", var_dict sirve para usarse para lookup ya que el programa consulta a la memoria constantemente y el diccionario tiene una complejidad de tiempo lineal $O(1)$, por ultimo tengo un diccionario de constantes const_dict el cual sirve oara evitar la duplicacion de constantes.

```
# Definimos un mapa donde almacenaremos a la direccion de memoria
self.debug_memory = {}
```

```

# Mapa para almacenar variables usando su nombre como llave
# Este diccionario nos sirve para tener un lookup rapido de
# El diccionario tiene complejidad lineal para busqueda, lo
# por lo que la velocidad de ejecucion de nuestro programa c
self.var_dict = {}

# Mantenemos un diccionario de constantes para evitar la dup
# Utilizaremos una tupla como llave, donde guardaremos al va
# como 5 y 5.0, donde 5 es un entero y 5.0 es un float, por
self.const_dict = {}

```

Por ultimo esta `assign_address`, que sirve para asignar y almacenar a las direcciones de memoria, donde se forma la llave para los diccionarios de variables usando al scope y al tipo de variable, luego usamos a esa llave para acceder al contador y compararlo con la cantidad limite de direcciones, si aun hay direcciones, asignamos la siguiente direccion disponible a una variable (el valor actual para la llave que se esta usando) y se le suma una unidad al contador, si se provee un id, se guarda en el diccionario de debug con su direccion de memoria como llave, luego verificamos a los scopes para saber si el valor que almacenamos es una variable, de serlo, guardamos a la direccion en el diccionario de variables con el id como llave.

```

def assign_address(self, scope, var_type, id=None):
    # Revizamos si aun tenemos espacio disponible en la memoria

    key = f"{scope}_{var_type}"

    if self.memory_counters[key] > self.assignment_limits[key]:
        print(f"Memoria excedida para {key}")
        return -1

    # Asignamos una direccion de memoria dependiendo del tipo de
    address = self.memory_counters[key]
    self.memory_counters[key] += 1

    if id != None:
        self.debug_memory[address] = id

        if scope == "global" or scope == "local":
            self.var_dict[id] = address

    return address

```

add_temp() es una funcion auxiliar que se utiliza para asignarles direcciones de memoria a las variables temporales desde el parser, este metodo puede cambiar, ya que aun estoy decidiendo si gestionar a las variables temporales desde la memoria de ejecucion.

```
def add_temp(self, var_type, name):  
    return self.assign_address(scope='temp', var_type=var_type,
```

add_variable utiliza a la funcion assign_address para asignar una direccion de memoria a una variable, primero revisa que la variable no haya sido declarada anteriormente en var_dict, este metodo se usa principalmente en la funcion add_variable de variable table.

```
def add_variable(self, var_name, var_type, scope="global"):  
    if var_name in self.var_dict:  
        print(f"La variable {var_name} ya fue declarada")  
        return  
  
    address = self.assign_address(scope=scope, var_type=var_type)  
  
    return address
```

Puntos Neuralgicos

1. push_operand(): Agregar un operando y su tipo a las pilas de operandos y tipos

```
def push_operand(self, operand, operand_type):  
    self.operand_stack.append(operand)  
    self.type_stack.append(operand_type)
```

2. push_operator(): Agregar operador a pila de operadores

```
# Agregar un operador a la pila  
def push_operator(self, operator):  
    self.operator_stack.append(operator)
```

3. generate_arithmetic_operation(): Generar cuadruplo aritmetico, utiliza al cubo semantico para validar a los tipos de las temporales resultantes

```
# Cuadruplo para operaciones aritmeticas, se utiliza al cubo semantico  
def generate_arithmetic_operation(self, semantic_cube: SemanticCube):  
    if not self.operator_stack:  
        return None
```

```

operator = self.pop_operator()

# El procesamiento se lleva a cabo de izquierda a derecha, p
# Teniendo 'a + b', tenemos que el proceso seria:
# operand_stack.append(a)
# type_stack.append(int)

# operator_stack.append(+)

# operand_stack.append(b)
# type_stack.append(int)

# Por ende, podemos asumir que el operando de la derecha es r
right, right_type = self.pop_operand()
left, left_type = self.pop_operand()

# Usamos al cubo semantico para validar el tipo del resulrac
result_type = semantic_cube.get_result_type(operator=operator

if result_type is None:
    print(f"Error: Operacion invalida {left_type} {operator}")
    return None

# Generamos una nueva variable temporal para almacenar el re
temp = self.add_temp()

# Creamos un cuadruplo
self.add_quad(operator=operator, arg1=left, arg2=right, resu

# Guardamos el resultado en la pila de operandos
self.push_operand(operand= temp, operand_type=result_type)

return temp

```

4. generate_assignment(): Generar cuadruplo de asignacion

```

# No todas las operaciones necesitan a los 4 argumentos ya que r
# En ocaciones los resultados dentro de un cuadruplo pueden repr
def generate_assignment(self, var_name):
    exp_operand, exp_type = self.pop_operand()
    self.add_quad(operator="=", arg1=exp_operand, arg2=None, res
    #
        asignacion    valor a asignar

```

5. `begin_if()`: Iniciar estructura if con su salto correspondiente

```
# Generamos un GOTOF (Go To False) al inicio del if y marcamos u  
# la cual sera llenada una vez que sepamos como se evalua la con  
def begin_if(self):  
    condition, condition_type = self.pop_operand()  
  
    # Evaluamos el cuadruplo con -1 para señalarlo como un salto  
    # indicando asi que sera evaluado una vez que se evalua la c  
    pending = self.add_quad("GOTOF", condition, None, -1)  
  
    self.push_jump(pending)
```

6. `end_if()`: Completa el salto del if

```
# Completamos el salto pendiente, para entonces ya sabemos a que  
def end_if(self):  
    pending = self.pop_jump()  
    current_position = self.get_current_position()  
  
    # Llenamos el salto pendiente con el cuadruplo evaluado  
    self.fill_quad(position=pending, value=current_position)
```

