



Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 501)

Actividad Final Mini-Proyecto: Baby_Duck

Profesores:

Ing. Elda G. Quiroga, M.Sc.

Dr. Iván Mauricio Amaya Contreras

Dr. Jesús Guillermo Falcón Cardona

Juan Carlos Garfias Tovar

A01652138

16 de noviembre del 2023

Índice

Introducción.....	2
Lenguaje imperativo y máquinas virtuales.....	2
Expresiones regulares para elementos léxicos.....	3
Diagramas del lenguaje.....	5
Gramática BNF.....	5
Gramática en ANTLR.....	7
Cubo semántico.....	8
Directorio de Funciones y a las Tablas de Variables.....	9
Puntos neurálgicos.....	10
Generación de cuádruplos.....	11
Puntos neurálgicos en cuádruplos.....	12
Máquina virtual.....	15
Inicialización.....	15
Funcionamiento.....	15
Manejo de Variables y Constantes.....	15
Ejecución de Cuádruplos.....	15
Funciones Específicas.....	16
Depuración y Visualización.....	16
Manejo de memoria.....	16
Asignación de Direcciones de Memoria.....	16
Funciones de Asignación.....	16
Manejo de Tipos de Constantes.....	17
Traducción y Detección de Variables.....	17
Ejecución de Cuádruplos.....	17
Rangos de memoria.....	17
Salto con labels y placeholders.....	18
Clases y paquetes del programa.....	19
Pruebas de ejecución.....	20
Conclusión.....	21
Anexos.....	22

Introducción

La creación y el diseño de lenguajes de programación se han convertido en una herramienta esencial para la adaptación y optimización de procesos específicos en una variedad de áreas de la ciencia y la tecnología en la era digital actual. En este contexto, se presenta "Baby_Duck", un microlenguaje imperativo procedural clásico.

El objetivo principal de este proyecto es desarrollar un compilador para "Baby_Duck", pasando por las etapas fundamentales de análisis léxico, sintaxis, validaciones semánticas, generación de código intermedio y, finalmente, interpretación de código a través de una máquina virtual. El comprender el funcionamiento de los lenguajes de programación y los procesos de compilación e interpretación es fundamental. Este conocimiento no solo permite una comprensión profunda de la ejecución de instrucciones a nivel de máquina, sino que también es crucial para la optimización y adaptación del código a diferentes requisitos y plataformas. En este contexto, el proyecto presente ofrece una oportunidad única para profundizar en los detalles de la creación de un lenguaje y su compilador, fortaleciendo así la base teórica y práctica en el campo de la ciencia computacional.

Este documento abordará aspectos clave como las expresiones regulares para elementos léxicos, las reglas gramaticales en la creación de lenguajes, y el uso de diagramas y estructuras de datos en la implementación de un lenguaje de programación. Se explorará cómo las expresiones regulares facilitan el análisis léxico, permitiendo la identificación y categorización de tokens en el código fuente. Además, se discutirá la importancia de las Gramáticas Libres de Contexto (CFG) en la descripción de estructuras sintácticas y su aplicación en herramientas de compilación.

Finalmente, se examinará el papel de las máquinas virtuales en la ejecución de lenguajes de programación, destacando su contribución a la flexibilidad y portabilidad del código. Este análisis proporcionará una visión integral de cómo los lenguajes imperativos y las máquinas virtuales interactúan y se complementan en el ámbito de la programación moderna. Utilizando la programación del compilador para Baby_Duck como medio para poner en práctica los conocimientos adquiridos. Se utilizará ANTLR V.4 y Python 3.9.7 para la programación del compilador.

Lenguaje imperativo y máquinas virtuales

Un lenguaje procedural imperativo clásico se basa en la ejecución de instrucciones secuenciales y la definición de procedimientos o funciones que encapsulan un conjunto de estas instrucciones. El flujo del programa en este tipo de lenguajes se controla mediante estructuras de control como bucles y condicionales, y las variables almacenan y manipulan la información. Estos lenguajes reflejan un enfoque de programación en el que se le "ordena" al computador paso a paso qué hacer. El enfoque imperativo, en contraste con otros paradigmas, como el funcional o el orientado a objetos, se centra en el "cómo" se hacen las cosas en lugar del "qué" se quiere lograr.

La Máquina Virtual (MV) es esencial para la ejecución de lenguajes de programación porque permite la interpretación y ejecución del código escrito en una variedad de plataformas sin necesidad de recompilación. La MV ofrece flexibilidad y portabilidad, a diferencia de la compilación directa a código máquina, que está limitada por una arquitectura. Esto es particularmente crucial en entornos en los que se espera que una aplicación sea compatible con múltiples sistemas o dispositivos.

Cualquier científico de la computación o desarrollador necesita comprender cómo funcionan los lenguajes de programación y los procesos subyacentes de compilación e interpretación. Además de brindar una comprensión profunda de cómo se ejecutan las instrucciones a nivel de máquina, permite optimizar y adaptar el código para una variedad de requisitos. En particular, completar este proyecto ofrece una oportunidad única para profundizar en los detalles complejos de la creación de un lenguaje y su compilador, lo que fortalece la base teórica y práctica en el campo de la ciencia computacional.

Expresiones regulares para elementos léxicos

Las expresiones regulares son una herramienta poderosa para describir y reconocer secuencias de caracteres. En el contexto de la gramática para "Baby_Duck", las expresiones regulares para los elementos léxicos (tokens) son las siguientes:

1. **Identificadores (ID):** Representan nombres de variables, funciones, etc.
Expresión Regular: $[a-zA-Z][a-zA-Z0-9]^*$
2. **Números enteros (INTEGER):** Representan valores numéricos enteros.
Expresión Regular: $[0-9]^+$
3. **Números flotantes (FLOAT_NUM):** Representan valores numéricos con punto decimal.
Expresión Regular: $[0-9]^+ \backslash \. [0-9]^+$
4. **Cadenas de texto (STRING):** Representan secuencias de caracteres encerradas entre comillas.
Expresión Regular: $"\.*?"$
5. **Operadores y Comparadores:** Representan símbolos específicos utilizados para operaciones aritméticas y comparaciones.
 - PLUS: '+'
 - MINUS: '-'
 - MULTIPLY: '*'
 - DIVIDE: '/'
 - EQUALS: '='
 - NOTEQUALS: '!='

- GREATERTHAN: '>'
 - LESSTHAN: '<'
6. **Espacios en blanco (WS):** Representan espacios, tabulaciones y saltos de línea.
 Expresión Regular: $[\backslash t \backslash r \backslash n]^+$

Estas expresiones regulares sirven como base para el análisis léxico, permitiendo al *lexer* identificar y categorizar cada token en el código fuente.

