

*Luis Fernando Lomelín Ibarra*

---

# ***IntroProg el lenguaje estadístico para aprender a programar***



*Gracias a mi familia. A mi mamá y a mi hermano por apoyarme mucho durante todo este tiempo. Sin ellos realmente no sabría en donde estaría ahora.*

*Gracias a mis amigos. Por estar allí en los momentos buenos y malos. Por hacer la vida más interesante*

*Gracias a mis profesores. Por pasar su conocimiento con paciencia y entusiasmo, y por darme la oportunidad de hacer grandes cosas*



# ***Índice general***

<b>I</b>	<b>Descripción del proyecto y lenguaje</b>	<b>1</b>
<b>1.</b>	<b>Descripción del proyecto</b>	<b>3</b>
1.1.	Propósito y Alcance del Proyecto . . . . .	3
1.2.	Análisis de Requerimientos y descripción de los principales Test Cases . . . . .	4
1.3.	Descripción del proceso general de desarrollo . . . . .	5
1.3.0.1.	Semana 1: Avance del Léxico Sintaxis . . . . .	6
1.3.1.	Semana 2: Generación de la tabla de variables y el cubo semántico . . . . .	6
1.3.2.	Semana 3: Generar cuádruplos de expresiones . . . . .	7
1.3.3.	Semana 4: Código Intermedio para Condiciones y Bucles . . . . .	7
1.3.4.	Semana 5: Código Intermedio para Módulos (Funciones) . . . . .	8
1.3.5.	Semana 6: Máquina Virtual, Mapa de Memoria y ejecución de expresiones . . . . .	8
1.3.6.	Semana 7: Arreglos, Matrices y . . . . .	9
1.3.7.	Semana 8: Correcciones y Empuje final de documentación . . . . .	9
<b>2.</b>	<b>Descripción de IntroProg</b>	<b>11</b>
2.1.	Introducción a IntroProg . . . . .	11
2.2.	Características Principales . . . . .	12
2.2.1.	Estructura general de un programa de IntroProg . . . . .	12
2.2.2.	Variables y la declaración de variables . . . . .	13
2.2.3.	Expresiones, Imprimir, Asignación y Comentarios . . . . .	14
2.2.4.	Condiciones . . . . .	16
2.2.4.1.	Estructura si . . . . .	16
2.2.4.2.	Estructura si-sino . . . . .	17
2.2.5.	Bucles . . . . .	17
2.2.5.1.	Estructura Mientras . . . . .	17
2.2.5.2.	Estructura Por . . . . .	18
2.2.6.	Funciones . . . . .	19
2.2.6.1.	Declaración de función . . . . .	19
2.2.6.2.	Llamada de función . . . . .	20
2.2.6.3.	Llamadas recursivas en funciones . . . . .	21
2.2.7.	Arreglos, Matrices y Cubos . . . . .	21

2.2.7.1. Matrices y Cubos . . . . .	22
2.2.8. Funciones Especiales . . . . .	23
2.3. Listado de errores . . . . .	25

## II Descripción del compilador y la máquina virtual 27

### 3. Descripción del compilador 29

3.1. Equipo de cómputo, lenguaje y utilerías especiales usadas en el desarrollo del proyecto . . . . .	30
3.2. Descripción del Análisis de Léxico . . . . .	30
3.3. Descripción del Análisis de Sintaxis . . . . .	32
3.4. Descripción de Generación de Código Intermedio y Análisis Semántico . . . . .	36
3.4.0.1. Direcciones de Memoria Virtuales . . . . .	36
3.4.0.2. Códigos Intermedios (Cuadрупlos) . . . . .	37
3.4.0.3. Cubo Semántico . . . . .	39
3.4.0.4. Diagramas de Sintaxis y Acciones Semánticas . . . . .	41
3.5. Descripción de Administración de Memoria usado en la compilación . . . . .	62
3.5.0.1. Directorio de Funciones y el Directorio de Funciones Especiales . . . . .	62
3.5.0.2. Vector de Cuadрупlos y su contador . . . . .	63
3.5.0.3. Tabla de Constantes y su espejo . . . . .	64
3.5.1. Tabla de variables . . . . .	67
3.5.1.1. Cubo Semántico . . . . .	67
3.5.1.2. Contadores Globales de Variables Globales y temporales, Máximos de Variables y las Direcciones de Memoria . . . . .	71
3.5.1.3. Vector de relación línea código fuente a cuádruplo . . . . .	71
3.5.1.4. Variables auxiliares para el manejo de arreglos . . . . .	72
3.5.1.5. Pila de operadores, Pila de operandos y Pila de tipos . . . . .	72

### 4. Descripción de la máquina virtual 75

4.1. Equipo de cómputo, lenguaje y utilerías especiales usadas . . . . .	75
4.2. Descripción del proceso de Administración de Memoria en ejecución . . . . .	76
4.2.1. La Clase Memoria . . . . .	76
4.2.2. Manejo de memoria global, local y temporal . . . . .	76
4.2.3. El stack de memorias . . . . .	77
4.3. Constantes . . . . .	78

## III Pruebas del funcionamiento del lenguaje 79

<i>Contents</i>	IX
-----------------	----

<b>5. Pruebas del funcionamiento del lenguaje</b>	<b>81</b>
5.1. Programa Factorial . . . . .	81
5.2. Programa Fibonacci . . . . .	83
5.3. Programa Búsqueda y organización de arreglo . . . . .	85
5.4. Programa Multiplicación de Matrices . . . . .	88
5.5. Programa Uso de Funciones de estadística . . . . .	90

<b>IV Ejemplos de Documentación en código</b>	<b>93</b>
---	-----------





## Parte I

# Descripción del proyecto y lenguaje



# 1

## *Descripción del proyecto*

### CONTENTS

1.1	Propósito y Alcance del Proyecto .....	3
1.2	Análisis de Requerimientos y descripción de los principales Test Cases .....	4
1.3	Descripción del proceso general de desarrollo .....	5
1.3.0.1	Semana 1: Avance del Léxico Sintaxis .....	6
1.3.1	Semana 2: Generación de la tabla de variables y el cubo semántico .....	6
1.3.2	Semana 3: Generar cuádruplos de expresiones .....	7
1.3.3	Semana 4: Código Intermedio para Condiciones y Bucles .....	7
1.3.4	Semana 5: Código Intermedio para Módulos (Funciones) .....	8
1.3.5	Semana 6: Máquina Virtual, Mapa de Memoria y ejecución de expresiones .....	8
1.3.6	Semana 7: Arreglos, Matrices y .....	9
1.3.7	Semana 8: Correcciones y Empuje final de documentación .....	9

Una breve descripción del proyecto, describiendo su alcance al igual que su propósito.

### 1.1. Propósito y Alcance del Proyecto

El propósito de este lenguaje es ser un lenguaje introductorio a la lógica de programación para personas que no están muy familiarizadas con la programación y que no tienen una buena comprensión del inglés. Para lograr esto el lenguaje fue diseñado a ser similar a C/C++ en estructura, pero se utilizan palabras clave en español para facilitar el aprendizaje de programación para las personas que no tengan un buen dominio en el lenguaje. El objetivo del lenguaje es que sea un punto de entrada para chicos en prepa o alumnos de universidad a aventurarse a lenguajes con sintaxis similar a C/C++ pero que

no tienen una gran comprensión del inglés.

Con esto en mente el lenguaje busca ser una herramienta básica de programación que, aunque sea restrictiva a comparación de otros lenguajes, permita que el usuario experimente con funciones de estadística y con la lógica de programación para crear pequeños programas con los cuales pueden explorar temas de programación, ciencia de datos y estadística.

---

## 1.2. Análisis de Requerimientos y descripción de los principales Test Cases

Con el objetivo y alcance en definido, se plantearon los siguientes requerimientos que se deben de cumplir con el lenguaje.

- Crear palabras clave análogas o similares a las palabras clave del lenguaje de C/C++ en español
- Generar estructuras similares a las de C/C++ para estatutos como if y while, pero que sean legibles en español
- Permite el uso de estructuras de decisión (if, if else)
- Permite el uso de estructuras cíclicas (for, while, do while)
- Realizar operaciones matemáticas aritméticas sencillas como suma, restas, multiplicaciones y divisiones.
- Realizar operaciones lógicas y relacionales
- Poder generar funciones que reciban uno o más parámetros y que puedan regresar un resultado
- Una forma integrada en el lenguaje de programación para leer e imprimir resultados de la consola
- Manejo de variables locales y globales
- Manejo de arreglos integrados al lenguaje de programación
- Creación de números aleatorios en ciertas distribuciones integrados en el lenguaje de programación.
- Funciones integradas de las métricas centrales estadísticas (media, mediana, moda y varianza)
- Funciones de matemáticas integradas al sistema (exp, pow, sqrt entre otras)

- Manejo de vectores y matrices

Con estos requerimientos, también se ponderaron las limitaciones que iban a tener el lenguaje dado el corto tiempo y la falta de experiencia en este tipo de proyectos. Y se definieron las siguientes limitantes:

- Solo se cuentan con un tiempo relativamente corto (Alrededor de 6 semanas)
- Es un proyecto individual, aunque se quiera expandir más el alcance del proyecto se tendrá que limitar y ajustar el trabajo a lo que es posible
- El lenguaje va a ser restrictivo para que sea posible hacerlo en el tiempo dado. Esto quiere decir que la implementación de múltiples scopes dentro del lenguaje esta fuera del alcance del proyecto

Con estos requerimientos y limitantes en mente, se diseñaron las siguientes pruebas las cuales debería de ser posible hacerse en el lenguaje.

- Se deberá poder programar un Hola Mundo en el lenguaje
- Se deberá poder programar una implementación secuencial y recursiva de la función factorial
- Se deberá poder programar una implementación secuencial y recursiva de la serie de Fibonacci
- Se deberá poder programar una implementación de un Bubble Sort y un Search en un arreglo
- Se deberá poder programar una multiplicación de matrices
- Se deberá poder hacer un programa que genere un arreglo que todos sus elementos pertenezcan a una distribución y que se le pueda hacer un análisis (aunque sea superficial) a los datos generados.

Una vez definido los requerimientos, las limitantes y las pruebas se empezó con el desarrollo del proyecto.

---

### 1.3. Descripción del proceso general de desarrollo

A lo largo del desarrollo del proyecto se estuvieron haciendo avances semanales en el compilador y la máquina virtual. Como medida de seguridad, se uso la herramienta de Git y Github para mantener un historial de control sobre los cambios hechos al proyecto. El repositorio de IntroProg puede ser encontrado en la liga <https://github.com/LuisLomelinIbarra/Compiladores-2022>. A pesar

de mantener un control de versiones con Git, también se mantuvo una bitácora de desarrollo. A continuación se encuentra una bitácora de dichos avances en orden cronológico.

#### **1.3.0.1. Semana 1: Avance del Léxico Sintaxis**

Por el momento se realizó el lexer y parser con los diagramas en la propuesta. En este avance se realizó un programa de python que puede leer y aceptar (o rechazar) un archivo en el lenguaje de programación propuesto. Se realizaron los primeros dos archivos de prueba que son lenguaje valido. Estos son un helloworld y un archivo con operaciones y una función de factorial. En las pruebas realizadas regresa aceptado para los dos archivos. Faltan hacer más archivos de prueba para ver si se encuentran errores en el programa o si se debe de hacer un cambio a la gramática (en un caso extremo).

Actualmente el programa debe de:

- Compilar sin errores
- Leer un archivo y decidir si es o no el lenguaje
- Indicar la línea del error si se encontró alguno

#### **1.3.1. Semana 2: Generación de la tabla de variables y el cubo semántico**

Se avanzo con la generación del directorio de funciones y la tabla de variables. Además, se creó el cubo semántico para las operaciones del lenguaje. De igual manera se empezó a implementar el análisis semántico de las operaciones del lenguaje. Al realizar esto se tuvo que reestructurar algunas reglas de gramática para acomodar mejor los puntos neurálgicos. También implementando el análisis se encontraron los siguientes problemas.

1. Se podía generar arreglos de manera infinita ya que estaban al nivel de varcte. Como los arreglos textuales se pueden generar con expresiones, se podía anidar arreglos textuales en arreglos textuales. Para resolver esto se reestructuro la gramática para que solo se puedan utilizar en las asignaciones, las llamadas a función y las impresiones
2. Hay que experimentar más con la verificación de los parámetros en una llamada a función. Esto hay que revisar que no rompa con el orden de las operaciones y probablemente hay que verlo mejor. En este avance no se pudo aplicar.
3. No se sabe todavía cómo manejar las funciones de retorno

Actualmente el programa debe de:

- Detectar en las asignaciones que se haga la asignación correcta con respecto a el tipo de variable

- Detectar que se hagan las operaciones correctas con respecto a los tipos de los operandos en la mayoría de las operaciones
- Detectar que no haya definición múltiple de variables o arreglos
- Detectar que no haya definiciones múltiples de funciones
- Desplegar mensajes de error adecuados cuando haya errores de semántica

### 1.3.2. Semana 3: Generar cuádruplos de expresiones

En este avance ya se empezó a implementar los puntos neurálgicos de las expresiones. Actualmente, debe de poder generar el código intermedio de las expresiones en un archivo de texto llamado cuádruplos.txt. Falta probar bien todas las expresiones, pero con un archivo de pruebas rápidas con solo operaciones matemáticas parece ser que si genera bien el código. Se resolvieron algunos errores con la semántica para encontrar el tipo de un arreglo textual. También se cambió a que todos los elementos que se declaren en la función principal se declaren como su propia entrada en la entrada de principal en vez de que sea una adicción a la tabla global. También se arregló el error a la hora de declarar precedencia entre los operadores.

El avance de ahora debe de:

- Generar código intermedio de expresiones en un archivo de cuádruplos

COSAS QUE INVESTIGAR:

1. Hay que manejar bien la asignación de espacios en memoria de los cuádruplos para la generación de código intermedio. Por el momento solo va agregando de manera ascendente a un registro teóricamente infinito

### 1.3.3. Semana 4: Código Intermedio para Condiciones y Bucles

Se implemento la generación de cuádruplos para los ciclos y para las condiciones. Además, también se implementó el uso de direcciones virtuales en los cuádruplos generados. También se solucionaron ligeros errores que surgieron de algunos cambios menores a las expresiones. También ya se implementó la generación de los cuádruplos del estatuto imprimir (por fin). De igual manera se avanzaron con algunos puntos de módulos, principalmente agregar los parámetros correctos en el orden correcto a las funciones y la verificación de los tipos en las llamadas.

El avance de ahora debe de:

- Generar código intermedio en un archivo de cuádruplos, generando también los cuádruplos pertinentes a los estatutos de condición y de bucle.

**COSAS QUE INVESTIGAR:**

1. Como manejar los arreglos en memoria y como parámetros/argumentos de funciones.

**1.3.4. Semana 5: Código Intermedio para Modulos (Funciones)**

Se implemento todo lo de módulos. Para cada función ahora se va guardando la cantidad de variables que se están utilizando en ella, el cuádruplo donde inicia y si tiene retorno la dirección de la variable global de retorno. En la llamada de función se verifica que los parámetros llamados sean los correctos, de ser así se genera el cuádruplo para generar la memoria y los cuádruplos para copiar los parámetros a la nueva memoria. Al final se agrega el gosubfunc que apunta al inicio de la función llamada. En caso de ser función especial llama a el cuádruplo SPFUNC, indicando que es función especial.

El avance de ahora debe de:

- En el archivo de cuádruplos ahora ya debe de generar los cuádruplos correspondientes de cada función declarada (ERA y PARAMS). En caso de ser función especial llama al cuádruplo SPFUNC con el nombre de la función en vez de GOSUBFUNC.

**COSAS QUE CAMBIAR/ARREGLAR:**

1. Una mejor manera para manejar las constantes String
2. Corregir el error de la variable sintáctica que permite que se pueda declarar un arreglo así [1, [1,1]]

**1.3.5. Semana 6: Máquina Virtual, Mapa de Memoria y ejecución de expresiones**

Ya se empezó la máquina virtual y ya funciona leyendo los cuádruplos de expresiones, el estatuto de imprimir y con las funciones definidas por el usuario (incluyendo recursión). Para este avance implemente una clase Memory que representa la memoria. Esta clase se puede utilizar para traducir las memorias virtuales asignadas en la compilación a memoria en la máquina virtual. Para lograr esto cada scope esta representado por un vector de dos instancias de la clase. Una de ellas representa las variables y la otra representa los temporales. En esta entrega me di cuenta de varios errores en la generación de cuádruplos, específicamente en el estatuto de return estaba faltando agregar el cuádruplo de enfunc para regresar a la memoria anterior. Igual mi lógica de saltos en mi for me llevaba a contar uno ciclo demás lo cual tuve que arreglar

El avance de ahora debe de:



- El compilador debe de generar un archivo json con los cuádruplos, la tabla de funciones y las constantes.
- La máquina virtual debe de leer el json y ejecutar el código en él.

#### COSAS QUE CAMBIAR/ARREGLAR:

1. Formas de manejar los arreglos como parametros
2. Falta implementar la lógica de las funciones especiales
3. Terminar bien arreglos.

### 1.3.6. Semana 7: Arreglos, Matrices y

Ya está acabado el compilador. Actualmente el compilador recibe un archivo itp, con el lenguaje de IntroProg y genera un archivo obj (que es un json disfrazado) el cual es utilizado por la máquina virtual para ejecutar código. Es capaz de realizar el cálculo de factorial de manera secuencial y por función recursiva. También puede sacar el enésimo elemento de la secuencia Fibonacci de manera secuencial y de manera recursiva. También se puede hacer un Bubble Sort para organizar los elementos de un arreglo y se puede encontrar elementos dentro de los mismos.

También se le metió esfuerzo para que los mensajes de error sean lo más claro posibles.

El avance de ahora debe de:

- El compilador debe de generar un archivo obj que puede ser leído por la máquina virtual.
- La máquina virtual debe de leer el obj y ejecutar el código en él, incluyendo funciones especiales de estadística.

#### COSAS QUE CAMBIAR/ARREGLAR:

1. Actualizar los diagramas de sintaxis para reflejar los cambios que se hicieron durante la implementación
2. Actualizar la gramática en la documentación
3. Acabar la documentación

### 1.3.7. Semana 8: Correcciones y Empuje final de documentación

Cambios mínimos al compilador. Se refactorizo lo más que se pudo sin romper lo existente y se agregaron comentarios pertinentes a las funciones. También se agregó un token extra para el manejo de comentarios en el lenguaje.

El avance de ahora debe de:

- El compilador debe de generar un archivo obj que puede ser leído por la máquina virtual.
- La máquina virtual debe de leer el obj y ejecutar el código en él, incluyendo funciones especiales de estadística.

COSAS QUE CAMBIAR/ARREGLAR:

1. Acabar la documentación

# 2

## *Descripción de IntroProg*

### CONTENTS

2.1	Introducción a IntroProg .....	11
2.2	Características Principales .....	12
2.2.1	Estructura general de un programa de IntroProg .....	12
2.2.2	Variables y la declaración de variables .....	13
2.2.3	Expresiones, Imprimir, Asignación y Comentarios .....	14
2.2.4	Condiciones .....	16
2.2.4.1	Estructura si .....	16
2.2.4.2	Estructura si-sino .....	17
2.2.5	Bucles .....	17
2.2.5.1	Estructura Mientras .....	17
2.2.5.2	Estructura Por .....	18
2.2.6	Funciones .....	18
2.2.6.1	Declaración de función .....	19
2.2.6.2	Llamada de función .....	20
2.2.6.3	Llamadas recursivas en funciones .....	21
2.2.7	Arreglos, Matrices y Cubos .....	21
2.2.7.1	Matrices y Cubos .....	22
2.2.8	Funciones Especiales .....	23
2.3	Listado de errores .....	25

En esta sección se explica con más detalle lo que es el lenguaje y sus capacidades.