7. `begin_else()`: Iniciar estructura else con GOTO y completa al GOTOF de si if predecesor

```
def begin_else(self):  
    # Generamos un GOTO (Go To) para saltar al else  
    # Aun no sabemos donde terminara el else  
    goto_pos = self.add_quad("GOTO", None, None, -1)  
  
    # Completamos el GOTOF del if de este else, para entonces sa  
    # si la condicion del if evalua a falso  
    gotof_pos = self.pop_jump()  
    current_pos = self.get_current_position()  
    self.fill_quad(position=gotof_pos, value=current_pos)  
  
    # Guardamos el GOTO pendiente en la cola de saltos  
    self.push_jump(goto_pos)
```

8. `end_else()`: Completa el salto del else

```

# Completamos el cuadruplo pendiente de else
def end_else(self):
    goto_pos = self.pop_jump()
    current_pos = self.get_current_position()
    self.fill_quad(position=goto_pos ,value=current_pos)

```

9. begin_while(): Guarda la posición donde empieza el while para reevaluar la condición del while

```

def begin_while(self):
    current_pos = self.get_current_position()
    self.push_jump(position=current_pos)

```

10. generate_while_condition(): Generar cuadruplo GOTOF para la condición del while

```

# Generamos un GOTOF para el while, esto representará la condición
def generate_while_condition(self):
    condition, cond_type = self.pop_operand()

    gotof_pos = self.add_quad(operator="GOTOF", arg1=condition,
    self.push_jump(position=gotof_pos)

```

11. end_while(): Generar GOTO de retorno y completar salto del while

```

def end_while(self):
    gotof_pos = self.pop_jump()
    return_pos = self.pop_jump()

    self.add_quad(operator="GOTO", arg1=None, arg2=None, result=r

    current_position = self.get_current_position()
    self.fill_quad(position=gotof_pos, value=current_position)

```

12. generate_print(): Generar cuadruplos PRINT para cada expresión

```

def generate_print(self, expression_array):
    expressions = []

    for i in range(expression_array):
        operand, operand_type = self.pop_operand()

```

```

        expressions.append(operand)

    expressions.reverse()

    for expression in expressions:
        self.add_quad(operator="PRINT", arg1=expression, arg2=None)
        #             accion             expresion

```

13. `open_parenthesis()`: Agregamos el parentesis de apertura como plag en la pila de operadores

```

# Agregamos el operador de parentesis a la pila de operadores, e
def open_parenthesis(self):
    self.operator_stack.append("(")

```

14. `close_parenthesis()`: Procesar operadores dentro del parentesis y eliminar a su apertura de la pila de operadores una vez que se encuentre

```

def close_parenthesis(self, semantic_cube: SemanticCube):
    # Iteramos a travez de la pila de operadores utilizando a "("
    # donde el punto neuralgico generate_arithmetic_operation to
    # y consume a los operadores hasta llegar a la apertura del
    while self.operator_stack and self.peak_operator() != "(":
        self.generate_arithmetic_operation(semantic_cube=semantic_cube)

    # Eliminamos a la apertura del parentesis de la pila de oper
    if self.operator_stack and self.peak_operator() == "(":
        self.pop_operator()

```

15. `get_temp_address()`: Asigna direcciones de memoria para las variables temporales

```

def get_temp_address(self, name, var_type):
    return execution_memory.add_temp(var_type=var_type, name=name)

```

16. `add_const()`: Asigna direcciones de memoria a las constantes

```

def add_const(self, value, value_type):
    key = (value, value_type)

    if key in self.const_dict:
        return self.const_dict[key]

```

```

# Si la constante aun no existe
address = self.assign_address(scope="const", var_type=value_
self.const_dict[key] = address

return address

```

17. `p_func_start()`: Empieza la llamada a la funcion, utiliza a la funcion `begin_function_call` para registrar a la funcion dentro de la tabla de funciones junto con sus recursos

```

def p_func_start(p):
    '''func_start : empty'''
    func_name = p[-1]
    semantic_analyzer.np_start_function(func_name, 'void')
    intermediate_code_generator.begin_function_call(func_name)
    p[0] = None

# Llamadas a funciones
def begin_function_call(self, func_name):
    start_address = self.get_current_position()

    # Registramos a la funcion en la tabla de funciones junto co
    # Almacenamos la cantidad de variables locales y temporales
    # y la informacion de sus parametros
    self.function_table[func_name] = {
        "start_address": start_address,
        "param_addresses": [],
        "param_types": [],
        "local_vars": 0,
        "temp_vars": 0
    }

    print(f"Funcion {func_name} iniciada en {start_address}")

```

18. `p_func_end()`: Generamos un cuádruplo GOSUB para saltar a la funcion

```

def end_function_call(self, func_name):
    # Generamos un cuádruplo GOSUB para saltar a la funcion
    func_info = self.function_table.get(func_name)

    if func_info is None:
        print(f"Error: La funcion {func_name} no existe en la ta

```

```

        return

    start_address = func_info["start_address"]

    # Generamos el cuadruplo GOSUB
    self.add_quad(operator="GOSUB", arg1=func_name, arg2=None, r
    print(f"Llamada a funcion {func_name} finalizada, GOSUB a {s

    # Restauramos el estado previo a la llamada de funcion
    if self.call_stack:
        saved_state = self.call_stack.pop()
        self.current_function = saved_state["function_name"]
        self.param_counter = saved_state["param_counter"]
        self.argument_stack = saved_state["arguments"]
    else:
        self.current_function = None
        self.param_counter = 0
        self.argument_stack = []

```

19. p_program_start(): Inserta un cuadruplo GOTO al inicio de main

```

# Inicio del programa
def p_program_start(p):
    'program_start : empty'
    intermediate_code_generator.start_program()
    p[0] = None

def start_program(self):
    # Generamos un GOTO al inicio del main
    goto_pos = self.add_quad("GOTO", None, None, -1)
    print(f"Programa iniciado: GOTO al main {goto_pos}")

```

20. p_program_end(): Generamos un cuadruplo END para finalizar la ejecucion del programa

```

def p_program_end(p):
    'program_end : empty'
    intermediate_code_generator.end_program()
    p[0] = None

def end_program(self):
    # Generamos un cuadruplo END para finalizar el programa

```

```
self.add_quad("END", None, None, None)
print("Programa finalizado: END")
```

21. np_start_program(): Inicializa el análisis al reconocer la declaración del programa

```
def np_start_program(self, program_name):
    self.program_name = program_name
    if self.debug:
        print(f"Iniciando programa: {program_name}")
```