Reglas gramaticales (Context Free Grammar)

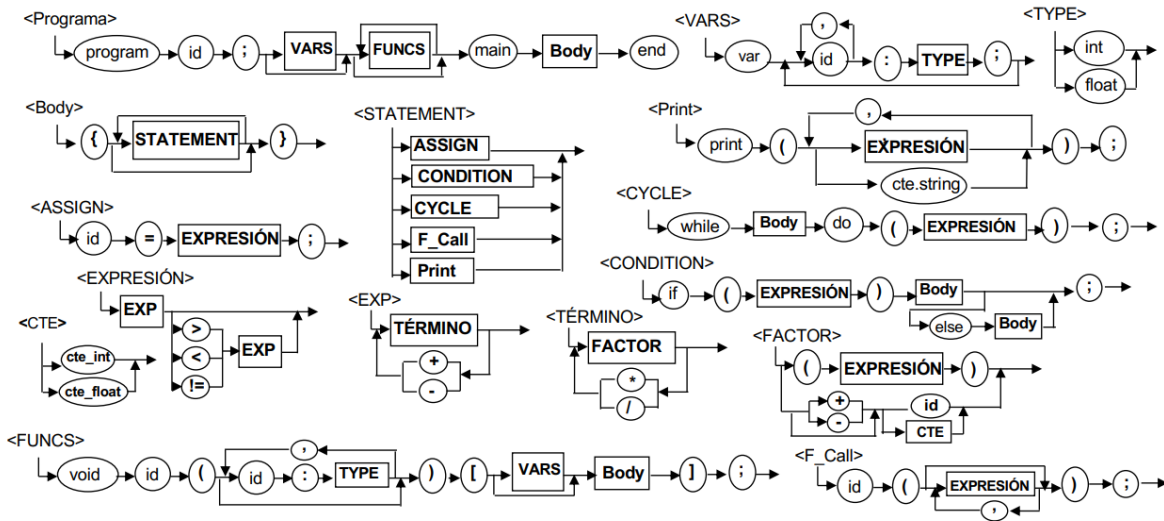
El núcleo de cualquier lenguaje son las reglas gramaticales, que proporcionan un conjunto organizado de instrucciones que definen cómo se pueden construir oraciones o estructuras válidas en ese lenguaje. Estas reglas son fundamentales para describir y discernir la estructura inherente de los lenguajes de programación en la informática y, más específicamente, en la teoría de lenguajes formales. La Gramática Libre de Contexto (GLC o CFG, por sus siglas en inglés, "Gramática Libre de Contexto") es una categoría particularmente importante de estas reglas.

Una CFG se distingue por su conjunto de reglas de producción, cada una de las cuales transforma un solo símbolo no terminal en una cadena que contiene tanto símbolos terminales como no terminales. Estas gramáticas son únicas por su naturaleza "libre de contexto", lo que significa que sus reglas de producción no están influenciadas por el contexto alrededor del símbolo no terminal. En otras palabras, cualquier reemplazo se realiza sin tener en cuenta los símbolos que se encuentran a su alrededor.

Las CFG son una parte importante de la compilación. Son capaces de describir con precisión las estructuras sintácticas de una amplia gama de lenguajes de programación. Esta habilidad permite a los compiladores y analizadores sintácticos procesar el código fuente de un programa y evaluar su validez de acuerdo con las reglas establecidas del lenguaje. Además, CFG proporciona la base teórica y práctica para una amplia gama de herramientas y técnicas en el campo de la compilación, incluidos los *parsers* descendentes y ascendentes.

Se ha optado por utilizar la Forma de *Backus-Naur* (BNF) como herramienta principal para la descripción detallada de la gramática de "Baby_Duck" mientras se desarrollaba este proyecto. BNF, conocida por su notación meta-sintáctica, se destaca por su capacidad intrínseca de representar con claridad y precisión las estructuras y reglas de una variedad de protocolos y lenguajes de programación. Esta notación facilita la comprensión y el análisis de la estructura del lenguaje y es una piedra angular en la teoría de compiladores, y se usa ampliamente en la academia y la industria para la creación y adaptación de lenguajes y sistemas. Garantiza un enfoque sistemático y estandarizado en el proceso de compilación y análisis.

Diagramas del lenguaje



Gramática BNF

<program> ::= 'program' <ID> ';' [<vars>] {<funcs>} <mainSection>

<mainSection> ::= <main> <body> <end>

<vars> ::= 'var' {<ID> {'<ID>' ':' <type> ';'}+}

<type> ::= <int> | <float>

<funcs> ::= <void> <ID> '(' [<ID> ':' <type> {'<ID> ':' <type>'}] ')' '[' [<vars>] <body> ']' ';' ;

<body> ::= '{' {<statement>} '}'

<statement> ::= <assign> | <condition> | <cycle> | <f_call> | <print>

<assign> ::= <ID> '=' <expression> ';' ;

<condition> ::= <if> '(' <expression> ')' <body> [<else> <body>] ';' ;

<cycle> ::= <while> <body> <do> '(' <expression> ')' ';' ;

<print> ::= <print_w> '(' (<expression> | <STRING>) {'(' <expression> | <STRING> ')' '*' } ')' ';' ;

<f_call> ::= <ID> '(' [<expression> {'<expression>'}] ')' ';' ;

<expression> ::= <exp> [<relop> <exp>]

<relop> ::= <GREATERTHAN> | <LESSTHAN> | <NOTEQUALS>

$\langle \text{exp} \rangle ::= \langle \text{termino} \rangle \{ (\langle \text{PLUS} \rangle \mid \langle \text{MINUS} \rangle) \langle \text{termino} \rangle \}^*$
 $\langle \text{termino} \rangle ::= \langle \text{factor} \rangle \{ (\langle \text{MULTIPLY} \rangle \mid \langle \text{DIVIDE} \rangle) \langle \text{factor} \rangle \}^*$

$\langle \text{factor} \rangle ::= \langle \text{parenthesized_expression} \rangle \mid \langle \text{unary_expression} \rangle \mid (\langle \text{ID} \rangle \mid \langle \text{cte} \rangle)$
 $\langle \text{parenthesized_expression} \rangle ::= '(' \langle \text{expression} \rangle ')'$
 $\langle \text{unary_expression} \rangle ::= (\langle \text{MINUS} \rangle \mid \langle \text{PLUS} \rangle) \langle \text{factor} \rangle$

$\langle \text{cte} \rangle ::= \langle \text{INTEGER} \rangle \mid \langle \text{FLOAT_NUM} \rangle$
 $\langle \text{INTEGER} \rangle ::= [0-9]^+$
 $\langle \text{FLOAT_NUM} \rangle ::= [0-9]^+ '.' [0-9]^+$
 $\langle \text{ID} \rangle ::= [a-zA-Z] ([a-zA-Z] \mid [0-9])^*$
 $\langle \text{STRING} \rangle ::= ''' . * ? '''$
 $\langle \text{PLUS} \rangle ::= '+'$
 $\langle \text{MINUS} \rangle ::= '-'$
 $\langle \text{MULTIPLY} \rangle ::= '*'$
 $\langle \text{DIVIDE} \rangle ::= '/'$
 $\langle \text{EQUALS} \rangle ::= '='$
 $\langle \text{NOTEQUALS} \rangle ::= '!='$
 $\langle \text{GREATERTHAN} \rangle ::= '>'$
 $\langle \text{LESSTHAN} \rangle ::= '<'$

