Diseño de Compiladores

Documentación

Covid-19--

Fecha:
3 de junio de 2020

Maestros:
Héctor Gibrán Ceballos Cancino
Elda Guadalupe Quiroga González

Autores:

Emilio Fernando Alonso Villa
[A00959385]

Juan Antonio Lizárraga Vizcarra
[A01282540]

Índice

Descripción del Proyecto

Propósito, Objetivos y Alcance

Al inicio del semestre se presentó el proyecto de crear un compilador de un *little language*, es decir un lenguaje de programación que tuviera los requerimientos mínimos para poder ser utilizado como un lenguaje normal. En primera instancia el equipo se dió a la tarea de presentar su propuesta de lenguaje, no obstante debido a la situación global con relación a la pandemia del COVID-19 y su subsecuente aislamiento social dicho objetivo fue sustituido, dando lugar a que el equipo decidiera llevar a cabo la propuesta presentada por la Ing. Elda Quiroga y el Dr. Héctor Ceballos cuyo nombre hace referencia al mismo virus, Covid-19--.

Los objetivos y alcance de la propuesta eran claros. Había que desarrollar un lenguaje imperativo, procedural, con fuerte tipado. Dicho lenguaje debería cumplir con el uso de:

- Operaciones aritméticas
- Operaciones lógicas
- Operaciones relacionales
- Uso de condicionales
- Uso de ciclos (controlados y no controlados)
- Arreglos de una y dos dimensiones
- Uso de *dataframes* para operaciones estadísticas

Con estos requerimientos en mente, a continuación se presentarán los resultados del desarrollo de dicho lenguaje.

Análisis de Requerimientos

Tabla de requerimientos

RF01	El lenguaje deberá poder resolver expresiones que incluyan operadores aritméticos, relativos, de comparación y lógicos.
RF02	El lenguaje deberá contar con estatutos de interacción para entrada y salida de datos.
RF03	El lenguaje deberá soportar estatutos de ciclos. Primero, un ciclo con límite inferior y superior definido (for loop). Segundo, un ciclo condicional (while).
RF04	El lenguaje deberá soportar estatutos de control. Esto incluye los estatutos condicionales if y else.
RF05	El lenguaje deberá soportar código modular por medio de funciones. Estas podrán regresar algún dato primitivo o no (ser de tipo void). Las funciones podrán recibir o no un conjunto de parámetros.

RF06	El lenguaje deberá soportar arreglos con los tipos de datos primitivos.
RF07	El lenguaje deberá poder operar con dataframes, formado a partir de comma-separated values (archivos .csv). Se deberán poder realizar operaciones estadísticas con estos dataframes (sacar media, rango, desviación estándar, etc.)

Casos de uso

Caso de uso: Realizar suma de operandos	**ID:** UC001
Descripción: El código obtiene la suma de dos operandos	
Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '+' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual obtiene la suma de los dos operandos
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● Se regresa el valor de la suma de los operandos
Flujo alternativo: ● Se arroja error si los tipos de datos no son compatibles	

Caso de uso: Realizar resta de operandos	**ID:** UC002
Descripción: El código obtiene la resta de dos operandos	
Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '–' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual obtiene la resta de los dos operandos
Condición de entrada:	● Operandos son compatibles

Condición de salida:	● Se regresa el valor de la resta de los operandos
Flujo alternativo: ● Se arroja error si los tipos de datos no son compatibles	

Caso de uso: Realizar producto de operandos	**ID:** UC003
Descripción: El código obtiene el producto de dos operandos	
Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '*' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual obtiene el producto de los dos operandos
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● Se regresa el valor del producto de los operandos
Flujo alternativo: ● Se arroja error si los tipos de datos no son compatibles	

Caso de uso: Realizar división de operandos	**ID:** UC004
Descripción: El código obtiene la división de dos operandos	
Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '/' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual obtiene el producto de los dos operandos
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● Se regresa el valor de la división de los operandos

Flujo alternativo: ● Se arroja error si los tipos de datos no son compatibles ● Si el denominador es cero, también se arroja error

Caso de uso: Comparar igualdad de operandos	**ID:** UC005
Descripción: El código determina si dos operandos son iguales en valor	
Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '==' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual determina si los operandos son equivalentes
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● Se regresa si los operandos son equivalentes
Flujo alternativo: ● Se arroja error si los tipos de datos no son compatibles	

Caso de uso: Comparar desigualdad de operandos	**ID:** UC006
Descripción: El código determina si dos operandos no son iguales en valor	
Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '!=' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual determina si los operandos no son equivalentes
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● Se regresa si los operandos no son equivalentes
Flujo alternativo:	

● Se arroja error si los tipos de datos no son compatibles

Caso de uso: Realizar NOT lógico	**ID:** UC007
Descripción: El código obtiene el opuesto lógico de un operando entero	
Flujo de eventos:	1. Programador escribe '!' 2. Programador escribe un operando (variable, constante) 3. Ejecutar código 4. Máquina Virtual obtiene el opuesto lógico
Condición de entrada:	● El operando es de tipo entero
Condición de salida:	● Se obtiene opuesto lógico del operando
Flujo alternativo: ● Se arroja error si el operando no es de tipo entero	