22. np_start_function(): Crea una función en el directorio y establece el scope actual

```
def np_start_function(self, func_name, return_type='void'):
    if self.debug:
        print(f"Declarando funcion: {func_name}")

    # Verificar que la funcion no este declarada
    if self.function_directory.function_exists(func_name):
        self.add_error(f"La funcion '{func_name}' ya fue declarada")
        return False

    # Agregamos a la funcion al directorio
    self.function_directory.add_function(func_name, return_type)
    self.function_directory.set_current_function(func_name)
    self.current_function = func_name
    return True
```

23. np_end_function(): Finaliza el análisis de una función y retorna al scope global

```
def np_end_function(self, func_name):
    if self.debug:
        print(f"Fin de la funcion: {func_name}")
    self.current_function = None
    self.function_directory.set_current_function(None)
```

24. np_add_parameter(): Agrega cada parámetro a la tabla de variables locales de la función

```
def np_add_parameter(self, param_name, param_type):
    if self.debug:
        print(f"Agregando parametro: {param_name}:{param_type}")
```

```

if self.current_function:
    func = self.function_directory.get_function(self.current_function)
    if func:
        if not func.add_parameter(param_name, param_type):
            self.add_error(
                f"Parametro '{param_name}' doblemente declarado"
            )

```

25. `np_declare_variable()`: Valida y registra una variable en el scope actual (global o local)

```

def np_declare_variable(self, var_name, var_type, is_global=False):
    # Definimos el scope
    scope = "global" if is_global else self.current_function
    if self.debug:
        print(f"Declarando variable: {var_name}:{var_type}, scope: {scope}")

    success = False
    if is_global:
        success = self.function_directory.add_global_variable(var_name, var_type)
    else:
        success = self.function_directory.add_local_variable(var_name, var_type, scope)

    if not success:
        self.add_error(
            f"La variable '{var_name}' ya fue declarada, scope: {scope}"
        )

```

26. `np_check_variable()`: Busca una variable y retorna su tipo, registra error si no existe

```

def np_check_variable(self, var_name):
    var = self.function_directory.lookup_variable(var_name)

    if var is None:
        self.add_error(f"La variable '{var_name}' no existe")
        return None

    return var.var_type

```

27. `np_check_operation()`: Valida una operación binaria usando el cubo semántico

```

def np_check_operation(self, operator, left_operand, right_operand,
    if left_type is None or right_type is None:
        return None

    result_type = semantic_cube.get_result_type(operator, left_type,

    if result_type is None:
        self.add_error(
            f"Operacion invalida: {left_operand}({left_type}) {opera
        )
        return None
    if self.debug:
        print(f"Resultado: {result_type}")
    return result_type

```

28. np_check_assignment(): Valida compatibilidad de tipos en una asignación

```

def np_check_assignment(self, var_name, expr_type):
    if self.debug:
        print(f"Verificando asignacion: {var_name} = <expr:{expr_type}

    # Checamos si la variable existe
    var = self.function_directory.lookup_variable(var_name)
    if var is None:
        self.add_error(f"La variable '{var_name}' no existe ")
        return False

    var_type = var.var_type

    # Verificar que los tipos sean compatibles mediante el cubo sema
    result_type = semantic_cube.get_result_type('=', var_type, expr_

    if result_type is None:
        self.add_error(
            f"Tipo incompatible: no se puede asignar {expr_type} a {
        )
        return False

    return True

```

29. np_start_function_call(): Verifica que una función exista antes de llamarla

```

def np_start_function_call(self, func_name):
    if self.debug:
        print(f"Llamando a la funcion: {func_name}")

    func = self.function_directory.get_function(func_name)
    if func is None:
        self.add_error(f"La funcion '{func_name}' no existe")
        return None

    return func

```

30. np_check_function_call(): Valida el número y tipos de argumentos en una llamada a función

```

def np_check_function_call(self, func_name, argument_types):
    if self.debug:
        print(f"Verificando llamada: {func_name}({' , '.join(argument_types)}")

    func = self.function_directory.get_function(func_name)
    if func is None:
        return False

    expected_types = func.get_parameter_types()

    # Verificar número de argumentos
    if len(argument_types) != len(expected_types):
        self.add_error(
            f"Numero incorrecto de argumentos en la funcion '{func_name}' "
            f"se esperan {len(expected_types)} pero se recibieron {len(argument_types)}"
        )
        return False

    # Verificar los tipos de los argumentos
    for i, (arg_type, expected_type) in enumerate(zip(argument_types, expected_types)):
        if arg_type != expected_type:
            # Verificar si hay conversiones validas
            if not (expected_type == 'float' and arg_type == 'int'):
                self.add_error(
                    f"Tipo incorrecto para el argumento {i+1} para la funcion '{func_name}' "
                    f"se espera {expected_type}, se recibio {arg_type}"
                )
            return False

```

```
return True
```

31. np_check_function_return(): Valida que el tipo de retorno sea compatible con la declaración

```
def np_check_function_return(self, func_name, return_type):
    func = self.function_directory.get_function(func_name)
    if func is None:
        return False

    expected_return_type = func.return_type

    # Si la funcion es void, no deberia tener return con valor
    if expected_return_type == 'void':
        self.add_error(
            f"La funcion '{func_name}' es void y no debe retornar un valor"
        )
        return False

    # Verificar compatibilidad de tipos
    if return_type != expected_return_type:
        # Permitir int -> float
        if not (expected_return_type == 'float' and return_type == 'int'):
            self.add_error(
                f"Tipo de retorno incompatible en '{func_name}': "
                f"se espera {expected_return_type}, se recibio {return_type}"
            )
            return False

    return True
```

32. register_function(): Registra una función en la tabla de funciones y crea variable global para retorno

```
def register_function(self, func_name, return_type='void'):
    start_address = self.get_current_position()

    # Si la funcion tiene tipo de retorno (no void), crear variable
    return_var_address = None
    if return_type != 'void':
        # Crear variable global con el nombre de la funcion para almacenar el retorno
        return_var_address = execution_memory.add_variable(func_name, return_type, start_address)
```

```

        if self.debug:
            print(f"Variable de retorno creada para {func_name}: {return_type}")

        # Registramos a la funcion en la tabla de funciones junto con su return_type
        self.function_table[func_name] = {
            "start_address": start_address,
            "param_addresses": [],
            "param_types": [],
            "local_vars": 0,
            "temp_vars": 0,
            "return_type": return_type,
            "return_var_address": return_var_address
        }