$\langle \text{if} \rangle ::= \text{'if'}$
 $\langle \text{else} \rangle ::= \text{'else'}$
 $\langle \text{while} \rangle ::= \text{'while'}$
 $\langle \text{do} \rangle ::= \text{'do'}$
 $\langle \text{print_w} \rangle ::= \text{'print'}$
 $\langle \text{void} \rangle ::= \text{'void'}$
 $\langle \text{int} \rangle ::= \text{'int'}$
 $\langle \text{float} \rangle ::= \text{'float'}$
 $\langle \text{main} \rangle ::= \text{'main'}$
 $\langle \text{end} \rangle ::= \text{'end'}$

$\langle \text{WS} \rangle ::= [\text{'\t\r\n'}]^+ \rightarrow \text{skip}$

$\langle \text{LPAREN} \rangle ::= '('$
 $\langle \text{RPAREN} \rangle ::= ')'$
 $\langle \text{LBRACE} \rangle ::= \{'$
 $\langle \text{RBRACE} \rangle ::= \}'$
 $\langle \text{LBRACKET} \rangle ::= '['$
 $\langle \text{RBRACKET} \rangle ::= ']'$
 $\langle \text{SEMICOLON} \rangle ::= ';'$
 $\langle \text{COMMA} \rangle ::= ','$
 $\langle \text{COLON} \rangle ::= ':'$

Gramática en ANTLR

```

grammar BabyDuck;

// Parser Rules
program: 'program' ID SEMICOLON (vars)? (funcs)* mainSection;

mainSection: main body end;

vars: 'var' (ID (COMMA ID)* COLON type SEMICOLON)+;

type: int | float;

funcs: void ID LPAREN (ID COLON type (COMMA ID COLON type)*)? RPAREN LBRACKET (vars)? body RBRACKET SEMICOLON;

body: LBRACE (statement)* RBRACE;

statement: assign | condition | cycle | f_call | print;

assign: ID EQUALS expression SEMICOLON;

condition: if LPAREN expression RPAREN body (else body)? SEMICOLON;

cycle: while body do LPAREN expression RPAREN SEMICOLON;

print: print_w LPAREN (expression | STRING) (COMMA (expression|STRING))* RPAREN SEMICOLON;

f_call: ID LPAREN (expression (COMMA expression)* )? RPAREN SEMICOLON;

expression: exp (relop exp)?;
relop: GREATERTHAN | LESSTHAN | NOTEQUALS;

exp: termino ((PLUS | MINUS) termino)*;
termino: factor ((MULTIPLY | DIVIDE) factor)*;

factor: parenthesized_expression | unary_expression | (ID | cte);
parenthesized_expression: LPAREN expression RPAREN;
unary_expression: (MINUS | PLUS) factor;

cte: INTEGER | FLOAT_NUM;

// Tokens
INTEGER: [0-9]+;
FLOAT_NUM: [0-9]+ '.' [0-9]+;
ID: [a-zA-Z] ([a-zA-Z] | [0-9])*;
STRING: '"' .*? '"';

// Operators and Comparators
PLUS: '+';
MINUS: '-';
MULTIPLY: '*';
DIVIDE: '/';
EQUALS: '=';
NOTEQUALS: '!=';
GREATERTHAN: '>';
LESSTHAN: '<';

// Keywords
if: 'if';
else: 'else';
while: 'while';
do: 'do';
print_w: 'print';
void: 'void';
int: 'int';
float: 'float';
main: 'main';
end: 'end';

// Whitespace
WS: [ \t\r\n]+ -> skip;

// Punctuation
LPAREN: '(';
RPAREN: ')';
LBRACE: '{';
RBRACE: '}';
LBRACKET: '[';
RBRACKET: ']';
SEMICOLON: ';';
COMMA: ',';
COLON: ':';

```


Cubo semántico

En la clase *Visitor* además de recorrer el árbol, analiza operaciones aritméticas y utiliza un "cubo semántico" para generar cuádruplos. El tipo de operandos determina el tipo de resultado de las operaciones en este cubo. La clase maneja operandos, operadores y cuádruplos con pilas.

- El método `check_semantic_cube` se utiliza para verificar si una operación dada entre dos tipos de datos es válida según el cubo semántico. Este método toma como argumentos los tipos de los operandos izquierdo y derecho y el operador, y devuelve el tipo de dato resultante si la operación es válida. Si la combinación de tipos y operador no es válida, se lanza una excepción `TypeError`.
- El método `process_operator` procesa un operador extrayendo los operandos y el operador de sus respectivas pilas, genera una nueva variable temporal para almacenar el resultado, y utiliza `check_semantic_cube` para validar la operación y obtener el tipo de dato resultante. Luego, se genera un cuádruplo con esta información.
- Los cuádruplos generados se almacenan en la lista `quadruples`, y cada cuádruplo contiene el operador, los operandos izquierdo y derecho, la variable temporal donde se almacena el resultado, el ámbito (scope) actual y, opcionalmente, el tipo de dato objetivo.

Uso del Cubo Semántico

El "cubo semántico" (Semantics) es esencialmente un diccionario de diccionarios que define cómo se deben manejar las operaciones entre diferentes tipos de datos. Por ejemplo, si tienes un operando de tipo `int` y otro de tipo `float`, y quieres sumarlos, el cubo te dice que el resultado debe ser de tipo `float`. Si una combinación de tipos y operador no está definida en el cubo, se considera un error.

Directorio de Funciones y a las Tablas de Variables

En el archivo *semanticTable.py* se definen dos clases, *DirFunc* y *VarTable*, que representan estructuras de datos para un Directorio de Funciones y Tablas de Variables, respectivamente. Ambas clases utilizan la biblioteca *pandas* para manejar y organizar la información.

Clase *DirFunc* (Directorio de Funciones):

- `__init__(self)`: Inicializa un *DataFrame* con columnas "id-name", "scope" y "var-table".
- `add_func(self, id_name, func_type, scope="Global")`: Añade una función al directorio. Si la función ya existe, se lanza un error.
- `delete(self)`: Borra todas las entradas del directorio de funciones.

Clase *VarTable* (Tabla de Variables):

- `__init__(self)`: Inicializa un *DataFrame* con columnas "id-name", "type", "value" y "scope".
- `add_var(self, id_name, var_type, value=None, scope="Global")`: Añade una variable a la tabla. Si la variable ya existe, se lanza un error.
- `delete(self)`: Borra todas las entradas de la tabla de variables.