### 2.1. Introducción a IntroProg

Es un lenguaje de programación sencillo con un enfoque hacia data science. La idea del lenguaje es permitir a personas interesadas en programar a experimentar con proyectos sencillos de ciencia de datos.

IntroProg permite al usuario aprender sobre la lógica de programación, permitiendo que el usuario pueda utilizar estructuras de decisión y estructuras de repetición. También permite que los usuarios experimenten un poco

con funciones estadísticas, como generar números que pertenecen a distintas distribuciones de probabilidad como la función Normal, Exponencial, Poisson y Geométrica. Además, permite que los usuarios puedan crear gráficas de los datos con los que trabajan.

---

## 2.2. Características Principales

IntroProg cuenta con múltiples características que la mayoría de los lenguajes de programación cuentan. Estos van de estructuras de control y flujo hasta el manejo de funciones y arreglos. Como IntroProg es un lenguaje de aprendizaje puede ser un poco restrictivo, pero en esta sección se va a describir con un poco más de detalle en que consta cada una de las características del lenguaje, incluyendo las funciones especiales.

### 2.2.1. Estructura general de un programa de IntroProg

Los archivos del lenguaje son archivos de texto con terminación *itp*. Para empezar a escribir un programa en IntroProg primero se escribe la palabra clave *programa* seguido por el nombre del programa. Después entre corchetes se escribe las declaraciones de variables, seguida por las declaraciones de funciones, seguido por la función principal. La función principal es la parte del código que se ejecuta primero.

```
programa pelos{  
    // Primero las variables globales  
    entero x;  
    bool bo;  
    // El codigo principal del programa  
    principal funcion {  
        // Variables locales a principal  
        entero x,y;  
    }{  
        x = 10;  
        y = 2;  
        x = x*y;  
        imprimir(x);  
    }  
}
```

**Figura 2.1**

Ejemplo de la estructura de un programa de IntroProg

Es importante mencionar que un programa puede o no tener una declaración de variables globales y declaración de funciones. Esto dependerá de las necesidades del programador cuando construya su código.

```
programa holaMundo {  
    principal funcion {} {  
        imprimir("Hola Mundo!!!!");  
    }  
}
```

### Figura 2.2

Ejemplo de un programa sin declaración de variables ni declaración de funciones

Aunque parezca un poco confuso al principio, se van a ir explicando parte por parte la estructura y funcionalidades del lenguaje en las siguientes secciones.

#### 2.2.2. Variables y la declaración de variables

Para poder programar primero se necesitan utilizar variables. Las variables son representaciones de información que pueden ser modificadas por el programador para calcular y modelar información. Las variables en IntroProg están representadas por un identificador que es una serie de letras, guiones y números. Los identificadores siempre deben de empezar con una letra.

Para declarar una variable no solo es importante tener un buen identificador sino también hay que indicar el tipo de la variable. En IntroProg se manejan cuatro tipos de variables los cuales son:

- **Enteros:** Números enteros (ej. 1,2,3). Estos son representados por la palabra clave *entero*
- **Flotantes:** Números Flotantes o decimales (ej. 3.1416), representados por la palabra clave *flotante*
- **Caracteres:** Letras (ej. 'a', 'b'). Estas estan representadas por la palabra clave *char*
- **Booleanos:** Valores Booleanos (ej. Verdadero o falso). Esta representados por la palabra clave *bool*

```
entero una_Variable_1;
```

**Figura 2.3**

Ejemplo de una declaración de variable

Las declaraciones solo se pueden hacer en los lugares apropiados en el programa para las declaraciones. Estos son el espacio de variables globales que se encuentra directamente después de la primera corcheta del programa, y en las corchetes de declaración de variables locales los cuales se encuentran después de los nombres de funciones.

Los programas cuentan con dos tipos de alcances de variables. Las variables globales, las cuales pueden ser accesados por todas las funciones del programa y las variables locales que solo pueden ser accesados por las funciones en las que fueron declaradas.

**2.2.3. Expresiones, Imprimir, Asignación y Comentarios**

Solo declarar las variables no es suficiente para hacer programas. Por ello IntroProg cuenta con una serie de estatutos básicos que son utilizados para realizar operaciones con las variables declaradas. Primero para poder asignarles información a las variables se tiene que hacer una asignación. Esto se logra utilizando el símbolo de `=`. Para asignar valores a variables primero se tiene que escribir el identificador de la variable seguido por el símbolo `=` seguido del valor que se le quiere asignar.

```
una_Variable_1 = 10;  
// Tambien se le pueden asignar  
// el valor de otra variable  
una_Variable_1 = x;
```

**Figura 2.4**

Ejemplo de asignación

Hay que tomar en consideración que solo se le pueden asignar valores a las variables que sean del tipo de la variable. En caso de que se le intente asignar un valor a la variable que no sea de su tipo, IntroProg va a intentar convertir el valor que va a ser asignado al tipo de la variable. Si no puede hacerlo el compilador va a levantar un error.

Ya con valores definidos se pueden empezar a hacer expresiones. Las expresiones en IntroProg son como las expresiones de matemáticas. Estas hacen una operación sobre las variables y constantes y generan un resultado. Estos resultados pueden ser guardados en otras variables.

```
una_Variable_1 = 10 * una_Variable_1 + 1;
```

**Figura 2.5**

Ejemplo de asignación con expresiones

IntroProg sigue la siguiente jerquía de operaciones:

- Parentesis : Simpre va a hacer primero lo que hay entre parentesis
- Negativos/Positivos: Primero se hace negativo o positivo el valor de la variable o constante (ej. -1, +x)
- Multiplicaciones/Divisiones : Operaciones de multiplicación y División
- Sumas/Restas : Operaciones de sumas y restas
- Relacionales : Operaciones relacionales, las cuales son mayor (>), menor (<), igual (==), mayor o igual (>=), menor o igual (<=) y diferente (!=)
- Lógicas : Operaciones Lógicas como *o* (||) e *y* (∧)

Es importante mencionar que todos los estatutos o expresiones que se hagan en IntroProg deben de terminar en un punto y coma. De otra manera el compilador va a marcar error.

Pero no solo es necesario hacer operaciones, también es importante saber el resultado de las operaciones. Para poder ver estos resultados es importante poder imprimir a pantalla los valores de las variables. Por eso IntroProg provee el estatuto de *imprimir* el cual permite imprimir a consola el valor de alguna variable o expresión. Para usar este estatuto se tiene que usar la palabra clave *imprimir* seguida por todos los elementos a imprimir separados por comas.

```
una_Variable_1 = 10 * una_Variable_1 + 1;  
imprimir(una_Variable_1,10);
```

**Figura 2.6**

Ejemplo de impresión

Este estatuto también puede manejar el uso de constantes de cadenas. Las constantes de cadenas son oraciones o palabras que estan entre comillas. Esto puede ser util para organizar bien la información cuando se despliega en pantalla.

```
una_Variable_1 = 10 * una_Variable_1 + 1;  
imprimir("Esto es una cadena : ",una_Variable,10);
```

**Figura 2.7**

Ejemplo del uso de cadenas en impresión

Finalmente IntroProg cuenta con un estatuto especial que puede ayudar a los programadores a organizar su código. Este estatuto es el comentario. El comentario se escribe como dos diagonales seguido de lo que el programador quiera escribir. Todo lo que siga después de las diagonales va a ser ignorado por el compilador.

```
una_Variable_1 = 10 * una_Variable_1 + 1;  
imprimir(una_Variable,10); // el comentario puede ser lo que sea  
// El compilador ignora esta linea  
// Y no casua errores
```

**Figura 2.8**

Ejemplo de Comentarios

**2.2.4. Condiciones**

Las estructuras de control más comúnmente utilizadas. En la mayoría de los lenguajes son reconocido como la instrucción o palabra clave *if*. En IntroProg esta estructura está presente bajo el nombre de *si*, y como su contraparte en C/C++ funciona de manera similar.

**2.2.4.1. Estructura si**

La estructura *si* está compuesta por la palabra clave seguida de una expresión entre paréntesis. Después de este paréntesis se escribe un bloque de código entre corchetes. El comportamiento de *si* es el siguiente. *Si* recibe una expresión booleana entre paréntesis y si la expresión resulta en verdadero ejecuta el código que se encuentra en los corchetes próximos a la estructura. Si la expresión es falsa se salta a la primera línea de código que sigue después de las corchetes.



```
si (expresion){  
    //Codigo a ejecutar en verdadero  
}  
//Continua codigo
```

**Figura 2.9**Ejemplo de *si*

#### 2.2.4.2. Estructura si-sino

Similar a sus contrapartes IntroProg también cuenta con la estructura de *if-else*. En IntroProg esta estructura se escribe con las palabras clave *si* y *sino*. Su funcionamiento es el siguiente. Como un *si* regular, primero se evalúa la expresión que se encuentra entre paréntesis enfrente de la palabra clave, si es verdadero ejecuta el código entre las corchetas seguidas de la expresión. En caso de que la condición sea falsa se salta a ejecutar las corchetas después de la palabra clave *sino*.

```
si (expresion){  
    //Codigo a ejecutar en verdadero  
}sino{  
    //Codigo a ejecutar en falso  
}  
//Continua codigo
```

**Figura 2.10**Ejemplo de *si-sino*

#### 2.2.5. Bucles

Los bucles son una de las estructuras de flujo esenciales de los lenguajes de programación. Y como todos los lenguajes, IntroProg también cuenta con formas de representar estas estructuras. En IntroProg hay dos formas de hacer bucles. La primera estructura es *mientras*, el cual es muy similar a su contraparte de C/C++ *while*. La segunda estructura es *por* el cual es similar a su contraparte de C/C++ *for*.

##### 2.2.5.1. Estructura Mientras

La estructura de *mientras* consta de la palabra clave seguida de una expresión entre paréntesis y un bloque de código entre corchetas. Lo que va a hacer es repetir el bloque de código mientras que la expresión que se encuentra

entre paréntesis enfrente del *mientras*. Si la expresión resulta en falso se salta a la siguiente línea de código después de la estructura.

```
mientras(expresion){  
    //Codigo a repetir mientras sea verdadero  
}  
// Codigo continua aqui
```

**Figura 2.11**

Ejemplo de *mientras*

**2.2.5.2. Estructura Por**

La segunda estructura de flujo de IntroProg es la estructura *por*, la cual es similar a su contraparte *for* en el lenguaje C/C++. La estructura consta de la palabra *por* seguida por una asignación, una expresión a evaluar y una segunda asignación indicando el paso a seguir después de cada ciclo encerrado entre un paréntesis. Seguido de este paréntesis sigue un bloque de código entre corchetes el cual va a ser ejecutado mientras que la expresión del *por* sea verdadera.

```
por(asignacion; expresion; asignacion;){  
    //Codigo a repetir  
}  
//Codigo continua
```

**Figura 2.12**

Ejemplo de *por*

El *por* también se puede escribir sin la primera asignación, asumiendo que la variables utilizadas en el paso y la expresión estén declaradas.

```
por(; expresion; asignacion;){  
    //Codigo a repetir  
}  
//Codigo continua
```

**Figura 2.13**

Ejemplo de *por* sin la primera asignación

### 2.2.6. Funciones

Otra parte importante de los lenguajes de programación es la posibilidad de poder romper el código en funciones que cumplen un funcionamiento específico, definido por el programador. IntroProg estructura sus funciones de la siguiente manera.

#### 2.2.6.1. Declaración de función

Antes de utilizar funciones es importante primero que el usuario declare las funciones. Para ello hay que escribir la declaración de las funciones en el espacio apropiado. En la estructura de un programa de IntroProg las funciones se declaran después de las declaraciones de las variables globales, pero antes de la función principal.

```
programa pelos{
    // Primero las variables globales
    entero x;
    bool bo;
    // Aquí van las funciones
    funcion vacio holaMundo(){}{
        imprimir("Hola Mundo!!!");
    }
    // El código principal del programa
    principal funcion {
        // Variables locales a principal
        entero x,y;
    }{
        x = 10;
        y = 2;
        x = x*y;
        imprimir(x);
        $holaMundo();
    }
}
```

**Figura 2.14**

Ejemplo del lugar donde se declaran las funciones

El programador puede declarar una función usando la palabra clave *funcion* seguida por el tipo de función. Después se escribe los parámetros de la función, escritos entre paréntesis denotando el tipo y el identificador del parámetro. Es importante remarcar que una función puede no tener parámetros. Se pueden definir múltiples parámetros separando con comas cada uno de ellos. Después de definir los parámetros se escribe entre corchetes las declaraciones de variables. Si no se planea usar variables locales en la función se puede dejar

vacío el espacio entre los corchetes. Después de declarar las variables se escribe el bloque de código que debe de ejecutar la función entre corchetes.

```
funcion vacio holaMundo(){}{  
    imprimir("Hola Mundo!!!");  
}
```

**Figura 2.15**

Ejemplo de función vacía

```
funcion entero suma( entero n, entero m){}{  
    regresa n * m;  
}
```

**Figura 2.16**

Ejemplo de función con retorno

En IntroProg hay dos tipos de funciones, las funciones vacías y las funciones con retorno. Las funciones vacías sirven solo para ejecutar código que no se esperan que regresen un valor, esto puede ser imprimir el resultado de ciertas variables o hacer cálculos temporales que no se van a utilizar después. Las funciones con retorno son funciones que realizan operaciones y que regresan un resultado. El valor de retorno de estas funciones están indicadas por la palabra clave *regresar* seguida por una expresión el cual su resultado es el que va a ser utilizado. Cabe recalcar que las funciones vacío no pueden tener el estatuto de *regresar*.

#### 2.2.6.2. Llamada de función

Una vez declarada las funciones pueden ser utilizadas como el elemento de una expresión. Esto se logra utilizando el símbolo de dólar (\$) seguido por el nombre de la función, seguido por un paréntesis con expresiones separados por comas, representando los argumentos. Si la función no tiene argumentos se dejan los paréntesis vacíos.

```
principal funcion {  
    // Variables locales a principal  
    entero x,y;  
}{  
    x = 10;  
    y = 2;  
    x = $suma(x, x*y);  
    imprimir(x);  
    $holaMundo();  
}
```

**Figura 2.17**

Ejemplo del lugar donde se declaran las funciones

### 2.2.6.3. Llamadas recursivas en funciones

IntroProg también soporta el uso de funciones recursivas. Esto se puede lograr haciendo la llamada de la misma función dentro de su declaración. Esto puede ser útil para resolver ciertos problemas de programación.

```
funcion entera factorial ( entero n) {}{  
    regresa n * $factorial(n-1);  
}
```

**Figura 2.18**

Ejemplo de función recursiva

### 2.2.7. Arreglos, Matrices y Cubos

En IntroProg las variables no solo se pueden usar para guardar solo un valor. Se pueden declarar arreglos los cuales pueden tener múltiples valores dependiendo de su declaración. Para declarar un arreglo se usa una sintaxis muy similar a la declaración simple. La declaración de un arreglo se da escribiendo primero el tipo de la variable, seguido por su identificador, seguido del tamaño del arreglo encerrado entre corchetes cuadradas.

```
entero arreglo [10]; // Un arreglo de 10 valores
```

**Figura 2.19**

Ejemplo de la declaración de un arreglo

Para poder acceder a uno de los valores que tiene el arreglo se puede llamar en una expresión de la siguiente manera. Se escribe el nombre del arreglo seguido de la posición del valor entre corchetes cuadradas. Las posiciones en IntroProg se empiezan a contar desde el 0 hasta el número entero anterior al tamaño.

```
arreglo[0] = 10; // primera posicion del arreglo
arreglo[9] = 2003; // Ultima posicion del arreglo
//Tambien se pueden usar para operaciones
arreglo[2] = arreglo[0] * arreglo[9];
```

**Figura 2.20**

Ejemplo de la llamada de un arreglo

Los arreglos en IntroProg también se les pueden asignar valores de manera directa a todas las casillas del arreglo. Esto se puede lograr listando los valores entre corchetes cuadradas y separando los valores entre comas.

```
arreglo = [1,2,3,4,5,6,7,8,9,10]
```

**Figura 2.21**

Ejemplo de una asignación textual a un arreglo

**2.2.7.1. Matrices y Cubos**

IntroProg no solo maneja arreglos sino también maneja matrices y cubos. La forma de interactuar con ellos es muy similar a los arreglos. Solo se necesita hacer lo mismo que se hace con un arreglo solo que en vez de utilizar solo un set de corchetes cuadradas se usan la cantidad de corchetes cuadradas a las dimensiones a utilizar. Es importante mencionar que IntroProg solo maneja hasta cubos y no permite crear variables con más dimensiones.

```
entero mat[3][3];
entero cubo[2][2][2];
```

**Figura 2.22**

Ejemplo de la declaración de una matriz y un cubo

```
mat[0][0] = 10;  
cubo[0][1][1] = 10;
```

**Figura 2.23**

Ejemplo de la llamada de una matriz y un cubo

```
mat = [[1,2,3],[4,5,6],[7,8,9]];  
cubo = [[[1,2],[3,4]],[[5,6],[7,8]]];
```

**Figura 2.24**

Ejemplo de una asignación textual de una matriz y un cubo

**2.2.8. Funciones Especiales**

IntroProg también cuenta con una serie de funciones especiales que pueden ser utilizadas en cualquier programa. Estas funciones tienen un enfoque hacia estadística, y le permiten a los usuarios hacer programas más complejos. Estas funciones son:

- \$leer() : Regresa la lectura capturada de consola
- \$modulo(flotante a, flotante b): Regresa el resultado de  $a \% b$
- \$suma(flotante a[]): Suma todos los elementos de un arreglo y regresa sus resultados
- \$raiz(flotante a): Regresa el flotante resultante de la raíz de a
- \$exp(flotante a): Regresa la exponencial de  $e^a$
- \$elevar(flotante a, flotante b): Regresa el resultado de  $a^b$
- \$techo(flotante a): Regresa un flotante a redondeado para arriba
- \$piso(flotante a): Regresa un flotante a redondeado para abajo
- \$cos(flotante a): Regresa el coseno de a
- \$sen(flotante a): Regresa el seno de a
- \$tan(flotante a): Regresa la tangente de a
- \$cotan(flotante a): Regresa la cotangente de a
- \$sec(flotante a): Regresa la secante de a
- \$cosec(flotante a): Regresa la cosecante de a

- `$log(flotante a)`: Regresa el logaritmo natural de a
- `$minimo(flotante a[])`: Regresa el valor más chico en el vector a
- `$maximo(flotante a[])`: Regresa el valor máximo de a
- `$redondear(flotante a)`: Regresa un flotante a redondeado
- `$productoPunto(flotante a[], flotante b[])`: Regresa el producto punto entre los vectores de entrada a y b
- `$media(flotante a[])`: Regresa la media de a
- `$mediana(flotante a[])`: regresa la mediana de a
- `$moda(flotante a[])`: Regresa el elemento con la moda más alta de a
- `$varianza(a)`: Regresa la varianza de a
- `$percentil(flotante a[], flotante q)`: Regresa el valor en el que se encuentran q
- `$aleatorio(flotante min, flotante max)`: Regresa un número flotante aleatorio entre los rangos de argumentos mínimos y máximos
- `$wilcoxon(flotante x[])`: Realiza la prueba de Wilcoxon en la serie de datos en x
- `$wilcoxonComp(flotante x[], flotante y[])`: Realiza la prueba de Wilcoxon se realiza la prueba sobre los datos x y y.
- `$regresionSimple(flotante x[], flotante y[], flotante xi)`: Dado un set de x y y se usará regresión lineal simple para encontrar  $f(xi)$  y se regresara ese valor
- `$normal(flotante media, flotante desv)`: Regresa un número escalar que pertenezca a la distribución normal dado los parámetros
- `$poisson(flotante lambda)`: Regresa un número aleatorio de la distribución Poisson con la lambda dada
- `$dexponencial(flotante beta)`: Regresa un número aleatorio de la distribución Exponencial correspondiente a la beta (o  $1/\text{Lambda}$ ) dada.
- `$dgeometrica(flotante exito)`: Te regresa un valor con la distribución geométrica con la probabilidad de exito dada.
- `$histograma(flotante x[], flotante rango)`: Genera un histograma a partir de los datos en el vector de x con un rango entre los datos de rango.
- `$diagramadecaja(flotante x[])`: Genera un diagrama de caja y bigotes de los datos en en x



- `$grafDispersion(flotante x[], flotante y[])`: Genera un gráfico de dispersión con los valores de 'x' y 'y'

Para llamar a una función especial se usa la misma sintaxis que una función normal.

```
x = $aleatorio(1,100);
```

### Figura 2.25

Ejemplo del uso de funcinoes especiales

---

## 2.3. Listado de errores

El compilador y la máquina virtual manejan distintos tipos de errores que pueden suceder cuando se hace un programa. Como el lenguaje está diseñado para ser un punto de entrada a la programación se intentó cubrir la mayor cantidad de errores de programación comunes y se agregaron mensajes de error apropiados a la situación del error. Estos mensajes de error tratan de explicar de una manera clara cuál es la situación que causó el error al usuario.