```

33. `add_function_param()`: Agrega un parámetro a la tabla de funciones

```

def add_function_param(self, func_name, param_address, param_type):
    if func_name in self.function_table:
        self.function_table[func_name]["param_addresses"].append(param_address)
        self.function_table[func_name]["param_types"].append(param_type)
    if self.debug:
        print(f"Parametro agregado a {func_name}: {param_address} {param_type}")

```

34. `end_function()`: Genera cuádruplo ENDFUNC para terminar declaración de función

```

def end_function(self, func_name):
    # Generamos un cuádruplo ENDFUNC
    self.add_quad(operator="ENDFUNC", arg1=None, arg2=None, result=None)
    if self.debug:
        print(f"Funcion {func_name} finalizada")

```

35. `begin_function_call()`: Genera cuádruplo ERA para reservar espacio de memoria para la llamada

```

def begin_function_call(self, func_name):
    if self.current_function is not None:
        self.call_stack.append({
            "function_name": self.current_function,
            "param_counter": self.param_counter,
            "arguments": self.argument_stack.copy()
        })
        self.argument_stack = []

```

```

self.current_function = func_name
self.param_counter = 0

# Generamos un cuádruplo ERA
self.add_quad(operator="ERA", arg1=func_name, arg2=None, result=
if self.debug:
    print(f"Llamando a la funcion: {func_name}")

```

36. process_function_argument(): Procesa un argumento y genera cuádruplo PARAM

```

def process_function_argument(self, argument_address, argument_type)
    if self.current_function is None:
        print("Error: No se puede procesar un argumento si no hay un
        return

    # Comprobamos que la funcion exista en la tabla de funciones
    func_info = self.function_table.get(self.current_function)
    if func_info is None:
        print(f"Error: La funcion {self.current_function} no existe
        return

    # Verificamos que la cantidad de argumentos no exceda a la canti
    expected_params = len(func_info["param_addresses"])
    if self.param_counter >= expected_params:
        print(f"Error: Demasiados argumentos para la funcion {self.c
        return

    # Obtenemos la direccion del parametro correspondiente
    param_address = func_info["param_addresses"][self.param_counter]

    # Generamos un cuádruplo PARAM
    self.add_quad(operator="PARAM", arg1=argument_address, arg2=None
    self.param_counter += 1

```

37. end_function_call(): Genera cuádruplo GOSUB para saltar a la función

```

def end_function_call(self, func_name):
    # Generamos un cuádruplo GOSUB para saltar a la funcion
    func_info = self.function_table.get(func_name)

    if func_info is None:
        print(f"Error: La funcion {func_name} no existe en la tabla

```

```

        return

    start_address = func_info["start_address"]

    # Generamos el cuádruplo GOSUB
    self.add_quad(operator="GOSUB", arg1=func_name, arg2=None, resul

    # Restauramos el estado previo a la llamada de funcion
    if self.call_stack:
        saved_state = self.call_stack.pop()
        self.current_function = saved_state["function_name"]
        self.param_counter = saved_state["param_counter"]
        self.argument_stack = saved_state["arguments"]
    else:
        self.current_function = None
        self.param_counter = 0
        self.argument_stack = []

```

38. generate_return(): Genera cuádruplo RETURN para asignar valor a variable global de retorno

```

def generate_return(self, func_name, return_value_address):
    func_info = self.function_table.get(func_name)
    if func_info is None:
        print(f"Error: Funcion {func_name} no encontrada")
        return

    return_var_address = func_info.get("return_var_address")
    if return_var_address is None:
        print(f"Error: La funcion {func_name} es void y no puede ret
        return

    # Generamos cuádruplo RETURN: asigna el valor a la variable glob
    self.add_quad(operator="RETURN", arg1=return_value_address, arg2

```

39. start_program(): Genera GOTO inicial al main (pendiente de llenar)

```

def start_program(self):
    # Generamos un GOTO al inicio del main
    goto_pos = self.add_quad("GOTO", None, None, -1)
    if self.debug:
        print(f"Programa iniciado: GOTO al main {goto_pos}")