Ventajas de usar un *DataFrame*:

- **Organización estructurada:** Los *DataFrames* proporcionan una estructura tabular que es fácil de leer y manipular.
- **Flexibilidad:** Es fácil añadir, modificar o eliminar registros en un *DataFrame*.
- **Funcionalidad integrada:** *pandas* ofrece una amplia gama de funciones para manipular, filtrar, ordenar y agrupar datos.
- **Rendimiento:** *pandas* está optimizado para operaciones de datos y puede manejar grandes conjuntos de datos de manera eficiente.
- **Integración con otras bibliotecas:** *pandas* se integra bien con otras bibliotecas de análisis de datos y visualización, lo que facilita la expansión de la funcionalidad si es necesario.
- **Verificación de duplicados:** Como se muestra en el código, es sencillo verificar la existencia de registros duplicados antes de añadir nuevos registros.

Este código cuenta el manejo de errores, el cual se centra específicamente en evitar declaraciones múltiples de funciones o variables con el mismo nombre. Para lograr esto, antes de añadir una nueva función o variable, el código verifica si el nombre ya existe en el *DataFrame* correspondiente. Si se detecta una coincidencia, se lanza una excepción *ValueError* con un mensaje que indica el error de "declaración múltiple".

Puntos neurálgicos

El código define una clase `Listener` que hereda de `BabyDuckListener`. Esta clase se utiliza para analizar y procesar el árbol de sintaxis abstracta generado por un analizador sintáctico, específicamente para un lenguaje llamado "BabyDuck". Aquí están los puntos neurálgicos y cómo funcionan en este código:

- **Inicialización y Atributos:**

La clase `Listener` se inicializa con un directorio de funciones (`dir_func`), una tabla de variables (`var_table`) y una pila de ámbitos (`scope_stack`).

- **exitVars:**

Este método se activa al salir de un contexto de declaración de variables. Determina el ámbito de las variables (ya sea global o local a una función) y añade las variables a la tabla de variables.

- **exitStatement:**

Se activa al salir de un contexto de declaración de sentencias. Este método identifica y procesa diferentes tipos de sentencias, como asignaciones, condiciones (if-else) y ciclos (while-do). También maneja llamadas a funciones.

- **enterFuncs y exitFuncs:**

`enterFuncs` se activa al entrar en un contexto de declaración de funciones y añade el nombre de la función al `scope_stack`.

exitFuncs se activa al salir de un contexto de declaración de funciones. Verifica si la función ya ha sido declarada y, si no es así, la añade al directorio de funciones. También gestiona las variables locales de la función.

- **exitAssign:**

Se activa al salir de un contexto de asignación. Verifica si la variable a la que se asigna ha sido declarada previamente.

- **is_var_declared:**

Es una función auxiliar que verifica si una variable ha sido declarada en el ámbito actual o en cualquier ámbito externo.



Manejo de Errores

El código utiliza el manejo de errores para gestionar situaciones como declaraciones múltiples de funciones y asignaciones a variables no declaradas. Cuando se detecta uno de estos errores, se lanza una excepción `ValueError` con un mensaje descriptivo. De igual manera se detectan errores en runtime como división con 0.

Generación de cuádruplos

La clase `Visitor`, derivada de “`BabyDuckVisitor`”, es una pieza central en la implementación del compilador para el lenguaje `BabyDuck`, encargada de recorrer el árbol de análisis sintáctico generado por `ANTLR4`. Su función principal es la generación y manejo de cuádruplos, que son estructuras de datos utilizadas para representar instrucciones intermedias en la compilación de programas.

Los cuádruplos son el corazón de la generación de código intermedio, y la clase `Visitor` los maneja a través de una serie de listas que actúan como pilas (*stacks*), específicamente para operandos, operadores, tipos y saltos. Cada cuádruplo consta de cuatro partes: un operador, dos operandos y un resultado. Por ejemplo, la expresión aritmética $a + b$ se traduciría en un cuádruplo como $(+, a, b, t1)$, donde $t1$ es una variable temporal que almacena el resultado de la operación.

La clase `Visitor` implementa métodos para visitar diferentes nodos del árbol sintáctico, como expresiones, factores y términos, y en cada visita, realiza las siguientes acciones:

1. **Evaluación de Expresiones:** Al visitar expresiones, la clase determina la operación a realizar y empuja operandos y operadores a sus respectivas pilas. Si la expresión es compleja, descompone la operación en cuádruplos más simples, respetando la precedencia de operadores.
2. **Generación de Cuádruplos:** Cuando se encuentra con operadores, la clase `Visitor` genera cuádruplos. Si es necesario, crear variables temporales para almacenar resultados intermedios.
3. **Manejo de Tipos:** La clase verifica la compatibilidad de tipos utilizando un "cubo semántico", que define las operaciones válidas entre tipos de datos. Si los tipos no son compatibles, se lanza un error.

4. **Control de Flujo:** Para estructuras de control como `if` y `while`, la clase `Visitor` utiliza cuádruplos especiales que manejan los saltos condicionales y los bucles, actualizando la pila de saltos según sea necesario.
5. **Optimización de Saltos:** Antes de finalizar, la clase `Visitor` revisa y ajusta los objetivos de salto en los cuádruplos para asegurarse de que apunten a las posiciones correctas del código intermedio.

Puntos neurálgicos en cuádruplos

Los puntos neurálgicos en la clase **Visitor** son aquellos lugares donde se toman decisiones críticas para la generación de cuádruplos y el manejo del flujo de control del programa. Aquí hay una descripción de cada uno de ellos:

1. **Generación de Temporales (`new_temporary`):** Este método es crucial porque crea nuevas variables temporales que se utilizan para almacenar resultados intermedios de expresiones. Cada vez que se necesita un resultado intermedio, se genera un nuevo temporal para mantener la claridad y la secuencia del código intermedio.
2. **Comprobación de Tipos (`check_semantic_cube`):** Aquí se verifica la compatibilidad de tipos para las operaciones. Utiliza el cubo semántico para determinar si una operación entre dos tipos de datos con un operador específico es válida y qué tipo de dato resultará de esa operación.
3. **Procesamiento de Operadores (`process_operator`):** Este punto es donde se consumen los operadores de la pila y se generan cuádruplos basados en la precedencia de operadores. Se toman los operandos de sus pilas, se aplica el operador y el resultado se almacena en una variable temporal.
4. **Generación de Cuádruplos (`generate_quadruple`):** En este método se construyen los cuádruplos propiamente dichos y se añaden a la lista de cuádruplos. Es un punto neurálgico porque cada cuádruplo es una instrucción que luego será ejecutada por la máquina o transformada en código de máquina.
5. **Visita de Factores (`visitFactor`):** Al visitar un factor, que puede ser un identificador, una constante o una expresión entre paréntesis, este método decide cómo manejarlo y dónde colocar el resultado.
6. **Visita de Expresiones (`visitExp`):** Este método maneja las expresiones que pueden contener múltiples términos y operadores. Se asegura de que las operaciones se realicen en el orden correcto, respetando la precedencia de operadores y generando los cuádruplos necesarios.
7. **Visita de Términos (`visitTermino`):** Similar a `visitExp`, pero para términos que pueden ser multiplicados o divididos. También gestiona la precedencia de operadores y genera cuádruplos para estas operaciones.