Los errores que se pueden detectar durante compilación son los siguientes:

- Doble declaración de variables
- El uso de variables no declaradas.
- Errores de sintaxis. Esto incluye identificadores y palabras reservadas mal escritas, estatutos mal escritos entre otros errores de sintaxis.
- Número de parámetros incorrectos
- Asignación incorrecta de parámetros, si el valor pasado como parámetro no se puede convertir a al tipo del parámetro el compilador levanta un error.
- Operaciones con funciones vacías, el compilador no permite el uso de funciones vacías en expresiones
- Retornos en funciones vacías, El compilador levanta errores cuando detecta un retorno cuando hay retornos en funciones vacías
- No permitir el nombre de funciones y palabras reservadas como identificadores.

Los errores que puede detectar la máquina virtual son la siguiente:

- Operaciones con variables sin valor. Si se intenta realizar operaciones con variables las cuales no cuentan con un valor se levanta un error y se le indica al usuario cuál fue el posible error.
- Detectar si los subíndices del arreglo se salen del espacio definido por el arreglo. Similarmente detecta cuando el arreglo se sale de los límites del arreglo e indica al usuario mediante un mensaje de error lo que sucedió.
- Lectura incorrecta. Cuando se utiliza la lectura y se mete un valor que no es válido para lectura.

## Parte II

# Descripción del compilador y la máquina virtual



# 3

## Descripción del compilador

### CONTENTS

3.1	Equipo de cómputo, lenguaje y utilerías especiales usadas en el desarrollo del proyecto .....	29
3.2	Descripción del Análisis de Léxico .....	30
3.3	Descripción del Análisis de Sintaxis .....	32
3.4	Descripción de Generación de Código Intermedio y Análisis Semántico .....	36
	3.4.0.1 Direcciones de Memoria Virtuales .....	36
	3.4.0.2 Códigos Intermedios (Cuadрупlos) .....	37
	3.4.0.3 Cubo Semántico .....	39
	3.4.0.4 Diagramas de Sintaxis y Acciones Semánticas .....	41
3.5	Descripción de Administración de Memoria usado en la compilación .....	62
	3.5.0.1 Directorio de Funciones y el Directorio de Funciones Especiales .....	62
	3.5.0.2 Vector de Cuadрупlos y su contador .....	63
	3.5.0.3 Tabla de Constantes y su espejo .....	64
3.5.1	Tabla de variables .....	67
	3.5.1.1 Cubo Semántico .....	67
	3.5.1.2 Contadores Globales de Variables Globales y temporales, Máximos de Variables y las Direcciones de Memoria .....	71
	3.5.1.3 Vector de relación línea código fuente a cuádruplo .....	71
	3.5.1.4 Variables auxiliares para el manejo de arreglos .....	72
	3.5.1.5 Pila de operadores, Pila de operandos y Pila de tipos .....	72

En esta parte se describe con más detalle el compilador

---

### 3.1. Equipo de cómputo, lenguaje y utilerías especiales usadas en el desarrollo del proyecto

Una PC con Windows 10, se utilizó el lenguaje de programación Python 3.10 con apoyo de las librerías de *PLY*, *Numpy*, *re*, *json*.

Para el compilador se utilizó la librería de *Ply* para el manejo del análisis léxico, sintáctico, y la semántica estática del lenguaje. En el caso del análisis semántico también se utilizó la librería de *re* para manejar expresiones regulares en la lógica de semántica y código intermedio. Finalmente para poder guardar con mayor facilidad el código intermedio, la tabla de funciones y la tabla de constantes se uso la librería de JSON para exportarlos a un archivo.

---

### 3.2. Descripción del Análisis de Léxico

Para este lenguaje se opto por utilizar palabras clave en español. Esto se decidió con el objetivo de ser más intuitivo para las personas que hablan español y que tengan poco conocimiento del lenguaje inglés.

En IntroProg se usaron las siguientes Palabras reservadas:

- programa
- funcion
- si
- sino
- mientras
- por
- entero
- flotante
- char
- cadena
- bool
- vacío
- nulo
- verdadero

- falso
- principal
- imprimir
- regresar

A su vez los tokens importantes del lenguaje son los siguientes:

- `dígito` :  $[0 - 9]$
- `letra` :  $[a - zA - Z]$
- `CTE_INT` : `dígito+`
- `CTE_FLOAT` : `dígito+\.dígito+`
- `CTE_STRING` : `\” [ \^” ] \”`
- `COMMENT` : `\\\/.*`
- `espacio blanco` :  $[\backslash n \backslash t]^+$
- `ID` : `letra(letra | dígito | _ )*`
- `SEMICOLON` : `;`
- `COMMA` : `,`
- `EQ` : `=`
- `OPENPAR` : `\(`
- `CLOSEPAR` : `\)`
- `GT` : `<`
- `LT` : `>`
- `PLUS` : `+`
- `MINUS` : `-`
- `MUL` : `*`
- `DIV` : `/`
- `OPENCUR` : `\{`
- `CLOSECUR` : `\}`
- `OPENSQU` : `\[`
- `CLOSESQU` : `\]`

- AND : &&
- OR : ||
- EXLAM : !
- DLR : \$

---

### 3.3. Descripción del Análisis de Sintaxis

Con los tokens definidos en la parte léxica se desarrolló la siguiente gramática para describir el lenguaje :

1. PROGRAM  $\rightarrow$  PROGRAMA ID OPENCUR DECLARACIONES DECTODASFUNC PRINCIPAL FUNCION OPENCUR DECLARACIONES CLOSECUR BLOQUE CLOSECUR
2. DECLARACIONES  $\rightarrow$  DECLARACION DECLARACIONES  
|  $\epsilon$
3. DECLARACION  $\rightarrow$  DECVAR  
| DECARR
4. DECVAR  $\rightarrow$  TIPO ID DVNID SEMICOLON
5. DVNID  $\rightarrow$  COMMA ID DVNID  
|  $\epsilon$
6. DECARR  $\rightarrow$  TIPO ID OPENSQU CTE\_INT CLOSESQU DSEG SEMICOLON
7. DSEG  $\rightarrow$  OPENSQU CTE\_INT CLOSESQU DTER  
|  $\epsilon$
8. DTER  $\rightarrow$  OPENSQU CTE\_INT CLOSESQU  
|  $\epsilon$
9. TIPO  $\rightarrow$  ENTERO  
| FLOTANTE  
| CHAR  
| BOOL
10. DECTODASFUNC  
| DECFUNC DECTODASFUNC
11. DECFUNC  $\rightarrow$  FUNCION TIPOFUN ID OPENPAR FUNPARAM CLOSEPAR OPENCUR DECLARACIONES CLOSECUR BLOQUE
12. TIPOFUN  $\rightarrow$  TIPO  
| VACIO
13. FUNPARAM  $\rightarrow$  PARAM  
|  $\epsilon$
14. PARAM  $\rightarrow$  TIPO ID PARAMD PARAMS
15. PARAMS  $\rightarrow$  COMMA PARAM  
|  $\epsilon$



16.  $\text{PARAMD} \rightarrow \text{OPENSQU CTE\_INT CLOSESQU PDSEG}$   
|  $\epsilon$
17.  $\text{PDSEG} \rightarrow \text{OPENSQU CTE\_INT CLOSESQU PDTER}$   
|  $\epsilon$
18.  $\text{PDTER} \rightarrow \text{OPENSQU CTE\_INT CLOSESQU}$   
|  $\epsilon$
19.  $\text{BLOQUE} \rightarrow \text{OPENCUR ESTATUTOS CLOSECUR}$
20.  $\text{ESTATUTOS} \rightarrow \text{ESTATUTO ESTATUTOS}$   
|  $\epsilon$
21.  $\text{ESTATUTO} \rightarrow \text{IMPRESION}$   
|  $\text{ASIGNACION}$   
|  $\text{EXPRESION SEMICOLON}$   
|  $\text{CONDICION}$   
|  $\text{BUCLE}$   
|  $\text{RETURNF}$
22.  $\text{IMPRESION} \rightarrow \text{IMPRIMIR OPENPAR PRINTABLE PRINTARGS}$   
 $\text{CLOSEPAR SEMICOLON}$
23.  $\text{PRINTARGS} \rightarrow \text{COMMA PRINTABLE PRINTARGS}$   
|  $\epsilon$
24.  $\text{PRINTABLE} \rightarrow \text{EXPRESION}$   
|  $\text{CTE\_STRING}$   
|  $\epsilon$
25.  $\text{ASIGNACION} \rightarrow \text{ID ADIMS EQ EXPRESION SEMICOLON}$   
|  $\text{ID EQ ARR\_TEX SEMICOLON}$
26.  $\text{ADIMS} \rightarrow \text{OPENSQU EXPRESION CLOSESQU ASEGD}$   
|  $\epsilon$
27.  $\text{ASEGD} \rightarrow \text{OPENSQU EXPRESION CLOSESQU ATERD}$   
|  $\epsilon$
28.  $\text{ATERD} \rightarrow \text{OPENSQU EXPRESION CLOSESQU}$   
|  $\epsilon$
29.  $\text{CONDICION} \rightarrow \text{SI OPENPAR EXPRESION CLOSEPAR BLOQUE}$   
 $\text{IFELSE}$
30.  $\text{IFELSE} \rightarrow \text{SINO BLOQUE}$   
|  $\epsilon$
31.  $\text{BUCLE} \rightarrow \text{WHILE}$   
|  $\text{FOR}$
32.  $\text{WHILE} \rightarrow \text{MIENTRAS OPENPAR EXPRESION CLOSEPAR BLOQUE}$
33.  $\text{FOR} \rightarrow \text{POR OPENPAR FORINIT EXPRESION SEMICOLON ASIGNACION}$   
 $\text{CLOSEPAR BLOQUE}$
34.  $\text{FORINIT} \rightarrow \text{ASIGNACION}$   
|  $\epsilon$
35.  $\text{RETURNF} \rightarrow \text{REGRESAR EXPRESION SEMICOLON}$
36.  $\text{EXPRESION} \rightarrow \text{EXPRESIONR}$   
|  $\text{EXPRLOG}$

- 37.  $\text{EXPRLOG} \rightarrow \text{EXPRESION AND EXPRESION}$   
 $\quad | \text{EXPRESION OR EXPRESION}$   
 $\quad | \epsilon$
- 38.  $\text{EXPRESIONR} \rightarrow \text{EXP}$   
 $\quad | \text{EXPR}$
- 39.  $\text{EXPR} : \text{EXP LT EXP}$   
 $\quad | \text{EXP GT EXP}$   
 $\quad | \text{EXP EXLAM EQ EXP}$   
 $\quad | \text{EXP EQ EQ EXP}$   
 $\quad | \text{EXP LT EQ EXP}$   
 $\quad | \text{EXP GT EQ EXP}$   
 $\quad | \epsilon$
- 40.  $\text{EXP} \rightarrow \text{TERMINO}$   
 $\quad | \text{TERMINOSS}$
- 41.  $\text{TERMINOSS} \rightarrow \text{EXP PLUS EXP}$   
 $\quad | \text{EXP MINUS EXP}$   
 $\quad | \epsilon$
- 42.  $\text{TERMINO} \rightarrow \text{FACTOR}$   
 $\quad | \text{FACTORESS}$
- 43.  $\text{FACTORES} \rightarrow \text{TERMINO MUL TERMINO}$   
 $\quad | \text{TERMINO DIV TERMINO}$   
 $\quad | \epsilon$
- 44.  $\text{FACTOR} \rightarrow \text{SIGNOVAR VARCTE}$   
 $\quad | \text{OPENPAR EXPRESION CLOSEPAR}$
- 45.  $\text{SIGNOVAR} \rightarrow \text{PLUS}$   
 $\quad | \text{MINUS}$   
 $\quad | \epsilon$
- 46.  $\text{VARCTE} : \text{ID}$   
 $\quad | \text{CTE\_INT}$   
 $\quad | \text{CTE\_FLOAT}$   
 $\quad | \text{CTE\_STRING}$   
 $\quad | \text{CTE\_BOOL}$   
 $\quad | \text{CTE\_CHAR}$   
 $\quad | \text{LLAMADAFUNC}$   
 $\quad | \text{LLAMADAARR}$   
 $\quad | \text{NULO}$
- 47.  $\text{LLAMADAFUNC} \rightarrow \text{DLR ID OPENPAR CALLPARAMS CLOSEPAR}$
- 48.  $\text{CALLPARAMS} \rightarrow \text{CPARAM}$   
 $\quad | \epsilon$
- 49.  $\text{CPARAM} \rightarrow \text{EXPRESION CPARAMS}$
- 50.  $\text{CPARAMS} \rightarrow \text{COMMA EXPRESION CPARAMS}$   
 $\quad | \epsilon$
- 51.  $\text{LLAMADAARR} \rightarrow \text{ID OPENSQU EXPRESION CLOSESQU LLSEGD}$
- 52.  $\text{LLSEGD} \rightarrow \text{OPENSQU EXPRESION CLOSESQU LLTERD}$   
 $\quad | \epsilon$
- 53.  $\text{LLTERD} \rightarrow \text{OPENSQU EXPRESION CLOSESQU}$   
 $\quad | \epsilon$

- 54.  $\text{ARR\_TEX} \rightarrow \text{OPENSQU ATPRIC CLOSESQU}$
- 55.  $\text{ATPRIC} \rightarrow \text{ATPRE ATPRISIG}$   
|  $\epsilon$
- 56.  $\text{ATPRE} \rightarrow \text{EXPRESION}$   
|  $\text{ATSEGD}$
- 57.  $\text{ATPRISIG} \rightarrow \text{COMMA ATPRE ATPRISIG}$   
|  $\epsilon$
- 58.  $\text{ATSEGD} \rightarrow \text{OPENSQU ATSEGC CLOSESQU}$
- 59.  $\text{ATSEGC} \rightarrow \text{ATSEGE ATSEGSIG}$   
|  $\epsilon$
- 60.  $\text{ATSEGE} \rightarrow \text{EXPRESION}$   
|  $\text{ATTERD}$
- 61.  $\text{ATSEGSIG} \rightarrow \text{COMMA ATSEGE ATSEGSIG}$   
|  $\epsilon$
- 62.  $\text{ATTERD} \rightarrow \text{OPENSQU ATTERC CLOSESQU}$
- 63.  $\text{ATTERC} \rightarrow \text{ATTERE ATTERSIG}$   
|  $\epsilon$
- 64.  $\text{ATTERE} \rightarrow \text{EXPRESION}$
- 65.  $\text{ATTERSIG} \rightarrow \text{COMMA ATTERE ATTERSIG}$   
|  $\epsilon$

Con esta gramática se procedió a generar diagramas sintácticos.

---

### 3.4. Descripción de Generación de Código Intermedio y Análisis Semántico

En esta sección se explica el funcionamiento del Código Intermedio y el funcionamiento del Análisis semántico.

#### 3.4.0.1. Direcciones de Memoria Virtuales

El compilador en todas sus operaciones asume una abstracción del manejo de memoria en la máquina virtual. Para lograr esto el compilador asigna direcciones de memoria virtuales a las constantes y variables que encuentra en su ejecución. Estas direcciones están son las siguientes.

- Direcciones de las variables Globales
  - Enteros : 1000
  - Flotantes: 3000
  - Caracteres :5000
  - Booleanos: 6000
  - Apuntadores: 7000
- Direcciones de las Variables Locales
  - Enteros : 9000
  - Flotantes : 11 000
  - Caracteres : 13 000
  - Booleanos : 14 000
  - Apuntadores : 15 000
- Direcciones de las Variables Temporales
  - Enteros: 17 000
  - Flotantes: 19 000
  - Caracteres: 21 000
  - Booleanos: 22 000
  - Apuntadores: 23 000
- Direcciones de las Constantes
  - Enteros: 25 000
  - Flotantes: 27 000
  - Caracteres: 29 000

- Booleanos: 30 000
- Cadenas: 31 000

No solo asigna direcciones virtuales también mantiene un conteo de los recursos que se están utilizando. Si se pasa de la cantidad de variables de cada tipo y scope que se pueden usar en un espacio de memoria el compilador va a levantar un error. Estos límites son los siguientes:

- Máximo de Enteros = 2000
- Máximo de Flotantes = 2000
- Máximo de Caracteres = 1000
- Máximo de Booleanos = 1000
- Máximo de Apuntadores = 2000
- Máximo de Cadenas = 1000

Con estos máximos se puede detectar si el usuario está declarando variables demás. Cada uno de estos máximos se dan por scope, es decir un programa puede tener un máximo de 2000 enteros globales y 2000 enteros locales en cada función. Si el usuario intenta declarar más variables que esas el compilador va a levantar un error indicando que se sobre paso el número de variables permitidos.

