

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> ) [ [<VARs>] <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.

Puntos Neuralgicos - Inicializacion y Funciones

El punto neuralgico `np_start_program` corre cuando se reconoce la declaracion del programa (program ID;) e inicializa el analisis estableciendo el nombre del programa.

El punto neuralgico `np_start_function` corre cuando comienza la declaracion de una funcion verificando que no este doblemente declarada, creandola en el directorio, y estableciendo que todos los elementos siguientes pertenecen a esa funcion hasta que termine.

El punto neuralgico `np_end_function` corre cuando finaliza la declaracion de una funcion y arroja el ambito al global.

El punto neuralgico np_add_parameter se ejecuta para cada parametro en la lista de parametros validando que no este doblemente declarado y agregandolo como variable local de la funcion.

Puntos Neuralgicos - Variables

El punto neuralgico np_declare_variable se ejecuta al reconocer una declaracion de variable (ID : type;). Valida que la variable no este doblemente declarada en el ambito actual y la agrega a la tabla global si es variable global, o a la tabla local de la funcion actual si es local.

El punto neuralgico np_check_variable se ejecuta cuando se usa una variable en una expresion, busca la variable en el scope actual implementando la regla que las variables locales ocultan las globales, luego arroja al tipo de la variable si existe.

Puntos Neuralgicos - Operaciones

El punto neuralgico np_check_operation corre cuando se completa una operacion binaria, consulta el cubo semantico para obtener el tipo resultante y registra un error si la operacion no es valida, el tipo resultante se almacena para su uso.

El punto neuralgico np_check_assignment corre cuando se completa una asignacion (ID = expression;), este verifica que la variable exista, obtiene su tipo, consulta el cubo semantico con el operador = para validar que el tipo de la expresion sea compatible con el tipo de la variable, y registra error si no lo es.

Puntos Neuralgicos - Llamadas a Funcion

El punto neuralgico np_start_function_call corre cuando comienza una llamada a funcion verificando que la funcion este declarada y obteniendo su informacion.

El punto neuralgico np_check_function_call corre al completar una llamada a funcion, valida que el numero de argumentos sea correcto, que cada tipo de argumento sea compatible con el tipo del parametro correspondiente, y registra errores especificos indicando cual argumento es incompatible.

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 me
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 representara 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
```

Diagramas con los Cuadрупlos Representados