8. Asignación (visitAssign): Aquí se manejan las asignaciones, donde se evalúa la expresión del lado derecho y se genera un cuádruplo para asignar el resultado al identificador del lado izquierdo.

9. Condiciones (visitCondition): Este método maneja las estructuras de control **if-else**. Genera cuádruplos para evaluar la condición y para saltar al código correspondiente según si la condición es verdadera o falsa.

10. Ciclos (visitCycle): Maneja los bucles **while**, generando cuádruplos para la evaluación de la condición del bucle y para asegurar que el bucle continúe ejecutándose mientras la condición sea verdadera.

11. Impresión (visitPrint): Genera cuádruplos para las operaciones de impresión, asegurando que los valores o cadenas a imprimir se manejen correctamente.

12. Llamadas a Funciones (visitFCall): Este punto maneja las llamadas a funciones, pasando argumentos y gestionando el valor de retorno a través de cuádruplos.

13. Corrección de Objetivos de Salto (fix_jump_targets): Al final del proceso, este método ajusta los objetivos de salto en los cuádruplos para que apunten a las posiciones correctas, lo cual es esencial para el control de flujo correcto del programa.

A continuación, se muestra un ejemplo del resultado de la generación de cuádruplos:

```
0: ['GOTO', None, None, 'L8', 'Global']
1: ['*', '1', '1', '_t0', 'emptyFunction1']
2: ['+', '1', '1', '_t1', 'emptyFunction1']
3: ['/', '_t0', '_t1', '_t2', 'emptyFunction1']
4: ['+', '1', '_t2', '_t3', 'emptyFunction1']
5: ['=', '_t3', None, 'number1', 'emptyFunction1']
6: ['>', 'number1', '0', '_t4', 'emptyFunction1']
7: ('GOTO_F', '_t4', None, 'L3', None)
8: ['PRINT', 'number1', None, 'P7', 'emptyFunction1']
9: ['PRINT', '"Number 1 es mas grande "', None, 'P7',
10: 'emptyFunction1', None, 'L4', None)
11: ['PRINT', 'number1', None, 'P7', 'emptyFunction1']
12: ['PRINT', '"Number 1 es mas chico "', None, 'P7',
13: 'emptyFunction1', None, 'number1', 'emptyFunction2']
14: ['=', '3.0', None, 'number2', 'emptyFunction2']
15: ['LABEL', 'L5', None, 'P7', 'emptyFunction2']
16: ['>', 'number1', 'number2', '_t5', 'emptyFunction2']
17: ['GOTO_F', '_t5', None, 'P7', 'emptyFunction2']
18: ['+', 'number1', '1', '_t6', 'emptyFunction2']
19: ['=', '_t6', None, 'number1', 'emptyFunction2']
20: ['GOTO', 'L5', None, 'P7', 'emptyFunction2']
21: ['LABEL', 'P7', None, 'P7', 'emptyFunction2']
22: ['PARAM', '2', None, None, 'Global']
23: ['PARAM', '3', None, None, 'Global']
24: ['CALL', 'emptyFunction2', 2, None, 'Global']
25: ['PARAM', '2', None, None, 'Global']
26: ['PARAM', '3', None, None, 'Global']
27: ['CALL', 'emptyFunction2', 2, None, 'Global']
28: ['END', None, None, None, 'Global']
```

Formato de los cuádruplos

El formato de cuádruplos es fundamental para la construcción y ejecución del programa. Idealmente los cuádruplos son únicamente 4 elementos, sin embargo, el sistema implementado utiliza elementos de longitud variable estando entre 3 elementos hasta 5.

- Operador
- Operando izquierda
- Operando derecha
- Resultado
- Scope
- Tipo de variable del target

El archivo `vm.py` del repositorio `Baby_Duck` contiene la clase `VirtualMachine`, que se encarga de la ejecución de cuádruplos. Los cuádruplos son una estructura de datos utilizada en la generación de código intermedio en compiladores. Cada cuádruplo representa una instrucción o una operación y generalmente consta de cuatro partes:

- **Operador:** Indica la operación a realizar (por ejemplo, suma, resta, asignación, salto condicional, etc.).
- **Operando 1 y Operando 2:** Son los operandos sobre los cuales se realiza la operación. Pueden ser valores constantes, direcciones de memoria, o resultados de operaciones anteriores.
- **Resultado:** Es el lugar donde se almacenará el resultado de la operación. Puede ser una dirección de memoria o un registro.

En el contexto de `VirtualMachine`, los cuádruplos se manejan de la siguiente manera:

- **Inicialización:** La máquina virtual recibe una lista de cuádruplos (`self.quadruples`) y varias tablas de variables y funciones. Estos cuádruplos son generados durante la fase de análisis semántico del compilador y son pasados a la máquina virtual para su ejecución.
- **Ejecución:** La máquina virtual itera sobre la lista de cuádruplos y ejecuta cada operación. Para esto, utiliza un puntero de instrucción (`self.instruction_pointer`) que indica el cuádruplo actual. La ejecución de un cuádruplo puede involucrar operaciones aritméticas, saltos condicionales, llamadas a funciones, entre otras.
- **Manejo de Memoria:** La clase `VirtualMachine` gestiona diferentes rangos de memoria para variables globales, locales, temporales y constantes. Cada variable o constante se asigna a una dirección de memoria específica, y esta dirección se utiliza en los cuádruplos para referenciar las variables.
- **Funciones Auxiliares:** La clase contiene varias funciones auxiliares para manejar variables, constantes, y operaciones. Por ejemplo, `translate_variable` convierte un nombre de variable en su dirección de memoria correspondiente, y `getConstantType` determina el tipo de una constante.

Máquina virtual

La máquina virtual en el código del repositorio Baby_Duck es una implementación compleja que maneja la ejecución de cuádruplos generados por el compilador. Aquí hay un desglose detallado de su funcionamiento:

Inicialización

- La máquina virtual (VirtualMachine) se inicializa con cuádruplos, tablas de variables, funciones, tipos de variables y una pila de operandos.
- Se mantienen varias estructuras de datos para manejar variables globales, temporales, constantes y marcadores de posición (placeholders).
- Se definen rangos de direcciones de memoria para diferentes tipos de variables (enteros, flotantes, booleanos, etc.) y ámbitos (globales, locales).