#### 3.4.0.2. Códigos Intermedios (Cuádruplos)

Para el código intermedio del compilador se pensaron en las acciones básicas que debe de hacer la máquina virtual. Para indicar estas acciones se utilizaron el formato de cuádruplos para representar estos códigos. Teniendo claro esto, se definieron cuáles son las acciones más importantes que se necesitan para traducir el lenguaje a operaciones que la máquina virtual pueda interpretar con facilidad. Las acciones que son de interés para que la máquina virtual las ejecute son principalmente acciones de expresiones, asignación, saltos, manejo de memoria y código de funciones especial. Con esto en mente se desarrollaron los siguientes códigos:

- *Códigos de Expresiones y Asignación:* Son códigos de operación que se enfocan en realizar operaciones de expresión. Estos están formados por el operador, el cual puede ser cualquiera de los operadores de expresiones (+, -, \*, /, etc), seguidos por el operando izquierdo, seguido por el operando derecho y en la cuarta posición se la dirección virtual del resultado. Solo existe un caso especial con expresiones, el cual es cuando el operador es un + o un -, el cual puede recibir un solo operando y este representa cuando se definen números positivos y negativos.

```

[" / ", 17000, 25007, 17001]
[" * ", 17001, 25008, 17002]
[" + ", 25006, 17002, 17003]
[" - ", 25001, 25009, 17004]
[" < ", 17007, 25008, 22001]
[" > ", 17007, 25008, 22001]
[" >= ", 17007, 25008, 22001]
[" <= ", 17007, 25008, 22001]
[" == ", 17007, 25008, 22001]
[" != ", 17007, 25008, 22001]
[" || ", 22000, 22001, 22002]
[" & & ", 22002, 30000, 22003]
[" = ", 22003, " ", 14000]

```

**Figura 3.1**

Ejemplos de códigos Expresiones y asignación

- *Código de Saltos:* Son códigos que expresan un salto a otra instrucción. Existen dos tipos de este código los cuales son GOTO y GOTOIF. El primero está conformado por la palabra clave GOTO en la primera posición y en la última posición el número del cuádruplo a donde saltar. El segundo código tiene un comportamiento diferente al primero. Este código se usa para indicar que se debe de hacer el salto si el valor en una dirección es falsa. La Composición de este cuádruplo estará dada por la palabra GOTOIF en la primera posición, la dirección a evaluar en la segunda y en la última posición el número del cuádruplo a donde hacer el salto si la expresión es falsa.

```

[ 'GOTOIF' 22003 ' ' 112 ]
[ 'GOTO' ' ' ' ' 91 ]

```

**Figura 3.2**

Ejemplos de códigos de saltos

- *Código de Funciones:* Son cuádruplos que indican las acciones necesarias para ejecutar funciones. Para las funciones se crean cuatro códigos para indicar su comportamiento. Estos son ERA, PARAMETER, GOTOSUB, ENDFUNC y SPFUNC. ERA es un código que indica la creación de un nuevo espacio de memoria. Este se genera escribiendo el código ERA en la primera posición y en la segunda el nombre de la función. El segundo código, PARAMETER, es un código utilizado para indicar que se debe de copiar la información de una variable a el nuevo espacio de memoria generado. Este código se escribe en un cuádruplo como la palabra clave PARAMETER en la primera posición, seguida por la dirección a copiar en la segunda posición.

La tercera posición se puede dejar vacía si es una variable simple, pero si es un arreglo se escribe el tamaño en ella. Y en la última posición se escribe el tipo del parámetro y el número del parámetro separados por el símbolo #. El tercer código es GOTOSUB. Este código sirve para indicar el cambio de contexto a una función. Este código se escribe poniendo la palabra clave GOTOSUB en la primera posición, el nombre de la función en la segunda posición. En la última posición se escribe el número del cuádruplo donde empieza la función. El cuarto código ENDFUNC sirve para indicar el fin de una función y el regreso al contexto anterior de la memoria. Este cuádruplo es muy sencillo y consta solo de la palabra clave ENDFUNC en la primera posición. Finalmente se tiene el código SPFUNC. Este código se utiliza para indicar el uso de una función especial, en vez de una función del programa. Este código se genera de la siguiente manera. En la primera posición se escribe la palabra clave SPFUNC, en la segunda se escribe el nombre de la función, en la tercera posición el tipo de la función y en la última posición se escribe la dirección de la variable global de retorno si es una función que regresa un valor, sino se deja vacío.

```
["PARAMETER", 25001, "", "1#1"]
["ERA", "pop", "", ""]
["GOTOSUB", "pop", "", 1]
["ENDFUNC", "", "", ""]
["SPFUNC", "dexponencial", 1, 3026]
["SPFUNC", "diagramaDeCaja", -1, ""]
```

**Figura 3.3**

Ejemplos de códigos de funciones

### 3.4.0.3. Cubo Semántico

Para el uso del código intermedio para expresiones se definio como interactuan los tipos de las variables cuando se realizan cada tipo de operación. Esto esta dado por la siguiente tabla.

**Cuadro 3.1**

Tabla representando el cubo semántico

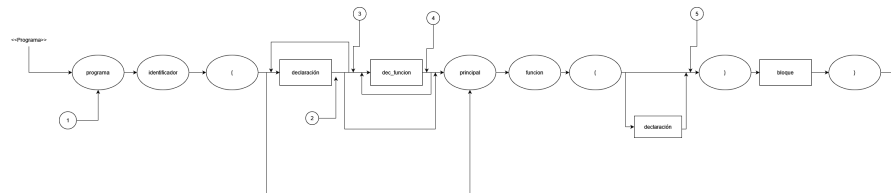
OpIzq	OpDer	=		&&	<	>	<=	>=	!=	==	+	-	*	\
ent	ent	ent	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
ent	flot	ent	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
ent	char	ent	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
ent	bool	ent	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
ent	cadena	err	err	err	err	err	err	err	err	err	err	err	err	err
flot	ent	flot	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
flot	flot	flot	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
flot	char	flot	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
flot	bool	flot	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
flot	cadena	err	err	err	err	err	err	err	err	err	err	err	err	err
char	ent	char	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
char	flot	char	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
char	char	char	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
char	bool	char	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
char	cadena	err	err	err	err	err	err	err	err	err	err	err	err	err
bool	ent	bool	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
bool	flot	bool	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
bool	char	bool	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
bool	bool	bool	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
bool	cadena	err	err	err	err	err	err	err	err	err	err	err	err	err
cadena	ent	err	err	err	err	err	err	err	err	err	err	err	err	err
cadena	flot	err	err	err	err	err	err	err	err	err	err	err	err	err
cadena	char	err	err	err	err	err	err	err	err	err	err	err	err	err
cadena	bool	err	err	err	err	err	err	err	err	err	err	err	err	err
cadena	err	err	err	err	err	err	err	err	err	err	err	err	err	err

- ent : Entero
- flot : Flotante
- err : Error



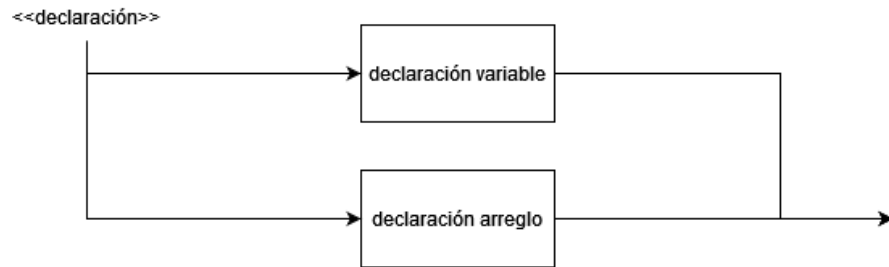
#### 3.4.0.4. Diagramas de Sintaxis y Acciones Semánticas

A Continuación, se demuestran los diagramas sintácticos con sus acciones semánticas correspondientes.

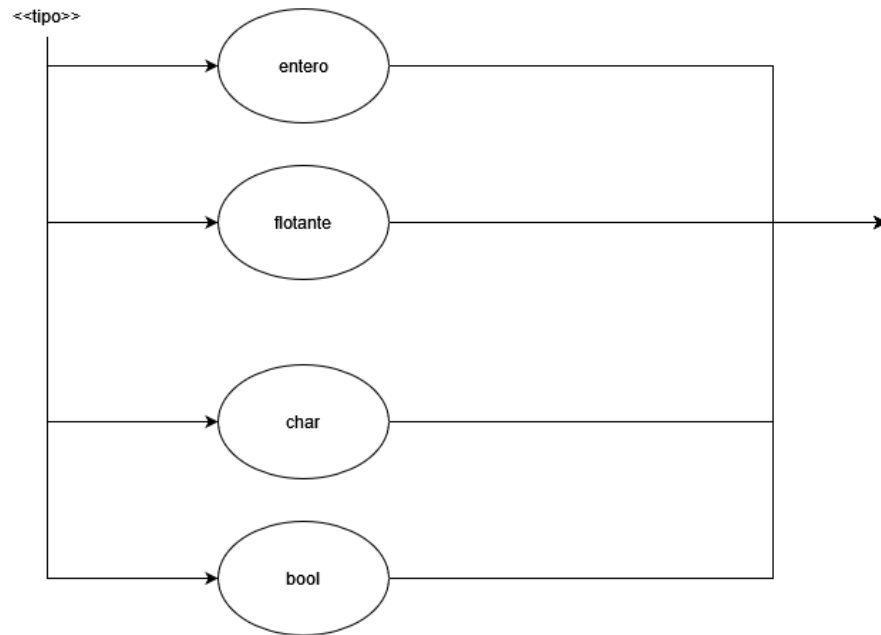


**Figura 3.4**  
Diagrama Programa

1. Se genera el cuádruplo de goto main y se mete el valor de contador de cuádruplos a la pila de saltos. Se actualiza la variable global del scope como global
2. Se agrega la entrada temporal de la variable a la tabla de variables en la entrada del espacio global en el directorio de funciones.
3. Se itera por la tabla de variables y se le asigna a cada variable una dirección virtual. Al iterar por la tabla también se actualiza el contador de recursos de las variables globales
4. Se agrega la función a el directorio de variables
5. Se genera la tabla de variables para el scope principal y se itera por la tabla asignando las direcciones virtuales adecuadas. Al terminar se hace pop a la pila de saltos y se actualiza el primer cuádruplo con el contador actual de cuádruplos.



**Figura 3.5**  
Diagrama Declaración



**Figura 3.6**  
Diagrama Tipo

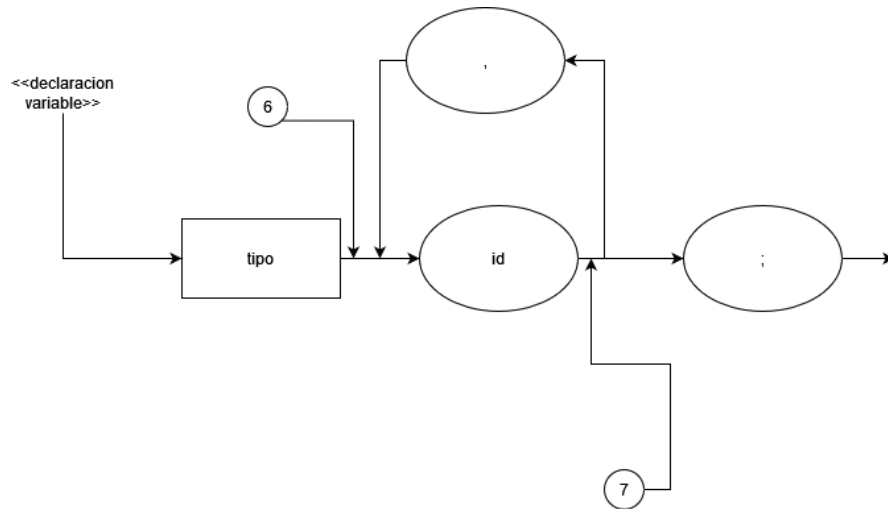
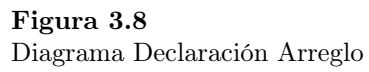
**Figura 3.7**

Diagrama Declaración Variable

6. Se guarda en una variable auxiliar global el tipo de variable
7. Se checa que el id no se repita con otros ids del scope de la declaración. Si no se repite se genera una nueva entrada de variable con el nuevo id con el tipo de variable dado por el punto 6



8. Se agrega la información de la primera dimensión a la entrada de la variable
9. Se agrega la información de la segunda dimensión a la entrada de la variable
10. Se agrega la información de la tercera dimensión a la entrada de la variable
11. Se hace el cálculo de las m y el tamaño del arreglo/matriz/cubo (depende de las dimensiones leídas) y se actualiza la entrada

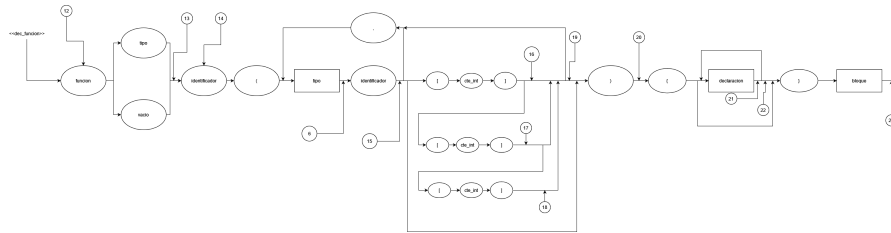
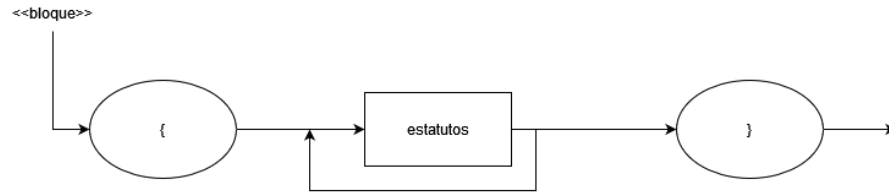
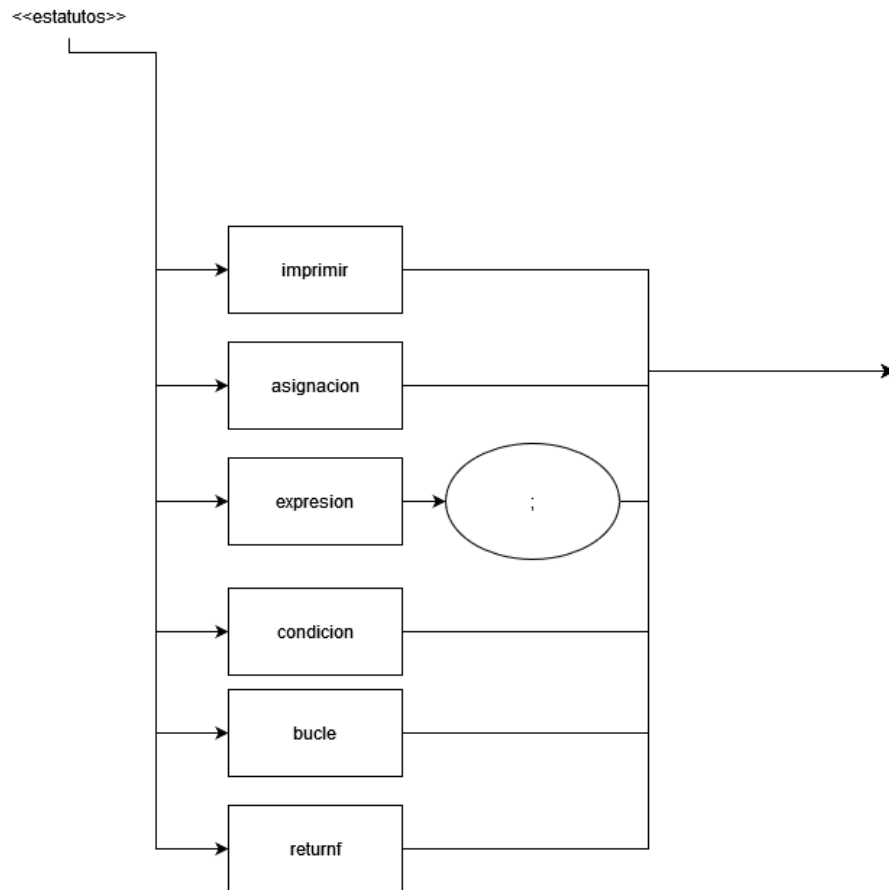
**Figura 3.9**

Diagrama Declaración Función

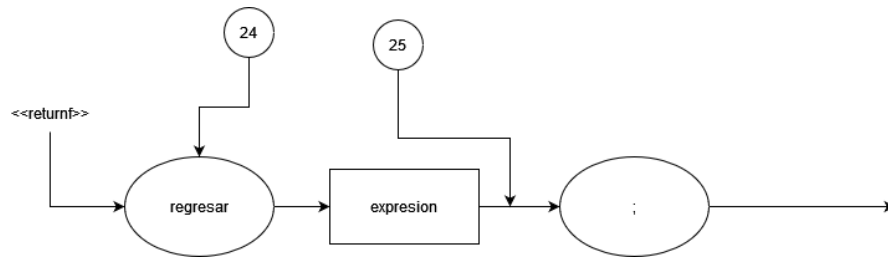
12. Se guarda en una variable auxiliar global el número del cuádruplo donde empieza la función y se genera una entrada temporal al directorio de funciones.
13. Se guarda el tipo de función a la entrada temporal
14. Se comprueba que el identificador no sea igual a otros identificadores, palabras clave o nombres de funciones especiales. Si no hay conflictos se agrega el nombre a la entrada temporal, si el tipo no es vacío, genera una variable global con el nombre y tipo de la función con la dirección virtual apropiada y se actualiza la variable global de scope y los contadores de recursos globales.
15. Se checa que el id no se repita entre otros parámetros. Si no se repite se agrega el parámetro con el id y el tipo detectado a la sección de parámetros de la entrada temporal
16. Se agrega la información de la primera dimensión a la entrada del parámetro
17. Se agrega la información de la segunda dimensión a la entrada del parámetro
18. Se agrega la información de la tercera dimensión a la entrada del parámetro
19. Se hace el cálculo de las m y el tamaño del arreglo/matriz/cubo (depende de las dimensiones leídas) y se actualiza la entrada
20. Se crean entradas a la tabla de variables locales de las funciones con la información de los parámetros
21. Se agrega la entrada temporal de la tabla de variables a la tabla de variables local.
22. Se itera por la tabla de variables y se le asigna a cada variable una dirección virtual.
23. Se genera el cuádruplo de ENDFUNC y se agregan la cantidad de recursos utilizados por la función a la entrada de la función.



**Figura 3.10**  
Diagrama Bloque

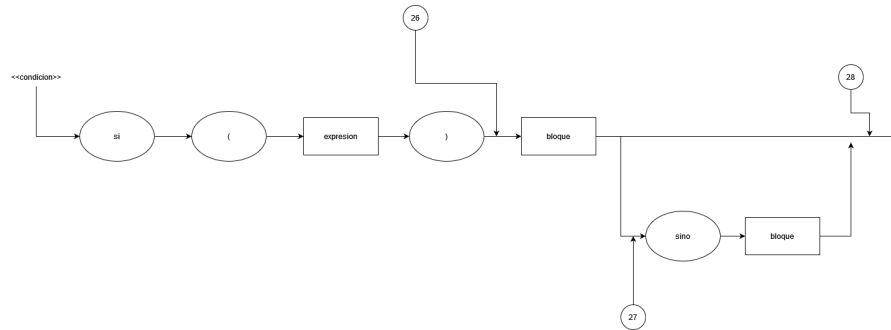


**Figura 3.11**  
Diagrama Estatutos



**Figura 3.12**  
Diagrama Return

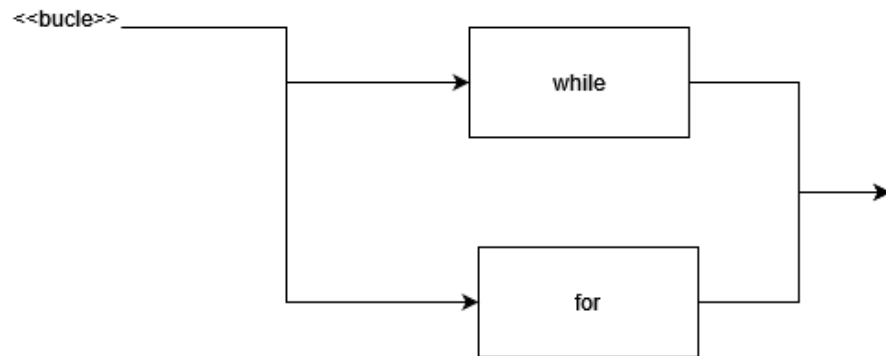
24. Checar si la función que llamo a regresar no sea una función vacía.  
Si es vacía levantar un error
25. Generar el cuádruplo de [RET, Dirección del resultado de la expresión, , Dirección de la variable global]



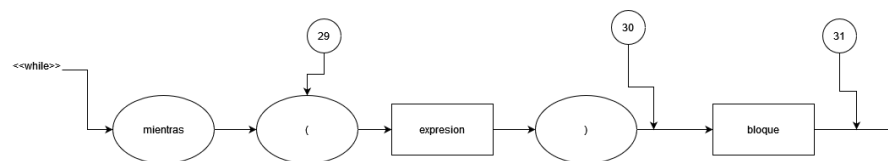
**Figura 3.13**  
Diagrama Condición

26. Se genera el cuádruplo de gotof con el resultado de la expresión y se agrega número del cuádruplo generado a la pila de saltos.
27. Se hace pop a la pila de saltos y se genera un cuádruplo de goto y se agrega su número de cuádruplo a la pila. Se actualiza el gotof con el contador actual de cuádruplos
28. Se hace pop a la pila de saltos y se actualiza el cuádruplo indicado por la pila con el contador actual.