```

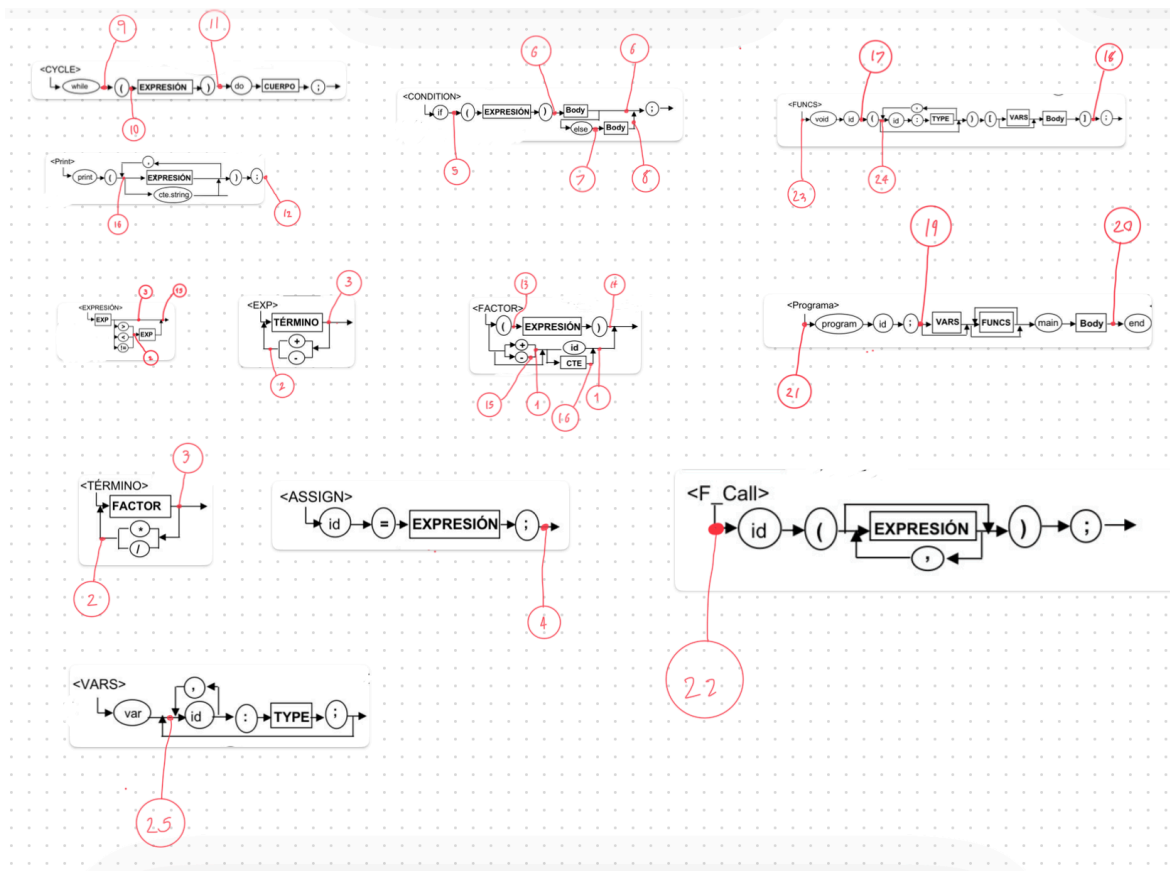
40. begin_main(): Llena el GOTO pendiente para saltar al inicio de main

```
def begin_main(self):  
    # Llenamos el GOTO pendiente de la posicion 0 al finalizar el ar  
    if len(self.quads) > 0 and self.quads[0][0] == 'GOTO' and self.c  
        self.fill_quad(0, self.get_current_position())  
    if self.debug:  
        print(f"Main Start: GOTO inicial apunta a {self.get_curr
```

41. end_program(): Genera cuádruplo END para finalizar el programa

```
def end_program(self):  
    # Generamos un cuádruplo END para finalizar el programa  
    self.add_quad("END", None, None, None)  
    if self.debug:  
        print("Programa finalizado: END")
```

Diagramas con los Cuadros Representados



Maquina Virtual

A continuación se explicará el funcionamiento de la máquina virtual, su funcionalidad está encapsulada en una clase la cual recibe a los cuádruplos desde el generador de código intermedio, y a las constantes desde la memoria de ejecución.

La clase se forma de los siguientes atributos:

1. **self.memory:** Este es un diccionario de diccionarios donde las llaves de los primeros diccionarios representan al alcance de la variable (global, local, temporal, constante), y el valor para estas llaves son diccionarios que guardan como llave a las direcciones de las variables, y como valor al valor de dicha variable, esta vez las variables no están separadas por tipos como lo estaban en la memoria de ejecución. El diccionario empieza vacío:

```
self.memory = {
    'global': {},          # Variables globales (1000-2999)
    'local': {},           # Variables locales (3000-4999) - por contexto
    'temp': {},            # Temporales (5000-6999) - por contexto
    'const': {}            # Constantes (7000-9999)
}
```

Conforme guardamos las constantes, nuestro diccionario muta de la siguiente manera:

```
self.memory = {
    'global': {
        1000: 10,
        1001: 20,          # y (dirección 1001, tipo int)
        2000: 15.0          # resultado (dirección 2000, tipo float)
    },
    'local': {
        # Variables locales como parámetros de funciones, salidas de
        3000: 10,
        3001: 20,
        3002: 30
    },
    'temp': {
        # Variables temporales
        5000: 30,
        6000: 15.0
    },
    'const': {
        # Valores numéricos y strings
    }
}
```

```

    7000: 2,
    9000: "Promedio de",
    9001: "y",
    9002: "es:"
}
}

```

2. **self.execution_stack**: Esta pila almacena los contextos de ejecución para las llamadas de función, donde cada vez que entramos a una función, agregamos contexto a una pila, y cuando la función termina, sacamos su contexto y volvemos al anterior. Un contexto se ve de la siguiente manera:

```

{
    'name': 'nombre_funcion',      # Nombre de la función
    'local': {},                  # Variables locales (3000-3999,
    'temp': {},                   # Variables temporales (5000-6999)
    'return_address': 25          # GOTO
}

```

3. **self.current_context**: Representa al contexto activo, si es None, significa que estamos en main, por lo que no hay funciones activas, cada contexto tiene sus propias variables locales y temporales separadas
4. **self.ip**: Es el índice del cuádruplo actual que se está ejecutando, lo incrementamos después de cada cuádruplo a excepción de cuando hay saltos
5. **self.quads**: La lista de instrucciones que se deben de ejecutar
6. **self.function_directory**: Almacena la información de todas las funciones del programa
7. **self.running**: Esta flag indica si la máquina virtual debe de seguir ejecutándose, cambia a false cuando se llega al cuádruplo end

Metodos

def get_segment(self, address): Recibe una dirección virtual y determina el segmento de memoria al que pertenece, como retorno arroja una tupla con el segmento y el tipo de dato que corresponden a la dirección de memoria, esto permite saber donde buscar o guardar un valor

```

# Determina el segmento de memoria basado en la dirección virtual
def get_segment(self, address):
    if 1000 <= address <= 1999:
        return 'global', 'int'

```

```

elif 2000 <= address <= 2999:
    return 'global', 'float'
elif 3000 <= address <= 3999:
    return 'local', 'int'
elif 4000 <= address <= 4999:
    return 'local', 'float'
elif 5000 <= address <= 5999:
    return 'temp', 'int'
elif 6000 <= address <= 6999:
    return 'temp', 'float'
elif 7000 <= address <= 7999:
    return 'const', 'int'
elif 8000 <= address <= 8999:
    return 'const', 'float'
elif 9000 <= address <= 9999:
    return 'const', 'string'
else:
    raise RuntimeError(f"Direccion de memoria invalida: {address}")