Funcionamiento

- La máquina virtual traduce los cuádruplos a cuádruplos de memoria, cambiando las variables, constantes y labels a direcciones de memoria para una ejecución más eficiente.
- La máquina virtual procesa cada cuádruplo en orden, utilizando un puntero de instrucción.
- Las operaciones incluyen asignaciones, operaciones aritméticas, saltos condicionales e incondicionales, llamadas a funciones, etc.
- Las variables se manejan mediante direcciones de memoria. La máquina virtual puede traducir nombres de variables a direcciones de memoria y viceversa.

Manejo de Variables y Constantes

- Las variables y constantes se almacenan en diferentes rangos de memoria.
- Las direcciones de memoria para variables temporales y constantes se asignan dinámicamente.
- La máquina virtual puede determinar el tipo de una constante (entero, flotante, cadena, booleano, marcador de posición).

Ejecución de Cuádruplos

- La máquina virtual ejecuta cuádruplos uno por uno, realizando la operación especificada.
- Las operaciones pueden ser aritméticas, lógicas, de control de flujo, etc.
- La ejecución de cuádruplos implica leer y escribir en la memoria, realizar cálculos y actualizar el puntero de instrucción.

Funciones Específicas

- La máquina virtual maneja llamadas a funciones, pasando parámetros y gestionando el ámbito de las variables.
- Implementa operaciones especiales como impresión (print), asignación de direcciones a variables temporales y constantes.

Depuración y Visualización

- La máquina virtual tiene capacidades de depuración para imprimir tablas de variables, pilas y valores almacenados en memoria.

Manejo de memoria

El manejo de memoria en la máquina virtual del repositorio Baby_Duck se realiza a través de la clase `VirtualMachine` definida en el archivo `vm.py`. Esta clase gestiona la asignación de direcciones de memoria para variables, constantes y temporales, así como la ejecución de cuádruplos. A continuación, se detalla cómo funciona este manejo de memoria:

Asignación de Direcciones de Memoria

La máquina virtual utiliza diferentes rangos de direcciones de memoria para distintos tipos de datos y ámbitos (global y local). Estos rangos se definen al inicio de la clase `VirtualMachine`:

- **Variables Temporales Globales y Locales:** Se asignan rangos específicos para enteros, flotantes y booleanos tanto en el ámbito global como local. Por ejemplo, `self.temporal_ints_global` y `self.temporal_ints_local` para enteros temporales globales y locales, respectivamente.
- **Constantes:** Se asignan rangos para enteros, flotantes, booleanos y cadenas de texto. Por ejemplo, `self.constant_ints` para constantes enteras.
- **Placeholders:** Se asigna un rango para placeholders, utilizados para etiquetas y posiciones en cuádruplos.

Funciones de Asignación

La clase `VirtualMachine` contiene métodos para asignar direcciones de memoria a variables y constantes:

- **assignAddressTemporals:** Asigna una dirección de memoria a una variable temporal basada en su tipo y ámbito (global o local). Por ejemplo, si se trata de un entero temporal en el ámbito global, se selecciona la primera dirección disponible en `self.temporal_ints_global` y se actualiza el rango.
- **assignAddressConstants:** Asigna una dirección de memoria a una constante. El tipo de la constante (entero, flotante, booleano, cadena o placeholder) determina el rango de direcciones a utilizar.

Manejo de Tipos de Constantes

- **getConstantType:** Determina el tipo de una constante (entero, flotante, cadena, booleano o placeholder) basándose en su valor. Utiliza expresiones regulares y comprobaciones de tipo para clasificar la constante.

Traducción y Detección de Variables

- **translate_variable:** Traduce el nombre de una variable a su correspondiente dirección de memoria. Busca en las tablas de variables globales y locales.
- **detranslate_variable:** Dado una dirección de memoria, devuelve el nombre de la variable asociada. Busca en las tablas de variables globales y locales.

Ejecución de Cuádruplos

La máquina virtual ejecuta cuádruplos (instrucciones de cuatro partes) para realizar operaciones. Cada cuádruplo contiene un operador y hasta tres operandos. La clase VirtualMachine interpreta y ejecuta estos cuádruplos, gestionando la memoria según sea necesario.

Rangos de memoria

Rango de Memoria	Tipo de Dato	Ámbito	Tupla de Rango de Memoria (Inicio, Fin)
self.temporal_ints_global	Enteros Temporales	Global	(8192, 9215)
self.temporal_floats_global	Flotantes Temporales	Global	(9216, 10240)
self.temporal_bools_global	Booleanos Temporales	Global	(10241, 11263)
self.temporal_ints_local	Enteros Temporales	Local	(11264, 12287)
self.temporal_floats_local	Flotantes Temporales	Local	(12288, 13312)
self.temporal_bools_local	Booleanos Temporales	Local	(13313, 14335)
self.constant_ints	Constantes Enteras	Constante	(14336, 15359)

self.constant_floats	Constantes Flotantes	Constante	(15360, 16383)
self.constant_bools	Constantes Booleanas	Constante	(16384, 17407)
self.constant_strings	Constantes de Cadena	Constante	(17408, 18431)
self.placeholders	Placeholders	Especial	(18432, 19455)

Salto con labels y placeholders

- **GOTO:** Este salto es incondicional. Cambia el puntero de instrucción a la dirección especificada en el cuádruplo. No evalúa ninguna condición.
- **GOTO_F y GOTO_T:** Estos saltos son condicionales. **GOTO_F** se utiliza para saltar a una dirección específica si la condición evaluada es falsa, mientras que **GOTO_T** se utiliza para saltar si la condición es verdadera. La condición se evalúa basándose en el valor de una variable o expresión booleana.
 - Para **GOTO_F** y **GOTO_T**, la máquina virtual primero evalúa la condición (que puede ser el resultado de una operación lógica o relacional). Si la condición para **GOTO_F** es falsa o para **GOTO_T** es verdadera, entonces el puntero de instrucción se actualiza a la dirección especificada en el cuádruplo. Si no, la ejecución continúa secuencialmente.
- **LABEL:** Este no es un salto en sí, sino una marca o etiqueta utilizada para identificar una posición específica en el código. Se utiliza en combinación con **GOTO**, **GOTO_F** y **GOTO_T** para dirigir el flujo del programa a puntos específicos. Las etiquetas se almacenan en un diccionario (`placeholders_positions`) con el nombre de la etiqueta como clave y la posición en los cuádruplos como valor.

La lógica de estos saltos se implementa dentro de un bucle que itera sobre los cuádruplos y ejecuta las instrucciones correspondientes. El puntero de instrucción (`instruction_pointer`) se actualiza según el tipo de salto y la condición evaluada.

En resumen, **GOTO** se usa para saltos incondicionales, **GOTO_F** y **GOTO_T** para saltos condicionales basados en el resultado de una evaluación booleana, y **LABEL** para marcar posiciones específicas en el código a las que se puede saltar.