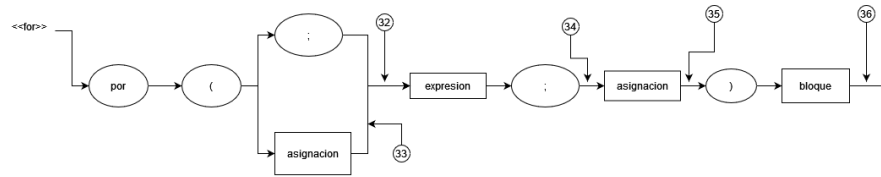


**Figura 3.14**  
Diagrama Bucle



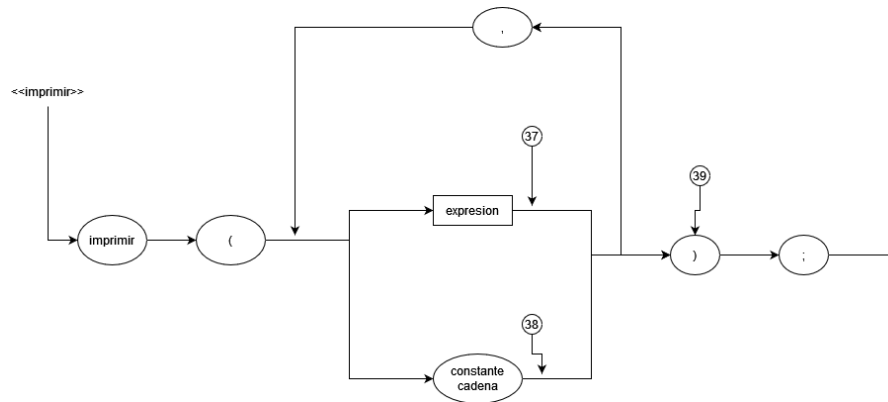
**Figura 3.15**  
Diagrama While

29. Se mete el contador actual de cuádruplos a la pila de operandos
30. Se genera un cuádruplo de gotof con el resultado de la expresión y se agrega su número de contador a la pila de saltos
31. Se le da pop dos veces a la pila de saltos y se genera un cuádruplo de goto con el segundo valor obtenido. Con el primer valor se obtiene el cuádruplo en donde esta el gotof y se actualiza con el contador actual de cuádruplos.



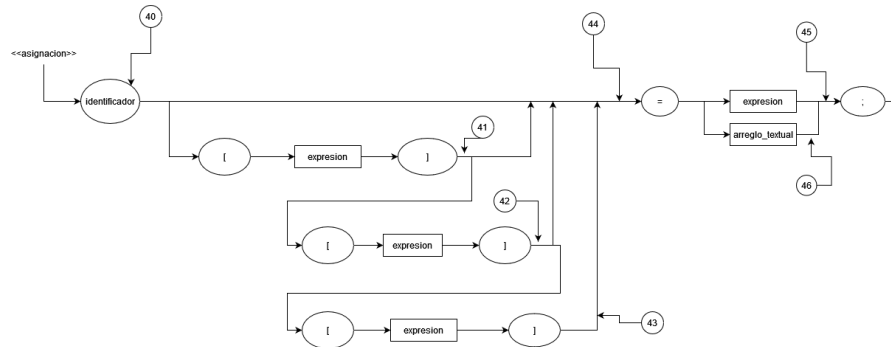
**Figura 3.16**  
Diagrama For

32. Se agrega el contador actual a la pila de saltos.
33. Se checa que la asignación sea una asignación entera
34. Se genera un gotof con la expresión y se agrega el número de cuádruplo del gotof a la pila de saltos. También se genera un cuádruplo de goto y también se agrega su número de cuádruplo a la pila de saltos.
35. Se le hace pop a los primeros cuatro elementos de la pila de saltos. Se genera un goto al cuádruplo donde inicia la condición del for, se actualiza el cuádruplo del goto que va al bloque después de la evaluación y se vuelven a insertar los elementos sobrantes.
36. Se le hace pop a dos elementos de la pila de operandos. Se genera un cuádruplo de goto a donde empieza la asignación de paso del for y se actualiza el cuádruplo de gotof de la condición del for.



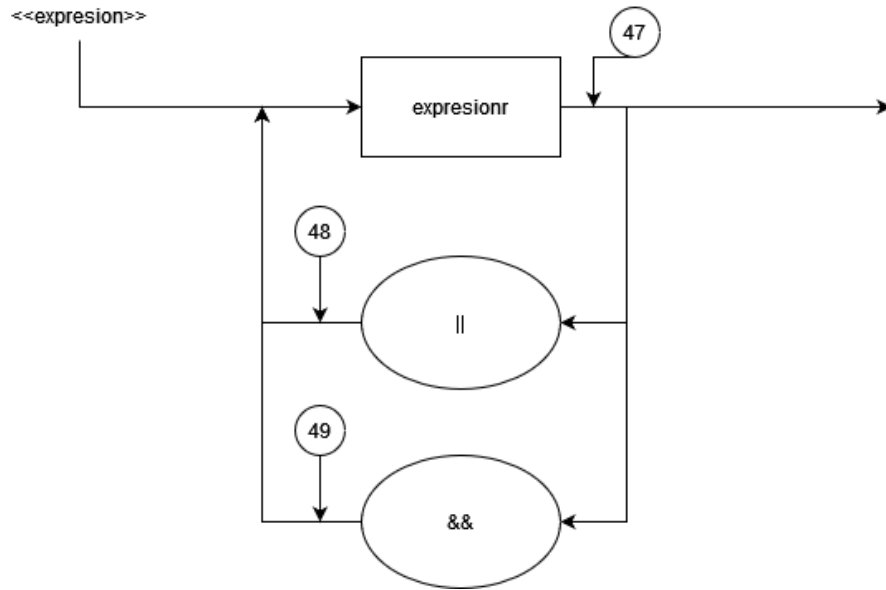
**Figura 3.17**  
Diagrama Imprimir

37. Se le hace pop a la pila de operandos y se agrega a la fila de impresiones
38. Se checa que la constante esta en la tabla de constantes, si no esta se agrega asignando la dirección virtual apropiada. Se agrega la dirección virtual a la pila de impresión.
39. Se recorre la fila de impresión y se va generando un cuádruplo de imprimir con la dirección virtual en la fila



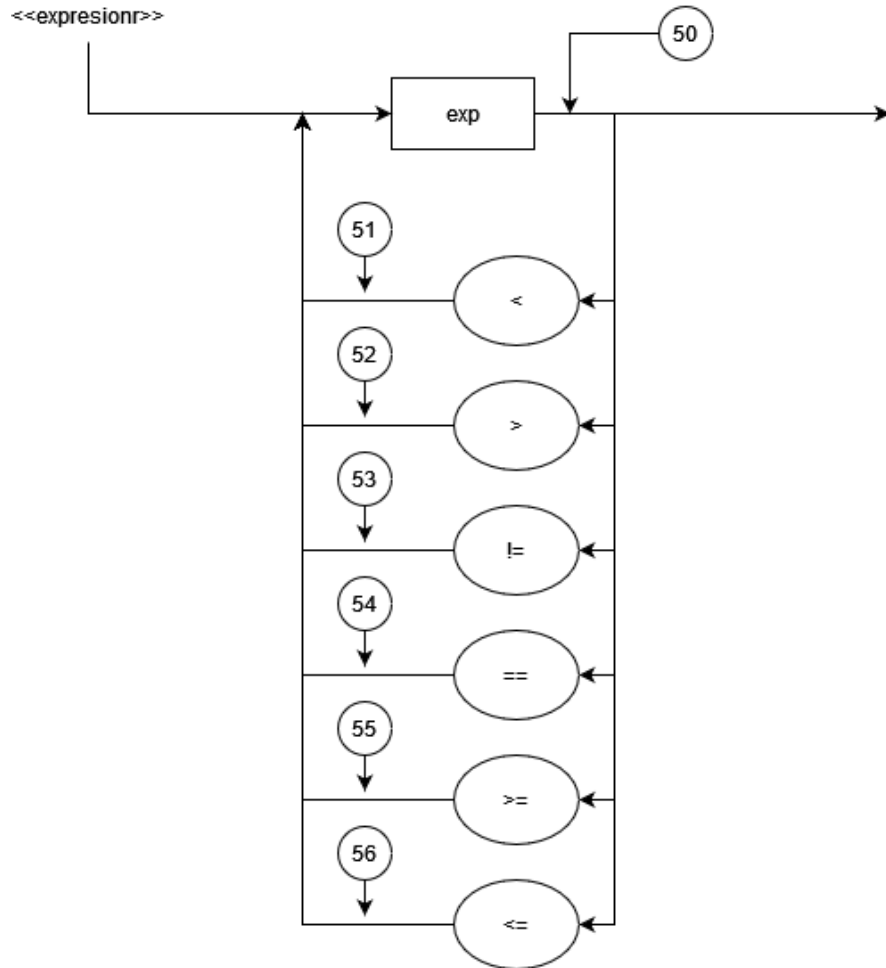
**Figura 3.18**  
Diagrama Asignación

40. Se checa que el identificador exista en el scope local o en scope global
41. Se hace pop a la pila de operandos y se revisa que el elemento recibido sea entero. Si es entero se agrega a la fila de subíndices.
42. Se hace pop a la pila de operandos y se revisa que el elemento recibido sea entero. Si es entero se agrega a la fila de subíndices.
43. Se hace pop a la pila de operandos y se revisa que el elemento recibido sea entero. Si es entero se agrega a la fila de subíndices.
44. Se revisa si el identificador es un arreglo. Si es un arreglo se checa que la cantidad de subíndices leídos coincida con las dimensiones del arreglo, de ser así itera por los subíndices generando los cuádruplos de verificación y las sumas apropiadas para calcular la dirección virtual del arreglo accesado.
45. Se checa contra el cubo semántico si se puede hacer la asignación. De ser posible se genera el cuádruplo de asignación apropiado.
46. Se checa que el identificador no sea una variable simple o una llamada de arreglo. También se checa que las dimensiones del arreglo textual son iguales a las del identificador.



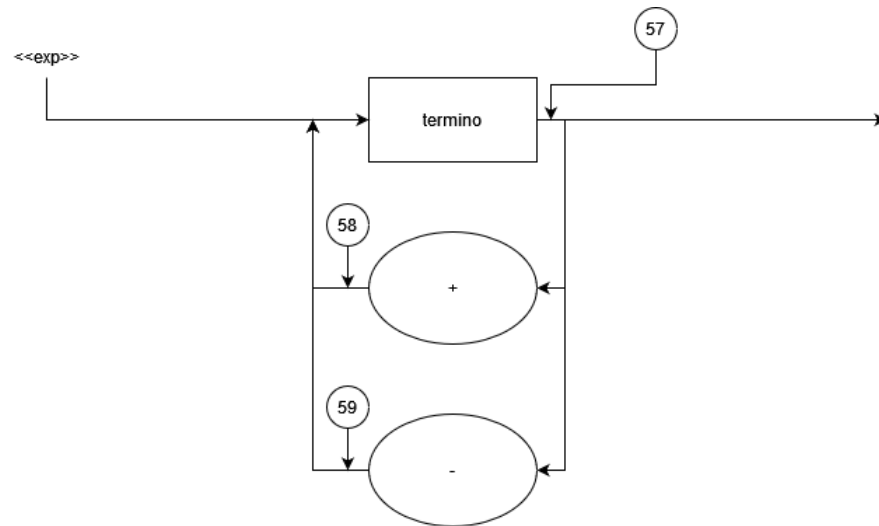
**Figura 3.19**  
Diagrama Expresión

47. Si la cima de la pila de operandos es igual a `&&` o `||`, hacer `pop` a la pila de operandos y generar el cuádruplo de expresión correspondiente a la operador detectado.
48. Meter `||` a la pila de operandos
49. Meter `&&` a la pila de operandos



**Figura 3.20**  
Diagrama ExpresiónR

50. Si la cima de la pila de operandos es igual a <, >, !=, <=, >= ó ==, hacer pop a la pila de operandos y generar el cuádruplo de expresión correspondiente a la operador detectado.
51. Meter < a la pila de operandos
52. Meter > a la pila de operandos
53. Meter != a la pila de operandos
54. Meter == a la pila de operandos
55. Meter >= a la pila de operandos
56. Meter <= a la pila de operandos



**Figura 3.21**  
Diagrama EXP

- 57. Si la cima de la pila de operandos es igual a + ó -, hacer pop a la pila de operandos y generar el cuádruplo de expresión correspondiente a la operador detectado.
- 58. Meter + a la pila de operandos
- 59. Meter - a la pila de operandos

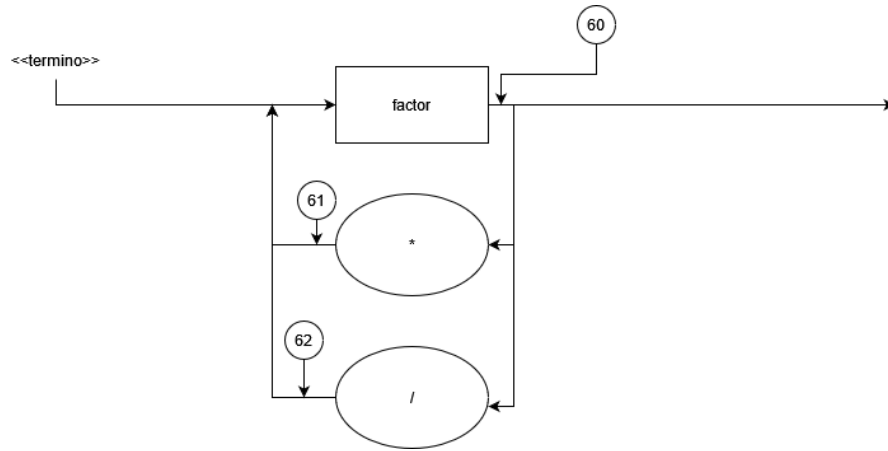
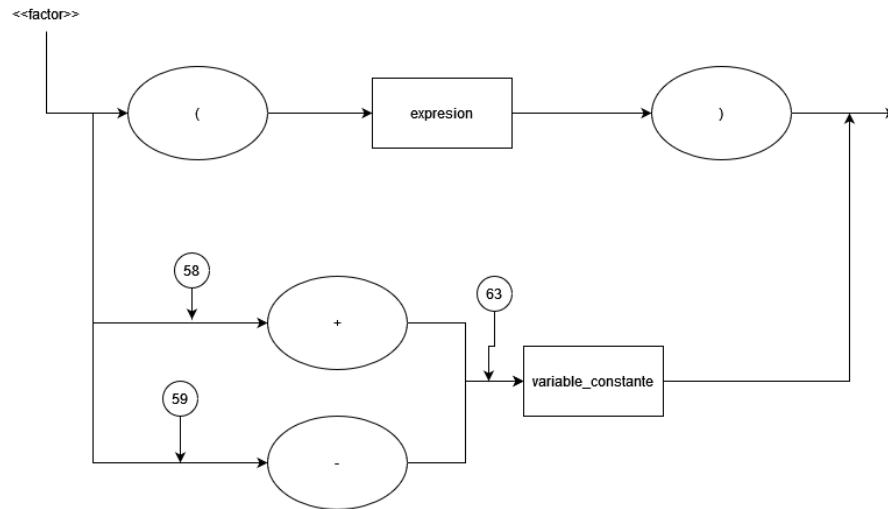
**Figura 3.22**

Diagrama Terminio

60. Si la cima de la pila de operandos es igual a \* ó / , hacer pop a la pila de operandos y generar el cuádruplo de expresión correspondiente a la operador detectado.
61. Meter \* a la pila de operandos
62. Meter / a la pila de operandos





**Figura 3.23**  
Diagrama Factor

63. Hacer pop a la pila de operador y hacer el cuádruplo de expresión unaria del operador obtenido de la pila

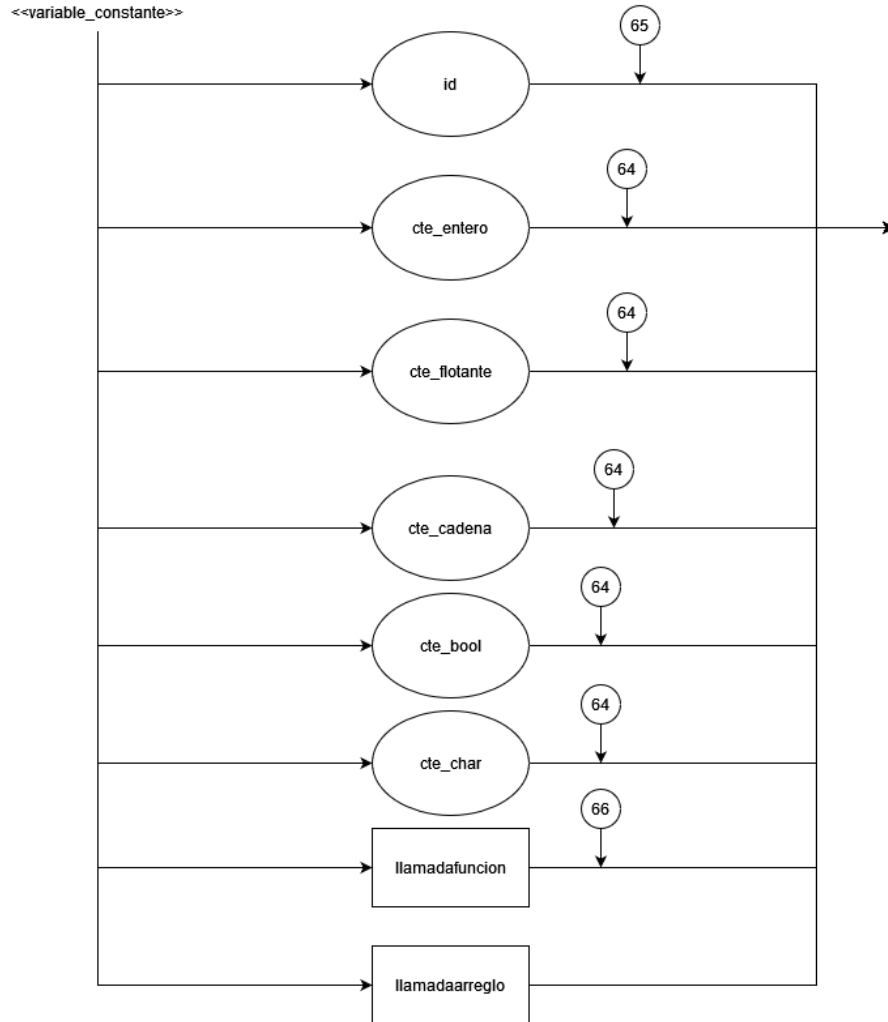
**Figura 3.24**

Diagrama Variable\_Constante

64. Checar si la constante existe en la tabla de constantes, si no existe se agrega a la tabla de constantes con la dirección virtual apropiada. Agregar la dirección virtual de la constante a la pila de operandos
65. Checar si el id existe en el scope local o global. Si existe agregar su dirección virtual a la pila de operandos.
66. Checa si la llamada a función es de retorno. Si es de retorno crear una variable temporal con el valor actual de la variable global de la función y agregar el temporal a la pila de operandos