Caso de uso: Realizar menor o igual que	**ID:** UC008
Descripción: El código determina si un operando es menor o igual a otro	
Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '<=' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual determina si primer operando es menor o igual que el segundo operando
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● Se obtiene determina si primer operando es menor o igual que el segundo operando
Flujo alternativo: ● Se arroja error si los tipos de datos no son compatibles	

Caso de uso: Realizar mayor o igual que	**ID:** UC009
Descripción: El código determina si un operando es mayor o igual a otro	
Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '>=' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual determina si primer operando es mayor o igual que el segundo operando
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● Se obtiene determina si primer operando es mayor o igual que el segundo operando
Flujo alternativo: ● Se arroja error si los tipos de datos no son compatibles	

Caso de uso: Realizar menor que	**ID:** UC010
Descripción: El código determina si un operando es menor que otro	
Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '<' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual determina si primer operando menor que el segundo operando
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● Se obtiene determina si primer operando menor que el segundo operando
Flujo alternativo: ● Se arroja error si los tipos de datos no son compatibles	

Caso de uso: Realizar mayor que	**ID:** UC011

Descripción: El código determina si un operando es mayor que otro	
Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '>' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual determina si primer operando mayor que el segundo operando
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● Se obtiene determina si primer operando mayor que el segundo operando
Flujo alternativo: ● Se arroja error si los tipos de datos no son compatibles	

Caso de uso: Realizar OR lógico	**ID:** UC012
Descripción: El código realiza un OR lógico entre dos operandos	
Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '||' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual realiza OR lógico entre los operandos
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● Se obtiene el resultado del OR lógico
Flujo alternativo: ● Se arroja error si los tipos de datos no son enteros	

Caso de uso: Realizar AND lógico	**ID:** UC013
Descripción: El código realiza un AND lógico entre dos operandos	

Flujo de eventos:	1. Programador escribe un operando (variable, constante) 2. Programador escribe '&&' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual realiza AND lógico entre los operandos
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● Se obtiene el resultado del AND lógico
Flujo alternativo: ● Se arroja error si los tipos de datos no son enteros	

Caso de uso: Realizar asignación	**ID:** UC014
Descripción: El código realiza una asignación a una variable	
Flujo de eventos:	1. Programador escribe una variable 2. Programador escribe '=' 3. Programador escribe una expresión 4. Ejecutar código 5. Máquina Virtual asigna el valor de la expresión a la variable
Condición de entrada:	● Operandos son compatibles
Condición de salida:	● El valor de la variable es el resultado de la expresión
Flujo alternativo: ● Se arroja error si los tipos de datos no son compatibles	

Caso de uso: Realizar condicional	**ID:** UC015
Descripción: El código realiza control de flujo mediante una condicional	
Flujo de eventos:	1. Programador escribe 'if' 2. Programador escribe una expresión condicional entre paréntesis 3. Programador escribe un bloque de código

	4. Ejecutar código 5. Máquina Virtual evalúa expresión 6. Máquina Virtual decide si se ejecuta bloque de código
Condición de entrada:	● Expresión dentro de condicional es de tipo entera
Condición de salida:	● Máquina virtual ejecuta los cuádruplos según la evaluación de la expresión en la condicional.
Flujo alternativo: ● Se agrega un bloque else, que contiene código a ejecutar en caso de que la expresión no evalúa a verdadero (diferente de cero) ● Se arroja error cuando expresión no es de tipo entero	

Caso de uso: Realizar ciclo delimitado	**ID:** UC016
Descripción: El código repite código dentro del ciclo de acuerdo a límites	
Flujo de eventos:	1. Programador escribe ‘for’ 2. Programador inicializa una variable previamente definida 3. Programador escribe ‘to’ 4. Programador define límite superior como expresión 5. Programador escribe un bloque de código 6. Programador ejecuta código 7. Máquina virtual inicializa la variable 8. Máquina virtual evalúa condición de límite superior 9. En éxito, máquina virtual ejecuta código 10. Máquina virtual incrementa iterador 11. Máquina virtual regresa al paso 8
Condición de entrada:	● Iterador debe ser numérico ● Expresión límite debe ser numérica
Condición de salida:	● Máquina virtual ejecuta los cuádruplos según la evaluación de la expresión en la condicional.
Flujo alternativo: ● Se arroja error cuando no se cumplen las condiciones de tipo de dato	

Caso de uso: Realizar ciclo condicional	**ID:** UC017

Descripción: El código repite código dentro del ciclo de acuerdo a la condición	
Flujo de eventos:	1. Programador escribe ‘while’ 2. Programador escribe una expresión entre paréntesis 3. Programador escribe un bloque de código 4. Programador ejecuta código 5. Máquina virtual evalúa condición 6. En éxito, máquina virtual ejecuta código 7. Máquina virtual regresa al paso 5
Condición de entrada:	● Expresión en la condición debe ser tipo entera
Condición de salida:	● Máquina virtual ejecuta los cuádruplos según la evaluación de la expresión en la condicional.
Flujo alternativo: ● Se arroja error cuando no se cumplen las condiciones de tipo de dato	