Clases y paquetes del programa

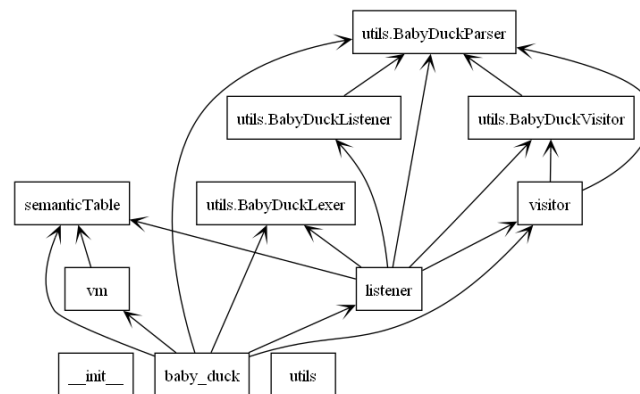
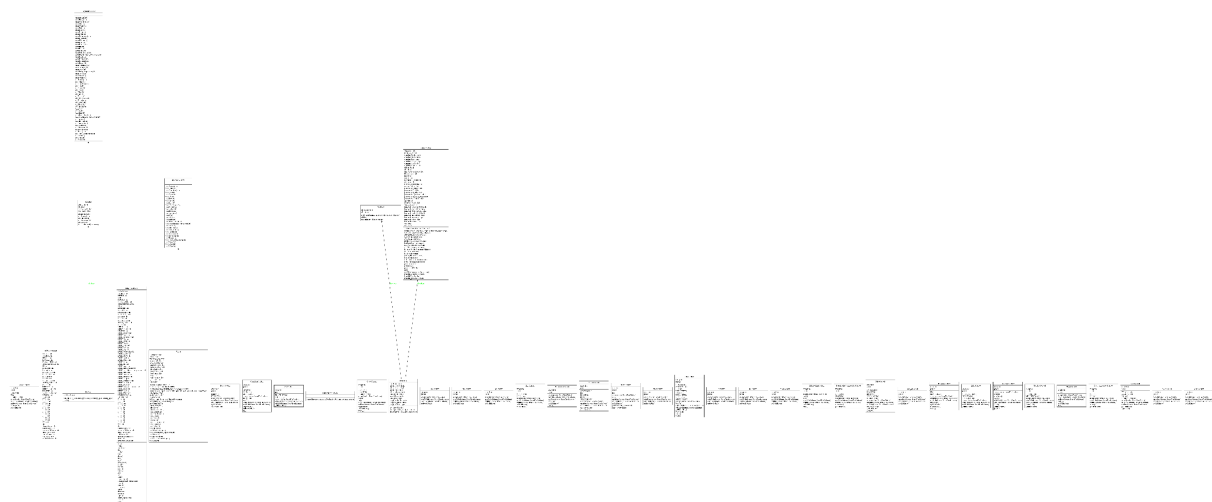


Diagrama de conexión de los paquetes y de la interdependencia entre bloques de código y clases.



baby_duck.py: este es el archivo principal del proyecto. Incluye la función principal e integra varios componentes como el lexer, el analizador, el detector de errores, el visitante y la máquina virtual.

listener.py: este archivo define una clase de escucha que se hereda de `BabyDuckListener`. Contiene métodos para procesar diferentes partes del árbol de análisis.

semanticTable.py: este archivo contiene definiciones relacionadas con el análisis semántico, como tablas de símbolos o funcionalidades de verificación de tipos.

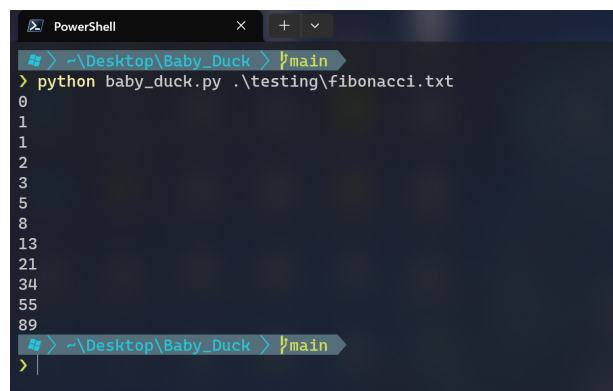
visitante.py: este archivo define una clase de visitante para atravesar el árbol de análisis.

vm.py: este archivo contiene la implementación de una máquina virtual para ejecutar el código analizado.

- `utils/BabyDuckLexer.py`, `utils/BabyDuckListener.py`, `utils/BabyDuckParser.py`, `utils/BabyDuckVisitor.py`: estos archivos en el directorio `utils` son parte del lexer y analizador generado por ANTLR para el lenguaje Baby Duck.

Ejecución del código

Para ejecutar el código es necesario haber instalado las librerías listadas en el `requirements.txt` que se encuentra en el repositorio. El código recibe dos parámetros, el archivo de entrada y la bandera de debugging. Por defecto esta bandera está en falso, sin embargo, si se desea observar que sucede a nivel memoria, variables y código en general es recomendable poner esta bandera como verdadera. La bandera de flag también permite ver cómo se van actualizando las variables temporales, globales y cómo se ejecutan los saltos. De igual manera, este modo permite ver los cuádruplos tanto en órdenes de ejecución como en órdenes a nivel memoria.

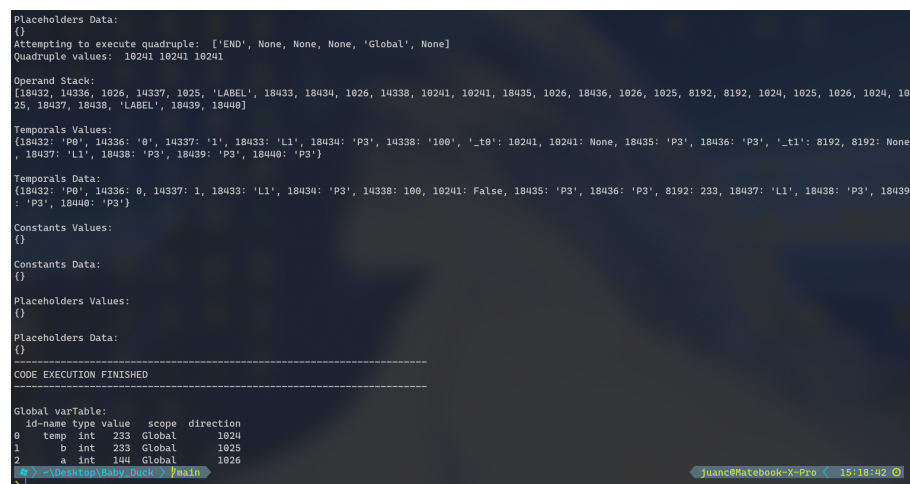


```

PowerShell
> -\Desktop\Baby_Duck > |main
> python baby_duck.py .\testing\fibonacci.txt
0
1
1
2
3
5
8
13
21
34
55
89
> -\Desktop\Baby_Duck > |main
>