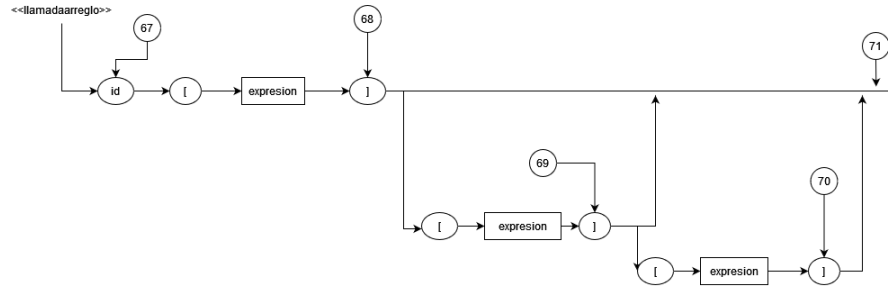
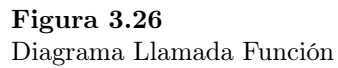
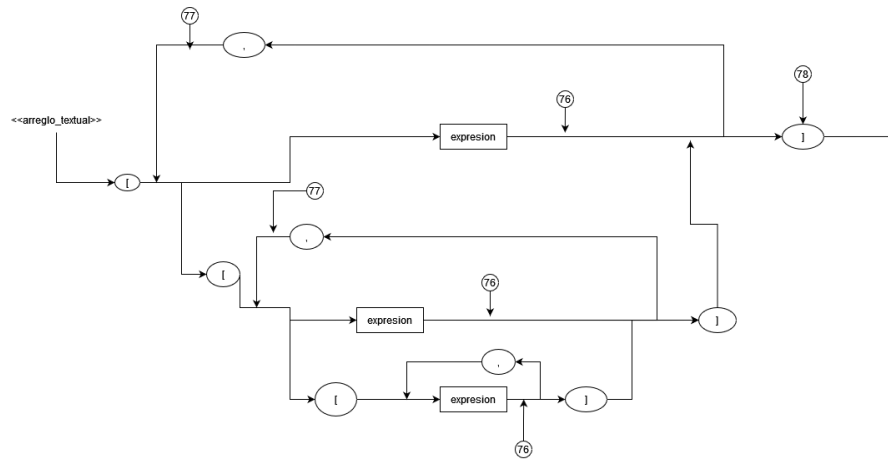
**Figura 3.25**

Diagrama Llamada Arreglo

67. Se revisa si el identificador es un arreglo.
68. Se hace pop a la pila de operandos y se revisa que el elemento recibido sea entero. Si es entero se agrega a la fila de subíndices.
69. Se hace pop a la pila de operandos y se revisa que el elemento recibido sea entero. Si es entero se agrega a la fila de subíndices.
70. Se hace pop a la pila de operandos y se revisa que el elemento recibido sea entero. Si es entero se agrega a la fila de subíndices.
71. Se checa que la cantidad de subíndices leídos coincida con las dimensiones del arreglo, de ser así itera por los subíndices generando los cuádruplos de verificación y las sumas apropiadas para calcular la dirección virtual del arreglo accesado.



72. Checar si es una función especial. Si es función especial, crear una variable global con el nombre y tipo de la función y la dirección de memoria apropiada.
73. Obtener la información de los parámetros del directorio de funciones y generar el cuádruplo de era.
74. Hacer pop a la pila de operandos y meter el elemento obtenido a una fila de parámetros.
75. Comparar si los elementos de la fila de parámetros coinciden con los parámetros de la función. Si los elementos coinciden generar los cuádruplos de parameter apropiados para cada elemento de la fila.



**figura 3.27**

Diagrama Arreglo\_Textual

76. Se checa el tipo de la expresión realizada y se mete a un stack auxiliar de tipos y se checa la congruencia de tipo con los demás leídos (si hay).
77. Se checa la congruencia de la dimensión del elemento leído, si las dimensiones son diferentes se levanta un error. Igual se guarda el tamaño de las dimensiones leídas.
78. Se calcula el tamaño del arreglo/cubo/matriz

---

### **3.5. Descripción de Administración de Memoria usado en la compilación**

Para la administración de memoria durante compilación se usaron principalmente los siguientes elementos:

1. Directorio de Funciones
2. Vector de Cuadros y su contador
3. Tabla de Constantes y su espejo
4. Cubo Semantico
5. Contadores Globales de Variables Globales y temporales
6. Máximos de Variables y las direcciones de memoria
7. Vector de relación línea código fuente a cuadro
8. Variables auxiliares para el manejo de arreglos
9. Pila de operadores, Pila de operandos y Pila de tipos
10. Diccionario de funciones especiales

Cada uno de estos se va a explicar con más detalle en las siguientes subsecciones.

#### **3.5.0.1. Directorio de Funciones y el Directorio de Funciones Especiales**

Para implementar el directorio de funciones se requería una estructura de datos que fuera de acceso rápido y que pudiera contener muchos tipos de variables e información dentro de ella. Afortunadamente en el lenguaje de Python no hay mucha restricción sobre el tipo de datos que se guardan en sus diferentes estructuras. Dado lo que se necesitaba que fuera el directorio de funciones se decidió utilizar la estructura de Diccionario de Python. Este nos permitiría guardar cualquier información necesaria con una llave única, además permite guardar cualquier otra estructura de datos dentro de cada entrada de un diccionario permitiendo una gran flexibilidad de la información que se puede guardar. También los diccionarios son de las estructuras son de rápido acceso para consultas.

nombre	tipo	address	params	varres	tmpres	vartab
global	vacio	-	-	1,0,1,1	-	Dict de python de var
pop	vacio	-	Dict de python de params	1,0,3,1	14, 1, 2, 7, 0	Dict de python de var
principal	vacio	-	Dict de python de params	1,0,1,1	14, 0, 0, 7, 0	Dict de python de var

**Cuadro 3.2**

Representación Lógica del directorio de funciones

Cada entrada de la tabla tiene cinco elementos importantes. El primero es el tipo de la función, este ayuda a identificar si la función tiene retorno o es vacía. La segunda es address. Esta es la dirección virtual de la función si es función de retorno. La tercera es params que es un diccionario de Python que tiene la información del tipo e información de arreglos si el parámetro es un arreglo. El tercero y cuarto elemento son una serie de contadores que representan los recursos locales y temporales respectivamente que tiene una función. El quinto elemento es la tabla de variables que contiene la información de las variables locales de la función.

En el caso de la entrada global, es la única entrada del directorio que no cuenta con información de variables temporales.

Para facilitar la integración de las funciones especiales en compilación también se usó una estructura muy similar al directorio de funciones. De esta manera se pueden usar muchos de los métodos para recorrer el directorio de funciones en el directorio de funciones especiales. Realmente la única diferencia que existe entre los dos es que el directorio de funciones especiales no cuenta con un contador de variables temporales ni una tabla de variables, ya que para ejecutar estas funciones en la máquina virtual solo se necesita saber la información de los parámetros.

nombre	tipo	address	params	varres
normal	flotante	1006	Dict de python de params	2,0,0,0
modulo	flotante	1007	Dict de python de params	2,0,0,0

**Cuadro 3.3**

Representación Lógica del directorio de funciones especiales

**3.5.0.2. Vector de Cuádruplos y su contador**

Para representar los cuádruplos se utilizaron la estructura de tuplas de Python. Pero un programa no cuenta con un solo cuádruplo sino una serie de múltiples cuádruplos ordenados que indican las instrucciones a ejecutar a la máquina virtual. Para lograr esto se creó una lista de tuplas, a la cual se le va agregando como en una fila cada cuádruplo generado. Para tener un mejor control sobre la cantidad de cuádruplos generados también se mantuvo una variable auxiliar global contadora de cuádruplos.

```
( 'GOTO' , ' ' , ' ' , 20)
```

**Figura 3.28**

Ejemplo de una tupla de Cuádruplo

```
cuadрупlos.append(( 'GOTO' , res , ' ' ))
global sclines
sclines.append(p.lineno(1))
psaltos.append(cuadcount)
cuadcount += 1
```

**Figura 3.29**

Ejemplo de generación de un cuádruplo

**3.5.0.3. Tabla de Constantes y su espejo**

Como el directorio de funciones, la tabla de constantes también va a ser una estructura que va a ser consultada muy frecuente mente. Como se había mencionado previamente, el diccionario de Python también es una buena solución para poder guardar y acceder con mucha facilidad información de manera frecuente. Por ello se optó por usar también el diccionario de Python para representar la tabla de constantes.

Constante	Dirección
25000	2
25001	1
310000	"Hola"

**Cuadro 3.4**

Representación Lógica de la tabla de constantes



```
{
    "2": 25000,
    "1": 25001,
    "3": 25002,
    "1001": 25003,
    "1003": 25004,
    "3000": 25005,
    "0": 25006,
    "5": 25007,
    "\"Factorial_de_n\\n\"": 31000,
    "10": 25008,
    "\"Factorial_Secuencial_:\"": 31001,
    "\"-----\\n_hola_aka_=\"": 31002
}
```

**Figura 3.30**

Ejemplo de el diccionario de constantes

Como Python no es un lenguaje fuertemente tipado, para generar las llaves del diccionario se transforma a string la constante para poderla encontrar fácilmente. También para reducir el tiempo de ejecución a la hora de crear el archivo obj, se mantiene una copia de la tabla de constantes con el formato correcto para la máquina virtual.

```

{
  "25000": 3,
  "25001": 1,
  "25002": 9,
  "25003": 9003,
  "25004": 14001,
  "25005": 9006,
  "25006": 10,
  "25007": 30,
  "25008": 2,
  "25009": 1000000,
  "25010": 20,
  "30000": true,
  "31000": "\"Hola_Mundo\"",
  "25011": 0,
  "31002": "\"Iteracion_por_la_matriz\"",
  "31003": "\"Iteracion_por_el_cubo\"",
  "31004": "\"\\t—Dim2\"",
  "31005": "\"\\t\\t—Dim_3\"",
  "31006": "\"Probando_arreglos_textuales\"",
  "29000": "'a'",
  "25012": 4,
  "25013": 5,
  "25014": 6,
  "29001": "'b'",
  "29002": "'c'",
  "30001": false,
  "25015": 127
}

```

**Figura 3.31**

Ejemplo de el diccionario de constantes para la máquina virtual

### 3.5.1. Tabla de variables

Para la tabla de variables también se necesitó una estructura de datos que se rápida de acceder y modificar. Como se mencionó con el directorio de funciones, el diccionario de Python resulto ser una estructura de datos que se adaptaba muy bien a solucionar este problema. Por ello también se utilizó un diccionario de Python para representar la tabla de variables.

nombre	tipo	dirección	dims	dimlen
a	entero	1000	-	-
b	entero	1001	2	[3,4], [4,0]

#### Cuadro 3.5

Representación Lógica de la tabla de variables

El diccionario de la tabla de variables constaba de que cada llave fuera el nombre de la variable y que dentro de esa llave se guardara otro diccionario con la información de la variable. Este diccionario tiene lo que es el tipo de la variable, su dirección de memoria, y si es un arreglo, las dimensiones del arreglo y la información para calcular la indexación.

#### 3.5.1.1. Cubo Semántico

Para el cubo semántico se decidió ir por una estructura que pudiera ser accesada rápidamente y que pudiera dar el resultado de una operación con llaves sencillas. Ya que se trabajó en el lenguaje de Python, se aprovechó de la estructura nativa de diccionarios. Estos pueden ser accesados rápidamente y se le pueden asignar llaves a cada entrada. Para hacer el código legible se decidió estructurar el cubo como un diccionario de diccionarios. De esta manera la llamada al cubo semántico sería acciendo al cubo con la primera llave siendo el operador, y las otras dos llaves siendo el operando izquierdo y el operando derecho.

**Cuadro 3.6**

Tabla representando el cubo semántico

OpIzq	OpDer	=		&&	<	>	<=	>=	!=	==	+	-	*	\
ent	ent	ent	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
ent	flot	ent	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
ent	char	ent	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
ent	bool	ent	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
ent	cadena	err	err	err	err	err	err	err	err	err	err	err	err	err
flot	ent	flot	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
flot	flot	flot	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
flot	char	flot	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
flot	bool	flot	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
flot	cadena	err	err	err	err	err	err	err	err	err	err	err	err	err
char	ent	char	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
char	flot	char	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
char	char	char	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
char	bool	char	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
char	cadena	err	err	err	err	err	err	err	err	err	err	err	err	err
bool	ent	bool	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
bool	flot	bool	bool	bool	bool	bool	bool	bool	bool	bool	flot	flot	flot	flot
bool	char	bool	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
bool	bool	bool	bool	bool	bool	bool	bool	bool	bool	bool	ent	ent	ent	ent
bool	cadena	err	err	err	err	err	err	err	err	err	err	err	err	err
cadena	ent	err	err	err	err	err	err	err	err	err	err	err	err	err
cadena	flot	err	err	err	err	err	err	err	err	err	err	err	err	err
cadena	char	err	err	err	err	err	err	err	err	err	err	err	err	err
cadena	bool	err	err	err	err	err	err	err	err	err	err	err	err	err
cadena	err	err	err	err	err	err	err	err	err	err	err	err	err	err

- ent : Entero
- flot : Flotante
- err : Error

```
rettype = ptipo.pop()
if (cubosem['='][functipo][rettype] != 'error'):
    dprint('Cubo_dice:⌋', cubosem['='][functipo][rettype])
    retop = pilaoperand.pop()
    .....
```

**Figura 3.32**

Ejemplo del uso de el Cubo Semántico

```

cubosem = {
  '=': {
    'entero': {
      'entero': 'entero',
      'flotante': 'entero',
      'char': 'entero',
      'bool': 'entero',
      'cadena': 'error'
    },
    'flotante': {
      'entero': 'flotante',
      'flotante': 'flotante',
      'char': 'flotante',
      'bool': 'flotante',
      'cadena': 'error'
    },
    'char': {
      'entero': 'char',
      'flotante': 'char',
      'char': 'char',
      'bool': 'char',
      'cadena': 'error'
    },
    'bool': {
      'entero': 'bool',
      'flotante': 'bool',
      'char': 'bool',
      'bool': 'bool',
      'cadena': 'error'
    },
    'cadena': {
      'entero': 'error',
      'flotante': 'error',
      'char': 'error',
      'bool': 'error',
      'cadena': 'err'
    },
  },
  ....

```

**Figura 3.33**

Muestra de la primera entrada del cubo semántico

### 3.5.1.2. Contadores Globales de Variables Globales y temporales, Máximos de Variables y las Direcciones de Memoria

Para mantener un registro del uso de recursos dentro del programa durante el proceso de compilación se crearon variables globales que funcionaban como contadores sobre cada tipo de variable. Cada vez que se generaba una nueva variable se aumentaba el contador y se verificaba que no se pasara de los máximos del tipo permitidos. Estos contadores se dejaron como variables globales porque había múltiples acciones semánticas que requerían consultar o actualizar sus valores. Para mantener legibilidad y por simplicidad se optó por dejarlo como variables globales.

Globales				
Enteros	Flotantes	Caracteres	Booleanos	Apuntadores
1	2	0	3	0
Temporales				
Enteros	Flotantes	Caracteres	Booleanos	Apuntadores
8	2	0	6	0

### Cuadro 3.7

Representación Lógica de los Contadores Globales

Similar que con los contadores globales también se dejó los máximos de variables y de direcciones de memoria como variables globales para mantener legibilidad y simplicidad en el código. Como muchas de las acciones semánticas también interactuar con ellas se mantienen globales.

### 3.5.1.3. Vector de relación línea código fuente a cuádruplo

Para desarrollar mensajes de error más detallados en la máquina virtual, se necesitaba saber en qué línea de código se causó el error. Para resolver este problema se creó una lista que aprovecha el rastreador de tokens del lexer de *ply* para guardar el número de la línea del código fuente en relación con un cuádruplo. Esta lista va creciendo a la par que la lista de cuádruplos y es pasada como información a la máquina virtual para que pueda usarla en sus mensajes de error.

```
// La lista de cuádruplos
Cuádruplos = [( 'GOTO', 'MAIN', '', '' ),
               ("*", 25006, 25007, 17000),
               ("/", 17000, 25007, 17001),
               ("*", 17001, 25008, 17002),
               .....]
// Lista de la línea de código fuente que la creo
sclines = [1, 9, 9, 9, 9, .....]
```

**Figura 3.34**

Ejemplo de como se ve la relación de cuádruplos contra líneas de código fuente

**3.5.1.4. Variables auxiliares para el manejo de arreglos**

Para el cálculo de la información de la indexación de arreglos se optó por dejar las variables como variables globales. Como no son muchas las variables para el cálculo de la fórmula de indexación, ya que este lenguaje se limita hasta cubos nada más, no se vio necesario crear una estructura extra para manejar su información. De igual manera, como son múltiples acciones semánticas que utilizan estas variables por conveniencia se dejaron como variables globales.

R	1
at1	2
at2	3
at3	0

**Cuadro 3.8**

Representación Lógica de las Variables Auxiliares de arreglos

**3.5.1.5. Pila de operadores, Pila de operandos y Pila de tipos**

Para el manejo de la lógica de expresiones como visto en clase se necesitaba utilizar una estructura stack para el manejo de las acciones semánticas. Python no cuenta con una estructura stack, pero cuenta con métodos que simulan a un stack en su estructura de lista. Por ello se manejó la Pila de Operadores, Pila de Operandos y la Pila de tipos como listas las cuales solo se podían interactuar con usando los métodos de *pop* y *append* que tienen las listas.



Pila de Operadores	+	*	....
Pila de Operandos	A	1000	x_inicial ....
Pila de Tipos	entero	entero	entero ....

**Cuadro 3.9**  
Representación Lógica de las Pilas



# 4

## Descripción de la máquina virtual

### CONTENTS

4.1	Equipo de cómputo, lenguaje y utilerías especiales usadas .....	75
4.2	Descripción del proceso de Administración de Memoria en ejecución .....	75
4.2.1	La Clase Memoria .....	76
4.2.2	Manejo de memoria global, local y temporal .....	76
4.2.3	El stack de memorias .....	77
4.3	Constantes .....	78

En esta parte se describe con más detalle lo que se utilizo para la máquina virtual y su manejo de memoria

### 4.1. Equipo de cómputo, lenguaje y utilerías especiales usadas

Una PC con Windows 10, se utilizó el lenguaje de programación Python 3.10 con apoyo de las librerías de *PLY*, *Numpy*, *re*, *json*, *SciPy* y *Matplotlib*.

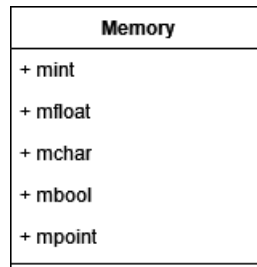
La máquina virtual utiliza las librerías de *emphJSON*, *SciPy*, *NumPy* y *Matplotlib*. Utiliza la librería de *JSON* para leer el archivo generado por el compilador y con toda la información del archivo va ejecutando el código intermedio generado por el compilador. Cuando la máquina lee una función especial utiliza las librerías de *SciPy*, *NumPy* y *Matplotlib* para ejecutar dichas funciones.