```

def get_value(self, address): Lee un valor desde una direccion virtual, primero se busca en el contexto actual, y luego en la memoria global

```

# Obtener el valor almacenado en una direccion virtual
def get_value(self, address):
    if address is None:
        return None

    # PRIMERO buscar en el contexto actual (para parametros y locales)
    if self.current_context:
        # Buscar en local del contexto
        if address in self.current_context['local']:
            return self.current_context['local'][address]
        # Buscar en temp del contexto
        if address in self.current_context['temp']:
            return self.current_context['temp'][address]

    segment, var_type = self.get_segment(address)

    if segment == 'global' or segment == 'const':
        # Variables globales y constantes estan en memoria principal
        if address in self.memory[segment]:
            return self.memory[segment][address]
        else:

```

```

        # Valor por defecto segun tipo
        return 0 if var_type in ['int', 'float'] else ""
    else:
        # Locales y temporales
        if self.current_context and address in self.current_context:
            return self.current_context[segment][address]
        elif address in self.memory[segment]:
            # Fallback a memoria principal (para main)
            return self.memory[segment][address]
        else:
            return 0 if var_type in ['int', 'float'] else ""

```

def set_value(self, address, value): Escribe un valor en una direccion virtual, si hay un contexto activo y la direccion esta dentro de las variables del contexto, escribe ahi dentro, si no, escribe en la memoria global.

```

def set_value(self, address, value):
    if address is None:
        return

    segment, var_type = self.get_segment(address)

    # Conversion de tipo si es necesario
    if var_type == 'int' and isinstance(value, float):
        value = int(value)
    elif var_type == 'float' and isinstance(value, int):
        value = float(value)

    # Si hay contexto y la direccion ya esta en local/temp del contexto
    if self.current_context:
        if address in self.current_context['local']:
            self.current_context['local'][address] = value
            return
        if address in self.current_context['temp']:
            self.current_context['temp'][address] = value
            return
        # Si es temporal, guardarlo en el contexto
        if segment == 'temp':
            self.current_context['temp'][address] = value
            return

    # Si no hay contexto o es global/const
    if segment == 'global' or segment == 'const':

```

```

        self.memory[segment][address] = value
    else:
        if self.current_context:
            self.current_context[segment][address] = value
        else:
            self.memory[segment][address] = value

```

def load_quads(self, quads): Recibe a los cuádruplos y los guarda en self.quads, luego coloca al instruction pointer en 0

```

def load_quads(self, quads):
    # Cargamos los cuádruplos a ejecutar
    self.quads = quads
    self.ip = 0

    # Verificar que exista el cuádruplo END
    has_end = any(q[0] == 'END' for q in quads)
    if not has_end and len(quads) > 0:
        print("Advertencia: No se encontro cuádruplo END. El pro

```

def load_constants(self, const_dict): Recibe a las constantes desde la memoria de ejecución y los almacena en el diccionario de const dentro de self.memory

```

# Cargamos las constantes del compilador a la memoria
def load_constants(self, const_dict):
    for (value, _), address in const_dict.items():
        self.memory['const'][address] = value

```

def create_context(self, func_name): Recibe al nombre de la función y retorna un contexto de ejecución para la función, este contexto estaría pendiente hasta que se ejecute el cuádruplo GOSUB

```

# Crea un nuevo contexto de ejecución para una llamada a función
def create_context(self, func_name):
    context = {
        'name': func_name,
        'local': {},
        'temp': {},
        'return_address': self.ip # Donde regresar después de l
    }

```

```
return context
```

push y pop context: Push context activa al contexto recibido, y si hay un contexto actual, mete al contexto actual dentro de la pila de ejecucion, pop context saca un contexto de la pila de ejecucion y lo establece como el contexto actual, si la pila esta vacia, cambia el contexto actual a none, lo que nos regresa a main

```
def push_context(self, context):  
    # Guarda el contexto actual y activa el nuevo  
    if self.current_context:  
        self.execution_stack.append(self.current_context)  
    self.current_context = context  
  
def pop_context(self):  
    # Restaura el contexto anterior  
    if self.execution_stack:  
        self.current_context = self.execution_stack.pop()  
    else:  
        self.current_context = None
```

def execute(self): Ejecuta a los cuádruplos de uno por uno usando el instruction pointer, lee cada cuádruplo, identifica al operador y llama a la función correspondiente, esta decisión se lleva a cabo mediante un switch

```
def execute(self):  
    # Ejecuta los cuádruplos cargados  
    self.running = True  
    self.ip = 0  
  
    # Proteccion contra loops infinitos  
    max_iterations = 100000  
    iteration_count = 0  
  
    while self.running and self.ip < len(self.quads):  
        iteration_count += 1  
        if iteration_count > max_iterations:  
            print(f"\nError: Posible loop infinito detectado des  
            print(f"IP actual: {self.ip}, Cuádruplo: {self.quads  
            break  
  
        quad = self.quads[self.ip]  
        operator, arg1, arg2, result = quad
```

```

# Dispatch de operaciones
if operator == '+':
    self.exec_add(arg1, arg2, result)
elif operator == '-':
    self.exec_sub(arg1, arg2, result)
elif operator == '*':
    self.exec_mult(arg1, arg2, result)
elif operator == '/':
    self.exec_div(arg1, arg2, result)
elif operator == '=':
    self.exec_assign(arg1, result)
elif operator == '>':
    self.exec_greater(arg1, arg2, result)
elif operator == '<':
    self.exec_less(arg1, arg2, result)
elif operator == '!=':
    self.exec_not_equal(arg1, arg2, result)
elif operator == 'GOTO':
    self.exec_goto(result)
    continue # No incrementar instruction pointer
elif operator == 'GOTOF':
    if self.exec_gotof(arg1, result):
        continue # Salto realizado, no incrementar inst
elif operator == 'GOTOT':
    if self.exec_gotot(arg1, result):
        continue
elif operator == 'PRINT':
    self.exec_print(arg1)
elif operator == 'UMINUS':
    self.exec_uminus(arg1, result)
elif operator == 'ERA':
    self.exec_era(arg1)
elif operator == 'PARAM':
    self.exec_param(arg1, result)
elif operator == 'GOSUB':
    self.exec_gosub(arg1, result)
    continue # El IP se maneja en gosub
elif operator == 'ENDFUNC':
    self.exec_endfunc()
    continue
elif operator == 'RETURN':
    self.exec_return(arg1, result)
elif operator == 'END':

```

```

        self.running = False
        continue
    else:
        print(f"Operador desconocido: {operator}")

    self.ip += 1

```

Los metodos para las operaciones aritmeticas y relacionales son practicamente iguales, reciben los mismos argumentos, los cuales son las direcciones de memoria de los argumentos 1 y 2 del cuádruplo, y la direccion del resultado, utilizamos a la funcion `get_value` para obtener los valores asociados a las direcciones de memoria de los argumentos, y luego usamos a la funcion `set_value` para asignarle el resultado de la operacion a la direccion de memoria del resultado del cuádruplo.