```

Ejecución en modo regular



```

Placeholders Data:
{}
Attempting to execute quadruple: ['END', None, None, None, 'Global', None]
Quadruple values: 10241 10241 10241
Operand Stack:
[18432, 14336, 1026, 14337, 1025, 'LABEL', 18433, 18434, 1026, 14338, 10241, 10241, 18435, 1026, 18436, 1026, 1025, 8192, 8192, 1024, 1025, 1026, 1024, 1025, 18437, 18438, 'LABEL', 18439, 18440]
Temporals Values:
{18432: 'P0', 14336: '0', 14337: '1', 18433: 'L1', 18434: 'P3', 14338: '100', '_t0': 10241, 10241: None, 18435: 'P3', 18436: 'P3', '_t1': 8192, 8192: None, 18437: 'L1', 18438: 'P3', 18439: 'P3', 18440: 'P3'}
Temporals Data:
{18432: 'P0', 14336: 0, 14337: 1, 18433: 'L1', 18434: 'P3', 14338: 100, 10241: False, 18435: 'P3', 18436: 'P3', 8192: 233, 18437: 'L1', 18438: 'P3', 18439: 'P3', 18440: 'P3'}
Constants Values:
{}
Constants Data:
{}
Placeholders Values:
{}
Placeholders Data:
{}
CODE EXECUTION FINISHED
Global varTable:
id-name type value scope direction
0 temp int 233 Global 1024
1 b int 233 Global 1025
2 a int 144 Global 1026
> -\Desktop\Baby_Duck > |main
>

```

Ejecución en modo debug, permitiendo ver los valores de temporales, placeholders, saltos, operand stack, tabla de variables, cuádruplos, entre otros

Pruebas de ejecución

Con el fin de probar el correcto funcionamiento del código se realizaron varios programas de prueba para verificar el correcto funcionamiento del compilador y de la máquina virtual. Cada uno de estos archivos tienen como objetivo el probar una funcionalidad núcleo del proyecto. El código está en gran parte preparado para comenzar el desarrollo de funciones. Sin embargo, se ha optado por dejar a un lado esta funcionalidad y dejarlo para futuras iteraciones del compilador.

Código	Resultado	Resultado obtenido
factorial.txt	120	120
fibonacci.txt	0 1 1 2 3 5 8 13 21 34 55 89	0 1 1 2 3 5 8 13 21 34 55 89
conditions.txt	foo is not 1	foo is not 1
cycle.txt	3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0	3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
nestedif.txt	foo is not 1 2ndloop foo is not 1	foo is not 1 2ndloop foo is not 1
noConditions.txt	3 3 5 The quick brown fox jumps over the lazy dog !@#\$\$%^&*()_+][\'	3 3 5 The quick brown fox jumps over the lazy dog !@#\$\$%^&*()_+][\'
suma.txt	4	4

Conclusión

A lo largo de este proyecto, he aprendido sobre cómo funciona un compilador por dentro, especialmente sobre la generación y el manejo de cuádruplos, así como sobre lo importante que es el cubo semántico para el análisis semántico. Esta experiencia me ha ayudado a crecer como programador e ingeniero al darme una comprensión más profunda de los principios de programación y compilación.

Entender cómo se traduce el código de alto nivel a instrucciones que la máquina puede ejecutar requiere investigar cómo funciona un compilador, especialmente a través de la implementación de un cubo semántico y la generación de cuádruplos. Gracias a esta comprensión, puede escribir código más eficiente y optimizado y diagnosticar y solucionar problemas relacionados con la compilación y ejecución de programas.

El uso del cubo semántico en mi proyecto ha sido crucial para garantizar la correcta interpretación semántica de las operaciones entre diferentes tipos de datos. Esta herramienta me ha permitido manejar de manera efectiva las complejidades asociadas con la compatibilidad de tipos y las conversiones de tipo, lo cual es fundamental en el diseño de lenguajes de programación y compiladores.

La gestión de la memoria, especialmente a través del uso de pilas para almacenar operandos, operadores y cuádruplos, ha sido un aspecto clave de este proyecto. Comprender cómo se almacena y se accede a la información en la memoria durante la ejecución de un programa me ha proporcionado valiosas perspectivas sobre la optimización del rendimiento y la eficiencia del código. Especialmente a nivel de memoria y de sus rangos.

Este proyecto ha mejorado significativamente mis habilidades de programación y mi pensamiento ingenieril. La capacidad de analizar y manipular la estructura interna de un programa me permite abordar problemas complejos con mayor confianza y eficiencia. Además, la experiencia adquirida en la gestión de tipos de datos, la memoria y la generación de código intermedio es invaluable para mi futuro desarrollo profesional en el campo de la informática.

El haber logrado un código que permita interpretar a otros códigos y realizar la compilación me pareció una tarea desafiante, que al haberla completado me ha dejado de muchos aprendizajes y nuevas ideas que me hacen entender aún más el funcionamiento de las computadoras, valorar el código limpio y a buscar crear código que esté optimizado, especialmente ahora que he logrado entender que sucede al momento de ejecutar un código de programación para cualquier lenguaje.

En resumen, este proyecto no solo ha fortalecido mis habilidades técnicas en programación y compilación, sino que también ha enriquecido mi comprensión teórica, preparándome para enfrentar desafíos más complejos en el mundo de la ingeniería de software y el desarrollo de sistemas.

Referencias

- ANTLR. (n.d.). ANTLR. Retrieved from <https://www.antlr.org/>
- (n.d.). Universidad Nacional del Centro de la Provincia de Buenos Aires. Retrieved from <https://users.exa.unicen.edu.ar/catedras/ccomp1/Apunte5.pdf>
- (n.d.). Presentación sobre gramáticas y lenguajes libres de contexto.. Retrieved from https://libroweb.alfaomega.com.mx/book/685/free/ovas_statics/cap9/Presentacion%20sobre%20gramaticas%20y%20lenguajes%20libres%20de%20contexto..pdf
- Garshol, L. M. (n.d.). BNF and EBNF: What are they and how do they work? Retrieved from <https://www.garshol.priv.no/download/text/bnf.html>
- Parr, T. (2013). The Definitive ANTLR 4 Reference. The Pragmatic Programmers.
- Tomassetti, F. (n.d.). Listeners and Visitors. Retrieved from <https://tomassetti.me/listeners-and-visitors/>
- Coding Ninjas. (n.d.). Memory Management in Compiler Design. Retrieved from <https://www.codingninjas.com/studio/library/memory-management-in-compiler-design>
- OpenGenus Foundation. (n.d.). Memory Allocation. Retrieved from <https://iq.opengenus.org/memory-allocation/>
- GeeksforGeeks. (n.d.). Intermediate Code Generation in Compiler Design. Retrieved from <https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/>

Anexos

Repositorio: https://github.com/SeaWar741/Baby_Duck