## 4.2. Descripción del proceso de Administración de Memoria en ejecución

Para lograr ejecutar de manera correcta el código compilado es importante poder representar y manejar el espacio de memoria en la máquina virtual, de manera en la que se pueda convertir las direcciones virtuales asignadas por el compilador a espacios de memoria en la máquina.

### 4.2.1. La Clase Memoria

Para manejar la memoria en la máquina virtual se desarrollo la clase Memory, la cual representa un conjunto de espacios de memoria para enteros, flotantes, caracteres, booleanos y apuntadores como cinco listas. Esta clase recibe un número representando la cantidad de espacios para cada tipo de variable en su constructor y se genera una lista con el tamaño apropiado.



**Figura 4.1**

Diagrama de Clase de Memory

De esta manera se puede mapear la memoria de la máquina virtual, las listas que contienen la información, a las direcciones virtuales del compilador realizando la siguiente formula.

$$Espacio\_en\_el\_arreglo = Dir\_Virtual - Dir\_Virtual\_Base\_del\_tipo$$

Con esta formula se puede sacar la posición en el arreglo se encuentra el valor de la variable.

### 4.2.2. Manejo de memoria global, local y temporal

Ahora para representar el espacio de memoria en la máquina virtual se necesita primero plantear el como se debe de manejar los scopes de las variables. Para manejar las variables globales basta con crear una instancia de Memory para representarlas, pero se empieza a complicar la situación cuando se intenta representar el scope de la función principal o de cualquier otra función. Esto

```

globalMem = Memory(1,6,0,3)
# El primero representa el local y el segundo el temporal
principalMem = [Memory(1,6,0,3), Memory(7,6,0,8)]

```

**Figura 4.2**

Representación en código de las memorias

se debe a que Memory solo cuenta con un conjunto de arreglos, pero estos espacios de memoria cuentan no solo con variables locales sino que también con variables temporales. Para lidiar con este problema el espacio de memoria de una función se representa por un vector con dos instancias de Memory. La primera representando las variables locales y la segunda representando las variables temporales.

Con este vector ahora ya se puede manejar todos los posibles scopes de la memoria.

#### 4.2.3. El stack de memorias

Habiendo resuelto el problema de las variables temporales y locales, surge un nuevo problema. ¿Cómo se sabe en qué memoria se está trabajando actualmente? Para resolver esto se implemento una pila, la cual maneja el contexto actual con el que se esta trabajando. La idea de usar una pila viene a la naturaleza del uso de espacios de memoria cuando se llama a una función. Lógicamente si seguimos el algoritmo que generan los cuádruplos, la máquina virtual va a parar la ejecución del código actual y va a saltar a realizar el código de la función. Después de acabar la función la máquina debe de regresar a donde estaba y continuar el código. Esto se podría resolver con una variable auxiliar, pero surge un problema si se utiliza una variable auxiliar cuando una función llama a otra función.

```

# Se llama funcion 1
auxMem = principalMem
.....
# Se llama funcion 2
auxMem = funcMem
# Ya se perido la memoria de principal

```

**Figura 4.3**

Representación de el uso de una variable auxiliar

Si solo se usa una variable auxiliar la nueva función va a causar que la memoria de la función que se anda ejecutando sobre escriba a la memoria que la llamo. Pero este problema se puede resolver con una pila. Si una función

llama a otra función en su ejecución lo que se puede hacer es ir guardando las memorias en la pila y cuando se dejen de utilizar se remueven del tope de la pila y se continúa utilizando el siguiente elemento en la pila. De esta manera se resuelve el problema de cuando una función llama a otra función.

```
# Se llama funcion 1
auxMem.append(principalMem)
.....
# Se llama funcion 2
auxMem.append(funcMem)
# Ahora solo se tiene que hacer pop al stack para obtener el contexto p
```

#### Figura 4.4

Representación de el uso de una pila para el manejo de memoria

Esta solución no solo aplica para las funciones que son dependientes a otras, sino que también soluciona el problema de el manejo de memorias en funciones recursivas.

---

Pila de Memorias		MemPrincipal	MemFuncRec	MemFuncRec	.....
------------------	--	--------------	------------	------------	-------

---

#### Cuadro 4.1

Representación de la pila en llamadas recursivas

### 4.3. Constantes

Para el manejo de constantes en la memoria de la máquina virtual no fue necesario crear un espacio de memoria con la clase Memory. Para reducir la cantidad de instancias generadas se opto por pasar la tabla de constantes de compilación al archivo obj que resulta del proceso. De esta manera se puede recuperar esta información cuando se lee el archivo. Esta información es guardada en un diccionario que puede ser accesado con la dirección virtual como la llave y nos ahorra el calculo de obtener la posición que debería de tener en memoria.

Parte III

Pruebas del  
funcionamiento del  
lenguaje





# 5

## *Pruebas del funcionamiento del lenguaje*

### CONTENTS

5.1	Programa Factorial .....	81
5.2	Programa Fibonacci .....	83
5.3	Programa Búsqueda y organización de arreglo .....	85
5.4	Programa Multiplicación de Matrices .....	88
5.5	Programa Uso de Funciones de estadística .....	90

En esta parte se demuestra código escrito en el lenguaje, y como el compilador y la máquina virtual interactúan con este código fuente.

### 5.1. Programa Factorial

En este programa se implementaron dos formas para calcular la factorial de un número. Una usando una función recursiva y la otra usando un método secuencial. A continuación, se presenta el código fuente:

```

programa factorial {

entero n;
entero r[ 2 ];
entero l[ 3 ][ 2 ];
flotante g[1][1][1];

funcion entero factorial (entero num){}{
    si(num > 0){
        regresar num * $factorial(num-1);
    }sino{
        regresar 1;
    }
}

principal funcion
{
    entero aka ,aux ,aux2;

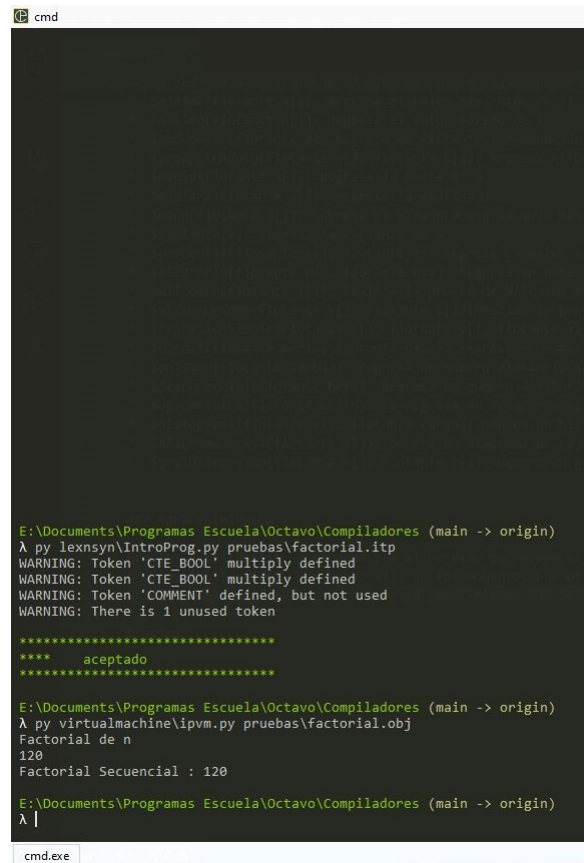
}
{
    n = $factorial(5);
    imprimir(" Factorial de n \n");
    imprimir(n);
    aka = 10-n;
    10 / (-aka) ;
    aka = 5;
    aux2 = aka;
    aux = 1;
    mientras(aux <= aux2){
        si (aux2 - aux > 0 ){
            aka = aka * (aux2-aux);
        }sino{
            aka = aka * 1;
        }
        aux = aux + 1;
    }
    imprimir(" Factorial Secuencial : ",aka);
}
}

```

**Figura 5.1**

Código fuente del programa factorial

A continuación se muestra su ejecución en consola:



```
cmd

E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ py lexsyn\IntroProg.py pruebas\factorial.itp
WARNING: Token 'CTE_BOOL' multiply defined
WARNING: Token 'CTE_BOOL' multiply defined
WARNING: Token 'COMMENT' defined, but not used
WARNING: There is 1 unused token

*****
****  aceptado
*****

E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ py virtualmachine\ipvm.py pruebas\factorial.obj
Factorial de n
120
Factorial Secuencial : 120

E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ |
```

**Figura 5.2**

El programa factorial compilado y ejecutado

---

## 5.2. Programa Fibonacci

En este programa se implementaron dos formas para calcular el número  $n$  de la serie de Fibonacci. Una usando una función recursiva y la otra usando un método secuencial. A continuación, se presenta el código fuente:

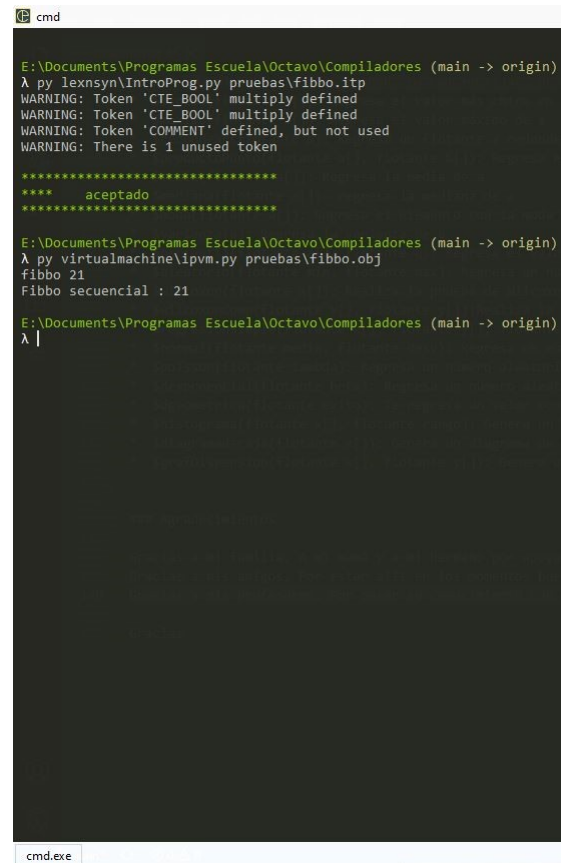
```
programa fibbo {
funcion entero fibbo(entero n){}{
    si(n == 0){
        regresar 0;
    }
    si(n == 1){
        regresar 1;
    }
    regresar $fibbo(n-1) + $fibbo(n-2);
}
principal funcion {
    entero pri,seg,ter,obj,aux;
} {
    obj = 8;
    imprimir("fibbo ", $fibbo(obj));

    pri = 0;
    seg = 1;
    aux = 1;
    mientras( aux < obj){
        ter = pri + seg;
        pri = seg;
        seg = ter;
        aux = aux + 1;
    }
    imprimir("Fibbo secuencial : ",ter);
}
}
```

**Figura 5.3**

Código fuente del programa fibbo

A continuación se muestra su ejecución en consola:



```
cmd
E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ py lexnsyn\IntroProg.py pruebas\fibbo.itp
WARNING: Token 'CTE_BOOL' multiply defined
WARNING: Token 'CTE_BOOL' multiply defined
WARNING: Token 'COMMENT' defined, but not used
WARNING: There is 1 unused token

*****
****  aceptado
*****

E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ py virtualmachine\ipvm.py pruebas\fibbo.obj
fibbo 21
Fibbo secuencial : 21

E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ |
```

**Figura 5.4**

El programa fibbo compilado y ejecutado

---

### 5.3. Programa Búsqueda y organización de arreglo

En este programa se implementaron una función de búsqueda en un arreglo y un Bubble Sort para organizar un arreglo. A continuación, se presenta el código fuente:

```

programa arraysearchnsort{
    entero a[10];
    funcion vacio imprimirArr(entero a[10]){
        entero x;
    }{
        por(x = 0; x < 10; x = x+1){
            imprimir("a[" ,x," ] = " ,a[x]);
        }
    }

    funcion vacio encontrar(entero x, entero a[10]){
        entero i ,aux,aux2;
        bool loEncontre;
    }{
        imprimir("\n-----\nBuscando " ,x," en el arreglo :");
        $imprimirArr(a);
        imprimir("\n\n");
        aux = -1;
        loEncontre = falso;
        por(i = 0; i < 10; i = i +1){
            si(a[i] == x){
                aux2 = i;
                i = 10+1;
                loEncontre = verdadero;
            }
        }
        si (loEncontre){
            imprimir("Encontro " ,x," en a[" ,aux2," ]");
        }sino{
            imprimir("No se encontro el numero");
        }
    }
}

principal funcion {
    entero i, j, aux;
} {

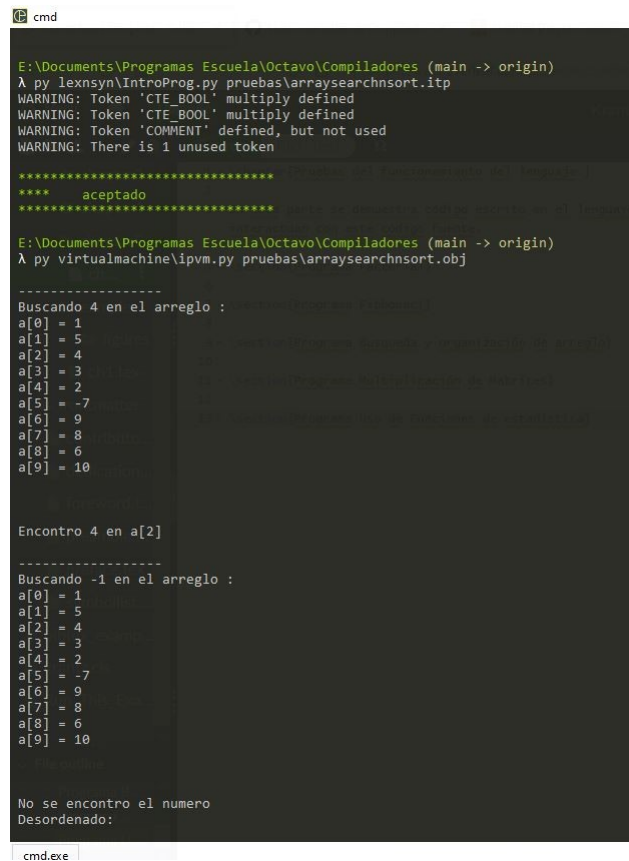
    a = [1,5,4,3,2,-7,9,8,6,10];
    $encontrar(4,a);
    $encontrar(-1,a);
    imprimir("Desordenado:\n");
    $imprimirArr(a);
    por(i = 0; i < 10; i = i+1){
        por(j = 0; j < 10 -i -1; j = j+1){
            si(a[j] > a[j+1]){
                aux = a[j];
                a[j] = a[j+1];
                a[j+1] = aux;
            }
        }
    }
    imprimir("\n\nOrdenado:\n");
    $imprimirArr(a);
    imprimir();
}
}

```

**Figura 5.5**

Código fuente del programa arraysearchnsort

A continuación se muestra su ejecución en consola:



```
cmd
E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ py lexsyn\IntroProg.py pruebas/arraysearchnsort.itp
WARNING: Token 'CTE_BOOL' multiply defined
WARNING: Token 'CTE_BOOL' multiply defined
WARNING: Token 'COMMENT' defined, but not used
WARNING: There is 1 unused token

***** Pruebas del funcionamiento del lenguaje I *****
**** aceptado ****
***** para se demuestre correcto respecto al lenguaje *****

E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ py virtualmachine\ipvm.py pruebas/arraysearchnsort.obj

-----
Buscando 4 en el arreglo : [1, 5, 4, 3, 2, -7, 9, 8, 6, 10]
a[0] = 1
a[1] = 5
a[2] = 4
a[3] = 3
a[4] = 2
a[5] = -7
a[6] = 9
a[7] = 8
a[8] = 6
a[9] = 10

Encontro 4 en a[2]

-----
Buscando -1 en el arreglo :
a[0] = 1
a[1] = 5
a[2] = 4
a[3] = 3
a[4] = 2
a[5] = -7
a[6] = 9
a[7] = 8
a[8] = 6
a[9] = 10

No se encontro el numero
Desordenado:
```

**Figura 5.6**

El Búsqueda y Sort compilado y ejecutado parte 1

```
No se encontro el numero
Desordenado:
a[0] = 1
a[1] = 5
a[2] = 4
a[3] = 3
a[4] = 2
a[5] = -7
a[6] = 9
a[7] = 8
a[8] = 6
a[9] = 10

Ordenado:
a[0] = -7
a[1] = 1
a[2] = 2
a[3] = 3
a[4] = 4
a[5] = 5
a[6] = 6
a[7] = 8
a[8] = 9
a[9] = 10

E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ |
cmd.exe
```

**Figura 5.7**

El Búsqueda y Sort compilado y ejecutado parte 2

---

## 5.4. Programa Multiplicación de Matrices

En este programa se implemento una función de multiplicación de matrices. A continuación, se presenta el código fuente:



```

programa matmul{
    entero matA[2][3];
    entero matB[3][4];
    entero r1, comun ,c2;
    funcion vacio imprimirMatA(entero a[2][3]){
        entero x,y;
    }{
        por(x = 0; x < 2; x = x+1){
            por(y = 0; y < 3; y = y+1){
                imprimir("mat[" ,x ,"][" , y, "] = ",a[x][y]);
            }
        }
    }

    funcion vacio imprimirMatB(entero a[3][4]){
        entero x,y;
    }{
        por(x = 0; x < 3; x = x+1){
            por(y = 0; y < 4; y = y+1){
                imprimir("mat[" ,x ,"][" , y, "] = ",a[x][y]);
            }
        }
    }

    funcion vacio imprimirMat(entero a[2][4]){
        entero x,y;
    }{
        por(x = 0; x < 2; x = x+1){
            por(y = 0; y < 4; y = y+1){
                imprimir("mat[" ,x ,"][" , y, "] = ",a[x][y]);
            }
        }
    }

    principal funcion{
        entero matR[2][4];
        entero i,j,k;
    }{
        r1 = 2;
        comun = 3;
        c2 = 4;
        matA = [[1,2,3],[4,5,6]];
        imprimir(" Matriz A :");
        $imprimirMatA(matA);
        matB = [[1,2,3,4],[5,6,7,8],[9,10,11,12]];
        imprimir(" Matriz B :");
        $imprimirMatB(matB);

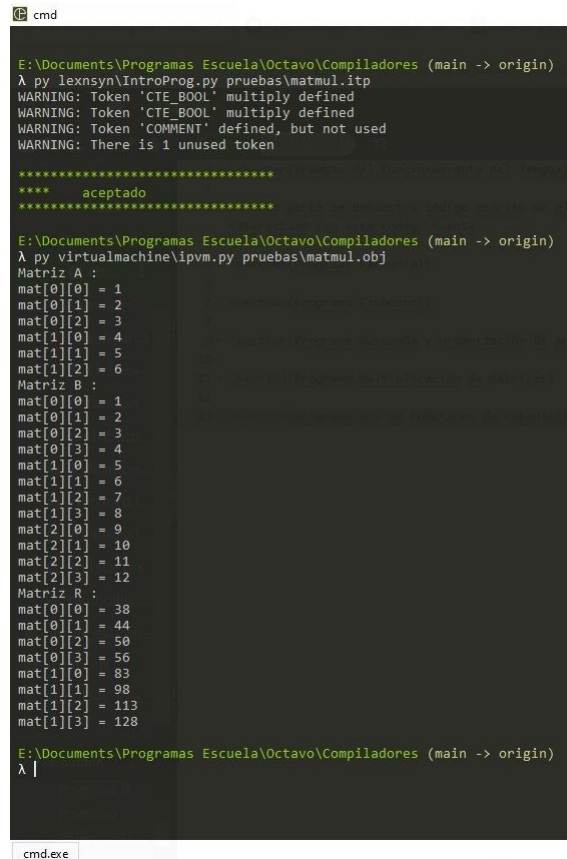
        por (i = 0; i < r1; i = i+1){ //Iterar sobre los renglones de la primera matriz
            por(j = 0; j < c2; j = j +1){ // Iterar sobre las columnas de la segunda matriz
                matR[i][j] = 0;
                por(k = 0; k < comun; k = k+1){
                    // Recorer la col/reg para hacer la multiplicacion
                    matR[i][j] = matR[i][j] + matA[i][k] * matB[k][j];
                    //matR[i][j] + matA[R_i][k] * matB[k][C_j]
                }
            }
        }

        imprimir(" Matriz R :");
        $imprimirMat(matR);
    }
}

```

**Figura 5.8**  
Código fuente del programa matmul

A continuación se muestra su ejecución en consola:



```
cmd
E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ py lexnsyn\IntroProg.py pruebas\matmul.itp
WARNING: Token 'CTE_BOOL' multiply defined
WARNING: Token 'CTE_BOOL' multiply defined
WARNING: Token 'COMMENT' defined, but not used
WARNING: There is 1 unused token

*****
****  aceptado
*****

E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ py virtualmachine\ipvm.py pruebas\matmul.obj
Matriz A :
mat[0][0] = 1
mat[0][1] = 2
mat[0][2] = 3
mat[1][0] = 4
mat[1][1] = 5
mat[1][2] = 6
Matriz B :
mat[0][0] = 1
mat[0][1] = 2
mat[0][2] = 3
mat[0][3] = 4
mat[1][0] = 5
mat[1][1] = 6
mat[1][2] = 7
mat[1][3] = 8
mat[2][0] = 9
mat[2][1] = 10
mat[2][2] = 11
mat[2][3] = 12
Matriz R :
mat[0][0] = 38
mat[0][1] = 44
mat[0][2] = 50
mat[0][3] = 56
mat[1][0] = 83
mat[1][1] = 98
mat[1][2] = 113
mat[1][3] = 128

E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ |
```

**Figura 5.9**  
El matmul compilado y ejecutado

## 5.5. Programa Uso de Funciones de estadística

En este programa se implementaron unas pruebas de Wilcoxon. Se crearon tres arreglos que simulaban muestras las cuales se popularon con números aleatorios pertenecientes a una distribución y se desplego la información básica de cada arreglo. Al final se hicieron unas pruebas de wilcoxon para probar cual de las muestras se acomodaba mejor.