Las unicas operaciones que son distintas son la asignacion, division y la aplicacion del menos unario ya que la asignacion solo asigna un argumento a una direccion, la division valida que el valor que se vaya a dividir no sea 0, y el menos unario solo vuelve negativo al segundo argumento; A continuacion se enlistan las declaraciones de los metodos aritmeticos:

```

# Operaciones aritmeticas
# result = arg1 + arg2
def exec_add(self, arg1, arg2, result):

    # Resta: result = arg1 - arg2
    def exec_sub(self, arg1, arg2, result):

        # Multiplicacion: result = arg1 * arg2
        def exec_mult(self, arg1, arg2, result):

            # result = arg1 / arg2
            def exec_div(self, arg1, arg2, result):

                # Menos unario: result = -arg1
                def exec_uminus(self, arg1, result):

```

Luego van los metodos de control de flujo, que cambian al instruction pointer a su siguiente posicion dependiendo de como evalúe cada condicion, reciben como argumentos a la direccion de memoria con el resultado de la condicion, y al indice al que debe de ser movido el instruction pointer, si la condicion solicitada por el salto se cumple, se cambia la posicion del instruction pointer.

```

# Control de flujo
# GOTO
def exec_goto(self, target):
    self.ip = target

# GOTOF, cambia el instruction pointer al siguiente cuádruplo si
def exec_gotof(self, condition, target):
    val = self.get_value(condition)
    if val == 0:
        self.ip = target
        return True
    return False

# GOTOT, cambia el instruction pointer al siguiente cuádruplo si
def exec_gotot(self, condition, target):
    val = self.get_value(condition)
    if val != 0:
        self.ip = target
        return True
    return False

```

def exec_print(self, arg1): Este metodo verifica si el argumento que se va a imprimir es un string, si lo es, se limpia eliminando las comillas y de esa manera solo imprimir el contenido del string, luego se imprime el contenido usando la funcion print de python con el argumento end=' ' para que el valor se imprima junto con un espacio en lugar de un \n (newline)

```

# Imprimir el valor de arg1
def exec_print(self, arg1):
    val = self.get_value(arg1)
    # Si es string, remover comillas
    if isinstance(val, str) and val.startswith('"') and val.endswith('"'):
        val = val[1:-1]
    print(val, end=' ')

```

def exec_era(self, func_name): Utiliza a la funcion create_context para preparar un espacio de la memoria para la funcion y luego registra a su contexto como pendiente

```

# Preparamos el espacio de memora para la funcion y creamos un r
def exec_era(self, func_name):
    context = self.create_context(func_name)

```

```
# Guardamos el contexto pendiente, se activara con GOSUB  
self.pending_context = context
```

def exec_param(self, arg_address, param_address): Este metodo pasa un argumento a un parametro de una funcion, primero se obtiene el valor del argumento, luego utilizamos a hasattr para verificar si pending_context existe, ya que este atributo se crea dinamicamente en exec_era y se elimina en exec_gosub, y si intentamos acceder a el cuando no existe, python nos arrojaría un attribute error

```
# Pasamos un argumento al parametro de la funcion  
def exec_param(self, arg_address, param_address):  
    value = self.get_value(arg_address)  
    # Guardamos en el contexto pendiente como local  
    # Los parametros siempre van al contexto local de la funcion  
    if hasattr(self, 'pending_context'): # Revisamos que el atributo existe  
        # Usar local para todos los parametros, independiente de la funcion  
        self.pending_context['local'][param_address] = value
```

def exec_gosub(self, func_name, start_address): Verificamos si hay algun contexto pendiente, si lo hay, le asignamos el indice despues del instruction pointer como salto ya que seria la siguiente instruccion despues de la ejecucion de la funcion, luego empujamos el contexto a la pila de ejecucion, eliminamos a el atributo pending context ya que el contexto ya esta en la pila de ejecucion, y movemos al instruction pointer al inicio de la funcion, si el atributo pending context no existe, solo seria necesario mover el instruction pointer al inicio de la funcion

```
# GOSUB  
# Saltamos a la funcion y activamos el contexto  
def exec_gosub(self, func_name, start_address):  
    if hasattr(self, 'pending_context'):  
        self.pending_context['return_address'] = self.ip + 1  
        self.push_context(self.pending_context)  
        del self.pending_context  
  
    self.ip = start_address
```

def exec_endfunc(self): Termina la ejecucion del contexto anterior, primero sacamos el siguiente salto del contexto de la funcion, sacamos el contexto de la pila de contexto y movemos el instruction pointer a la siguiente instruccion

```

# ENDFUNC
# Termina la ejecucion de la funcion, restaura el contexto anterior
def exec_endfunc(self):
    return_address = self.current_context['return_address']
    self.pop_context()
    self.ip = return_address

```

def exec_return(self, value_address, return_var_address): Asigna el valor de retorno a la direccion correspondiente a la funcion

```

# RETURN
# Asigna el valor de retorno a la variable local de la funcion
def exec_return(self, value_address, return_var_address):
    value = self.get_value(value_address)

    self.set_value(return_var_address, value)
    if hasattr(self, 'debug') and self.debug:
        print(f"RETURN: {value} -> direccion {return_var_address}")

```

def print_memory_state(self): Esta funcion imprime a las direcciones de memoria del contexto actual, sea main, o el de una funcion

```

# Helpers
def print_memory_state(self):
    print("\nEstado de la memoria:")
    print("Global:", self.memory['global'])
    print("Constantes:", self.memory['const'])
    if self.current_context:
        print("Contexto actual:", self.current_context['name'])
        print("  Local:", self.current_context['local'])
        print("  Temp:", self.current_context['temp'])

```

Por ultimo, la maquina virtual cuenta con un singleton para garantizar la existencia de una unica instancia de la maquina virtual para evitar la necesidad de recibirla como parametro para usarse en otras partes del codigo.

main.py

Esta es la clase encargada de administrar e integrar todos los modulos del compilador y la maquina virtual.

El constructor inicializa a los singletons de los modulos como atributos de la clase y define un atributo booleano llamado debug el cual nos permitira ocultar la visualizacion de los prints de debug.

El metodo compile es el encargado de compilar el codigo, recibe al programa como argumento, y empieza por vaciar al generador de codigo intermedio, al analizador semantico, y a la memoria de ejecucion para evitar que hayan variables, direcciones de memoria, cuadрупlos y funciones ajenas al programa que se desea ejecutar, luego se llama a la funcion parse() del parser generado con yacc para compilar el programa y asi llenar la tabla de variables, el directorio de funciones y generar los cuadрупlos necesarios para la ejecucion del codigo, si el parseo procede, se revisa si hay errores semanticos utilizando al metodo has_errors de el analizador semantico, de haber errores, la funcion arrojará false e imprimira los errores detectados, de otra manera, la funcion arrojará true, indicando que la compilacion fue exitosa

```
def compile(self, code):
    if self.debug:
        print(f"Compilando...:\n{code}\n")
        # Reset de todos los componentes
        self.intermediate_code_generator.reset()
        self.semantic_analyzer.reset()
        self.execution_memory.reset()

    try:
        # Parseamos el codigo
        result = self.parser.parse(code)
        if self.debug:
            print(f"Resultado del parseo: {result}")
        if result is None:
            print("Error: No se pudo parsear el codigo")
            return False

        # Analisis semantico
        if self.semantic_analyzer.has_errors():
            print("Errores semanticos encontrados:")
            self.semantic_analyzer.print_errors()
            return False

        return True # Compilacion exitosa

    except Exception as e:
        print(f"Error durante la compilacion: {e}")
        return False
```

La funcion run toma a los cuádruplos y a las constantes resultantes y se las da a la maquina virtual para la ejecutar a los cuádruplos, y por ulto

```
def run(self):
    # Cargar cuádruplos
    self.virtual_machine.load_quads(self.intermediate_code_generator.quads)
    self.virtual_machine.load_constants(self.execution_memory.constants)

    try:
        self.virtual_machine.execute()
    except Exception as e:
        print(f'Error durante la ejecucion: {e}')
```

Por ultimo, compile_and_run junta a las funciones de compilacion y ejecucion, primero se ejecuta la funcion compile, si arroja true, ejecuta al metodo run

```
def compile_and_run(self, code):
    if self.compile(code):
        if self.debug:
            print("MODULO DEBUG")
            self.print_function_table()
            self.print_quads()
            self.print_memory()

        self.run()
```

El resto de funciones son funciones auxiliares que imprimen distintos aspectos del compilador y activan el modo debug

```
def print_quads(self):
    self.intermediate_code_generator.print_quads()

def print_function_table(self):
    self.intermediate_code_generator.print_function_table()

def print_memory(self):
    print("\nMemoria: ")
    print("Variables:", self.execution_memory.var_dict)
    print("Constantes:", self.execution_memory.const_dict)
    print("\n")

def enable_debug(self):
```

```
self.debug = True
self.intermediate_code_generator.debug = True
```

Por ultimo esta el entry point de el proyecto, esta funcion lee argumentos desde la linea de comando, el primer argumento es la ruta al archivo con extension .patito, y el segundo argumento es un flag opcional --debug que habilita los prints de debugging del compilador, si el programa no recibe parametros, usa un codigo por defecto el cual esta escrito dentro del entry point. Primero se crea una instancia de la clase Compiler, luego se verifica si hay argumentos dentro de la funcion, si los hay, primero se lee al archivo y se guarda a su contenido en una variable, luego se verifica si se agrego al flag de debug como argumento, si esta presente, cambia el valor de debug en la clase del compilador, por ultimo, se le administra el codigo al compilador para que sea compilado y ejecutado.

```
def main():
    compiler = Compiler()

    # Determinar el codigo fuente a usar
    if len(sys.argv) > 1:
        # Leer desde archivo
        file = sys.argv[1]

        # Flag de debug
        if "--debug" in sys.argv or "-d" in sys.argv:
            compiler.enable_debug()
        try:
            with open(file, 'r') as f:
                code = f.read()
        except FileNotFoundError:
            print(f"Error: No se encontro el archivo '{file}'")
            return
    else:
        # Programa de ejemplo
        code = """program ejemplo;
var
    x, y : int;
main {
    x = 5;
    y = 10;
    print("Suma:", x + y);
}
end"""
        print("Usando programa de ejemplo")
```

```
# Compiler
compiler.compile_and_run(code=code)

if __name__ == "__main__":
    main()
```

Prompts utilizados

A lo largo de la entrega consulte a Claude para despejar mis dudas durante la elaboracion del compilador

"Como funcionan los indices en PLY?, es decir, que representa cada indice y por que el resultado se "Por que las temporales necesitan un contador único?" "Si voy a desarrollar una register based vm, que estructura dinamica deberia de usar para guardar mis direcciones de memoria?, estaba considerando utilizar un diccionario de diccionarios o un hashmap cuyos valores fueran listas doblemente enlazadas, que tan alejada esta esta idea?, es importante aclarar que aun no llego al punto de desarrollar la maquina virtual, solo voy a asignarles direcciones de memoria a mis varoables y constantes" "Como funciona la memoria de ejecucion?" "Como se integra comunmente la memoria de ejecucion?" "Cual es la diferencia entre manejar la creacion de temporales desde la memoria de ejecucion y el generador de codigo intermedio?" "Por que es posible acceder a partes anteriores de gramaticas desde otras reglas gramaticales y por que se usan indices negativos?" "Como funciona una pila de ejecucion?" "Inclui de manera correcta a el return de mis funciones en factor para poder usar a al valor de retorno de la funcion en operaciones sin tener la necesidad de crear una variable?" "Estoy almacenando de manera correcta a los valores de retorno de mis funciones?, y si que deberia de tomar en cuenta para que el contexto no falle en casos de recursion?"