A continuación, se presenta el código fuente:

```

programa pruebasDeWilcoxon{
  // La declaracion de variables
  flotante muestra1[100];
  flotante muestra2[100];
  // Declaracion de funciones
  funcion vacio imprimirDatos( flotante arr[100] ){
    flotante prom, mod, med;
  }{
    prom = $media(arr);
    mod = $moda(arr);
    med = $mediana(arr);
    imprimir(" La media de esta distribucion es : ", prom);
    imprimir(" La mediana de esta distribucion es : ", med);
    imprimir(" La moda de esta distribucion es : ", mod);
    imprimir(" El cuartil del 50 de los datos : ", $percentil(arr,50));
  }
  //Funcion principal
  principal funcion {
    flotante muestra3[100];
    entero i;
    flotante p;
  }{
    por(i = 0; i < 100; i = i+1){
      muestra1[i] = $normal(3.5,0.6);
      muestra2[i] = $dexponencial(4);
      muestra3[i] = $poisson(2);
    }

    imprimir("Muestra 1 : Normal");
    $imprimirDatos(muestra1);
    imprimir("Muestra 2 : Exponencial");
    $imprimirDatos(muestra2);
    imprimir("Muestra 3 : Poisson");
    $imprimirDatos(muestra3);

    //Pasar una prueba de wilcoxon p > 0.5
    p = $wilcoxonComp(muestra1,muestra2);
    imprimir("\n\nMuestra 1 v Muestra 2");
    si(p > 0.5){
      imprimir("Muestra 1 es la que mejor representa los datos");
    }sino{
      imprimir("Muestra 2 es la que mejor representa los datos");
    }

    p = $wilcoxonComp(muestra1,muestra3);
    imprimir("\n\nMuestra 1 v Muestra 3");
    si(p > 0.5){
      imprimir("Muestra 1 es la que mejor representa los datos");
    }sino{
      imprimir("Muestra 3 es la que mejor representa los datos");
    }

    p = $wilcoxonComp(muestra3,muestra2);
    imprimir("\n\nMuestra 3 v Muestra 2");
    si(p > 0.5){
      imprimir("Muestra 3 es la que mejor representa los datos");
    }sino{
      imprimir("Muestra 2 es la que mejor representa los datos");
    }
  }
}

```

**Figura 5.10**  
Código fuente del programa

A continuación se muestra su ejecución en consola:

```
E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ py lexsyn\IntroProg.py pruebas\pruebasDeWilcoxon.itp
WARNING: Token 'CTE_BOOL' multiply defined
WARNING: Token 'CTE_BOOL' multiply defined
WARNING: Token 'COMMENT' defined, but not used
WARNING: There is 1 unused token
*****
****  aceptado
*****
E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ py virtualmachine\lpvm.py pruebas\pruebasDeWilcoxon.obj
Muestra 1 : Normal
La media de esta distribucion es : 3.58217248805351 La mediana de esta distribucion es : 3.5205081442585566 La moda de esta distribucion es : 3.73457931963157
Q : 50.0
El cuartil del 50 de los datos : 3.5205081442585566
Muestra 2 : Exponencial
La media de esta distribucion es : 4.213870985229987 La mediana de esta distribucion es : 3.0053672261925315 La moda de esta distribucion es : 3.817501876873192
Q : 50.0
El cuartil del 50 de los datos : 3.0053672261925315
Muestra 3 : Poisson
La media de esta distribucion es : 1.98 La mediana de esta distribucion es : 2.0 La moda de esta distribucion es : 2
Q : 50.0
El cuartil del 50 de los datos : 2.0

Muestra 1 v Muestra 2
Muestra 2 es la que mejor representa los datos

Muestra 1 v Muestra 3
Muestra 3 es la que mejor representa los datos

Muestra 3 v Muestra 2
Muestra 2 es la que mejor representa los datos
E:\Documents\Programas Escuela\Octavo\Compiladores (main -> origin)
λ |
```

**Figura 5.11**

El programa de pruebasDeWilcoxon compilado y ejecutado

Parte IV

Ejemplos de  
Documentación en código



Ejemplo de funciones documentadas en el compilador:

```
# Func : getArrayData
# Params : Una dirección Virtual addr
# ret : La información de las dimensiones de un arreglo
# Desc : Regresa la información de las dimensiones de un arreglo
def getArrayData(addr):
    global dirfunc
    global currscope
    res = {}

    for var in dirfunc[currscope]['vartab'].values():

        if addr['address'] == var['address']:
            res['size'] = var['size']
            res['dimlen'] = var['dimlen']
            res['dims'] = var['dims']
            return res
    for var in dirfunc['global']['vartab'].values():

        if addr['address'] == var['address']:

            res['size'] = var['size']
            res['dimlen'] = var['dimlen']
            res['dims'] = var['dims']
            return res
    return res

# Func : addConst
# Param : Una constante cte, un tipo tipo y la línea del token line
# Desc : Agrega una constante a la tabla de constante y le asigna una dirección
# virtual de acuerdo al tipo de la constante.

def addConst(cte,tipo,line):
    # Rango de memoria constantes
    global constint
    global constfloat
    global constchar
    global constbool
    global conststring
    # Contadores constantes
    global cteintcount
    global ctefloatcount
    global ctecharcount
    global cteboolcount
    global ctestringcount
    # Maximos de variables
    global INTMAX
    global FLOATMAX
    global CHARMAX
    global BOOLMAX
    global STRINGMAX
    #Tabla const
    global ctetab
    global objctetab

    # Si la llave no es un string hacer una llave string
    if type(cte) is not str:
        ctkey = str(cte)
    else:
        ctkey = cte
    # Agregarla de acuerdo al tipo y con su dirección de memoria adecuada
    if tipo == 'entero':
        if cteintcount < INTMAX:
            ctetab[ctkey] = constint + cteintcount
            objctetab[ctetab[ctkey]] = cte
            cteintcount += 1
```

```

        return ctetab[ctkey]
    else:
        printerror(
            "Error de Semantica: sobrepaso el limite de constantes declaradas en la linea %r" % (line))
elif tipo == 'flotante':
    if ctefloatcount < FLOATMAX:
        ctetab[ctkey] = constfloat + ctefloatcount
        objctetab[ctetab[ctkey]] = cte
        ctefloatcount += 1
        return ctetab[ctkey]
    else:
        printerror(
            "Error de Semantica: sobrepaso el limite de constantes declaradas en la linea %r" % (line))
elif tipo == 'char':
    if ctecharcount < CHARMAX:
        ctetab[cte] = constchar + ctecharcount
        objctetab[ctetab[cte]] = cte
        ctecharcount += 1
        return ctetab[ctkey]
    else:
        printerror(
            "Error de Semantica: sobrepaso el limite de constantes declaradas en la linea %r" % (line))
elif tipo == 'bool':
    if cteboolcount < BOOLMAX:
        ctetab[cte] = constbool + cteboolcount
        if cte == 'verdadero':
            objctetab[ctetab[cte]] = True
        else:
            objctetab[ctetab[cte]] = False
        cteboolcount += 1
        return ctetab[ctkey]
    else:
        printerror(
            "Error de Semantica: sobrepaso el limite de constantes declaradas en la linea %r" % (line))
elif tipo == 'cadena':
    if ctestringcount < STRINGMAX:
        ctetab[ctkey] = conststring + ctestringcount
        objctetab[ctetab[ctkey]] = cte
        ctestringcount += 1
        return ctetab[ctkey]
    else:
        printerror(
            "Error de Semantica: sobrepaso el limite de constantes declaradas en la linea %r" % (line))

# Func : assignvirtualaddress
# Params : Una tabla de variables vartab, un scope addressscope y el número del linea de los tokens linenum
# Ret : Una tabla de variables actualizada con las direcciones de cada variable
# Desc : Asigna las direcciones virtuales a la tabla de variables
def assignvirtualaddress(vartab,addressscope,linenum):
    # Rangos globales de memoria
    global globalint
    global globalfloat
    global globalchar
    global globalbool
    global globalpoint
    # Direcciones locales
    global localint
    global localfloat
    global localchar
    global localbool
    global localpoint
    # Cant. Maximas de variables
    global INTMAX
    global FLOATMAX
    global CHARMAX

```



```

global BOOLMAX
global POINTAYMAX

# Contadores globales
global glbintcount
global glbfloatcount
global glbcharcount
global glbboolcount
#
# Contadores de variables
intcount = 0
floatcount = 0
charcount = 0
boolcount = 0

# Contadores de arreglos
pointcount = 0

if addressscope == 'global':
    for k in vartab.keys():
        if('dims' in vartab[k].keys()): # Asignacion de direcciones de arreglos
            incr = vartab[k]['size']
            isArr = True
        else:
            incr = 1
            isArr = False

        if(vartab[k]['tipo'] == 'entero'): #Asignar las direcciones enteras
            if (intcount < INTMAX):
                vartab[k]['address'] = globalint + intcount
                intcount += incr
                if isArr:
                    addConst(vartab[k]['address'], 'entero', linenum)
            else:
                printerror("Error de Semantica: sobrepaso el limite de variables declaradas en la linea %r" % (linenum))

        elif (vartab[k]['tipo'] == 'flotante'): #Asignar las direcciones flotantes
            if (floatcount < FLOATMAX):
                vartab[k]['address'] = globalfloat + floatcount
                floatcount += incr
                if isArr:
                    addConst(vartab[k]['address'], 'entero', linenum)
            else:
                printerror("Error de Semantica: sobrepaso el limite de variables declaradas en la linea %r" % (linenum))

        elif (vartab[k]['tipo'] == 'char'): #Asignar las direcciones char
            if (charcount < CHARMAX):
                vartab[k]['address'] = globalchar + charcount
                charcount += incr
                if isArr:
                    addConst(vartab[k]['address'], 'entero', linenum)
            else:
                printerror("Error de Semantica: sobrepaso el limite de variables declaradas en la linea %r" % (linenum))

        elif (vartab[k]['tipo'] == 'bool'): #Asignar las direcciones bool
            if (boolcount < BOOLMAX):
                vartab[k]['address'] = globalbool + boolcount
                boolcount += incr
            else:
                printerror("Error de Semantica: sobrepaso el limite de variables declaradas en la linea %r" % (linenum))

elif addressscope == 'local':
    for k in vartab.keys():
        if('dims' in vartab[k].keys()): # Asignacion de direcciones de arreglos
            incr = vartab[k]['size']
            isArr = True
        else:
            incr = 1

```

```

isArr = False

if (vartab[k]['tipo'] == 'entero'): #Asignar las direcciones enteras
    if (intcount < INTMAX):
        vartab[k]['address'] = localint + intcount
        intcount += incr
        if isArr:
            addConst(vartab[k]['address'], 'entero', linenum)
    else:
        printerror("Error de Semantica: sobrepaso el limite de variables declaradas en la linea %r" % (linenum))

elif (vartab[k]['tipo'] == 'flotante'): #Asignar las direcciones flotantes
    if (floatcount < FLOATMAX):
        vartab[k]['address'] = localfloat + floatcount
        floatcount += incr
        if isArr:
            addConst(vartab[k]['address'], 'entero', linenum)
    else:
        printerror("Error de Semantica: sobrepaso el limite de variables declaradas en la linea %r" % (linenum))

elif (vartab[k]['tipo'] == 'char'): #Asignar las direcciones char
    if (charcount < CHARMAX):
        vartab[k]['address'] = localchar + charcount
        if isArr:
            addConst(vartab[k]['address'], 'entero', linenum)
        charcount += incr
    else:
        printerror("Error de Semantica: sobrepaso el limite de variables declaradas en la linea %r" % (linenum))

elif (vartab[k]['tipo'] == 'bool'): #Asignar las direcciones bool
    if (boolcount < BOOLMAX):
        vartab[k]['address'] = localbool + boolcount
        if isArr:
            addConst(vartab[k]['address'], 'entero', linenum)
        boolcount += incr
    else:
        printerror(
            "Error de Semantica: sobrepaso el limite de variables declaradas en la linea %r" % (linenum)
        )

return vartab, intcount, floatcount, charcount, boolcount

```

Ejemplos de Documentación en la máquina virtual:

```
# Func: storeinmem
# Param: Una direccion virtual addr, un valor val, y opcionalmente un flag indicando
# si guardar en la variable temporal era
# Desc: Guardar en memoria el valor dado.
def storeinmem(addr, val, isera = False):
    scop, atype, aoff = getTypeAndOffset(addr)
    if scop == 'global':
        if atype == 0: #entero
            globmem.mint[aoff] = val
        elif atype == 1: #float
            globmem.mfloat[aoff] = val
        elif atype == 2: #char
            globmem.mchar[aoff] = val
        elif atype == 3: #bool
            globmem.mbool[aoff] = val
        elif atype == 4: #pointer
            globmem.mpoint[aoff] = val
    elif scop == 'local':
        if atype == 0: #entero
            if isera:
                eratemp[0].mint[aoff] = val
            else:
                memstack[-1][0].mint[aoff] = val
        elif atype == 1: #float
            if isera:
                eratemp[0].mfloat[aoff] = val
            else:
                memstack[-1][0].mfloat[aoff] = val
        elif atype == 2: #char
            if isera:
                eratemp[0].mchar[aoff] = val
            else:
                memstack[-1][0].mchar[aoff] = val
        elif atype == 3: #bool
            if isera:
                eratemp[0].mbool[aoff] = val
            else:
                memstack[-1][0].mbool[aoff] = val
        elif atype == 4: #pointer
            if isera:
                eratemp[0].mpoint[aoff] = val
            else:
                memstack[-1][0].mpoint[aoff] = val
    elif scop == 'temp':
        if atype == 0: #entero
            if isera:
                eratemp[1].mint[aoff] = val
            else:
                memstack[-1][1].mint[aoff] = val
        elif atype == 1: #float
            if isera:
                eratemp[1].mfloat[aoff] = val
            else:
                memstack[-1][1].mfloat[aoff] = val
        elif atype == 2: #char
            if isera:
                eratemp[1].mchar[aoff] = val
            else:
                memstack[-1][1].mchar[aoff] = val
        elif atype == 3: #bool
            if isera:
                eratemp[1].mbool[aoff] = val
            else:
                memstack[-1][1].mbool[aoff] = val
        elif atype == 4: #pointer
            if isera:
```

```

        if eratemp[1].mpoint[aoff] == None:
            eratemp[1].mpoint[aoff] = val
        else:
            storeinmem(eratemp[1].mpoint[aoff], val, isera)
    else:
        if memstack[-1][1].mpoint[aoff] == None:
            memstack[-1][1].mpoint[aoff] = val
        else:
            #val, _ = getexpoper(memstack[-1][1].mpoint[aoff])
            storeinmem(memstack[-1][1].mpoint[aoff], val)

# Func: valtonum
# Params: Un valor val, y un tipo t
# ret: Regresa el valor de v como el tipo t
# Desc: Cambiar valor a numero si es necesario
def valtonum(val,t):
    if t == 2:
        val = ord(val)
    elif t == 3:
        if val:
            val = 1
        else:
            val = 0
    return val

# Func: reusePointer
# Params: Una dirección virtual addr y un valor
# Reescribir el valor del apuntador cuando es necesario
def reusePointer(addr, val):
    scop, atype, aoff = getTypeAndOffset(addr)
    memstack[-1][1].mpoint[aoff] = val

# Func : exeExpresion
# Params : Un operador op, tres direcciones virtuales ladd, radd resadd
# Desc : Ejecuta la acción del cuádruplo y la guarda en memoria
def exeExpresion(op,ladd,radd,resadd):
    lop,lt = getexpoper(ladd)
    rop,rt = getexpoper(radd)
    if lop == None or rop == None:
        printerr(
            'Esta intentando realizar operaciones con variables que no cuentan con un valor.'
            +
            '\nRevisa el código para asegurar que no haya alguna variable sin valor en alguna de tus operaciones'
            ,ip)
    lop = valtonum(lop,lt)
    rop = valtonum(rop,rt)
    aux = 0
    # Operadores ['*', '/', '+', '-', '>', '<', '>=', '<=', '!=', '==', '||', '&&']
    if op == '*':
        # Multiplicacion
        aux = lop * rop

    elif op == '/':
        # Division
        aux = lop / rop

    elif op == '+':
        # Suma
        if rop == '':
            aux = abs(lop)

```

```

        else:
            aux = lop + rop

    elif op == '-':
        #Resta
        if rop == '':
            aux = -1 * abs(lop)
        else:
            aux = lop - rop

    elif op == '>':
        #Mayorque
        aux = lop > rop

    elif op == '<':
        #menor que
        aux = lop < rop

    elif op == '>=':
        # mayor igual
        aux = lop >= rop

    elif op == '<=':
        #menor igual
        aux = lop <= rop

    elif op == '==':
        #igual (relacional)
        aux = lop == rop

    elif op == '!=':
        #diferente
        aux = lop != rop

    elif op == '&&':
        # and
        aux = lop and rop

    elif op == '||':
        # or
        aux = lop or rop
    if type(resadd) is str: # Reescribir el pointer en caso de ser necesaria
        reusePointer(int(resadd[1:]),aux)
    else:
        storeinmem(resadd,aux)

```