Caso de uso: Declaración de una función	**ID:** UC018
Descripción: El código genera los cuádruplos para una subrutina del código	
Flujo de eventos:	1. Programador escribe ‘func’ 2. Programador escribe tipo de retorno o void 3. Programador escribe nombre de la función 4. Programador escribe parámetros entre paréntesis 5. Programador escribe bloque de código 6. Programador ejecuta código 7. Se generan los cuádruplos necesarios
Condición de entrada:	● Nombre de la función es único
Condición de salida:	● Compilador genera los cuádruplos para la función
Flujo alternativo: ● Se arroja error cuando el nombre ya existe (función redefinida)	

Caso de uso: Llamada de una función	**ID:** UC019

Descripción: El código hace una llamada a una subrutina	
Flujo de eventos:	1. Programador escribe el nombre de la subrutina 2. Programador escribe los argumentos a enviar entre paréntesis 3. Programador ejecuta código 4. Máquina virtual ejecuta método
Condición de entrada:	● Argumentos coinciden en número y tipo con parámetros
Condición de salida:	● Máquina virtual ejecuta la función
Flujo alternativo: ● Se arroja error cuando existe algún problema con los argumentos	

Caso de uso: Realizar escritura	**ID:** UC020
Descripción: El código imprime los contenidos del llamado	
Flujo de eventos:	1. Programador escribe ‘print’ 2. Programador escribe expresiones o letreros 3. Programador ejecuta el código 4. Máquina virtual imprime los contenidos del llamado
Condición de entrada:	● N/A
Condición de salida:	● Máquina virtual escribe los contenidos del llamado
Flujo alternativo: ● Máquina virtual manda error si el valor no ha sido inicializado	

Caso de uso: Realizar lectura	**ID:** UC021
Descripción: El código pide una entrada al usuario y la escribe en alguna variable	
Flujo de eventos:	1. Programador escribe ‘input’ 2. Programador escribe variables

	3. Programador ejecuta el código 4. Máquina virtual pide entrada a usuario 5. Programador escribe algún valor 6. Máquina virtual asigna ese valor a la variable
Condición de entrada:	● Tipos de datos son compatibles
Condición de salida:	● Entrada de usuario se le asigna a una variable
Flujo alternativo: ● Si los tipos de datos no son compatibles arroja error	

Caso de uso: Operaciones COVID	**ID:** UC022
Descripción: El código hace operaciones en un dataframe	
Flujo de eventos:	1. Programador abre un archivo con load_file() 2. Programador carga los datos en un dataframe con load_data() 3. Programador usa algún método COVID 4. Programador ejecuta código 5. Máquina virtual carga archivo y datos 6. Máquina virtual ejecuta método COVID
Condición de entrada:	● Archivo existe y se puede abrir ● Parámetros para llamados coinciden con su definición
Condición de salida:	● Máquina virtual realiza operaciones con el dataframe
Flujo alternativo: ● Si el archivo no existe el lenguaje arroja un error ● Si no se mandan los parámetros de manera correcta, se arroja error	

Test Cases

Nombre: Expresiones lineales	**ID:** CP01
Objetivo: Realizar operaciones aritméticas, lógicas y relacionales.	

Entradas:
`program Debugging;`
 	
`var`
`    int a, b;`
`    float x;`
 	
`main();`
`var int n;`
`{`
`    x = 1;`
`    n = 2;`
`    b = 3;`
`    a = n * x + 22  / x * 12;`
`    b = n && !b || 0;`
`    n = b == 3 || a + 2 == 268;`
`    print(a, "\n");`
`    print(b, "\n");`
`    print(n, "\n");`
`}`
	Salida:
	`Successful compilation!`
	`266`
	`0`
	`1`

Resultado de prueba: Exitosa	

Nombre: Expresiones no lineales	**ID:** CP02
Objetivo: Realizar condicionales y ciclos	

Entradas:
`program NonLinear;`
 	
`var`
`       int A, C, D;`
 	
`main();`
`var`
`    int i;`
`{`
`    D = 10;`
`    A = 1;`
`    while(A < D) {`
`        print(A, " ");`
`        A = A + 1;`
`    }`
	Salida:
	`Successful compilation!`
	`1 2 3 4 5 6 7 8 9`
	`0 1 2 3 4 5 6 7 8 9`
	`Under 100`

```
    print("\n");
    for i = 0 to A{
        print(i, " ");
    }
    if(i <= A){
        C = 100;
    }
    if(C <= 0){
        print("\nOver 100");
    }
    else {
        print("\nUnder 100");
    }
}
```

Resultado de prueba:
Exitosa

Nombre: Modulos	**ID:** CP03

Objetivo:
Utilización de código modular

Entradas:

```
program Modules;

func void printer(float text);
{
    print(text, "\n");
}

func float func1(float bar);
var float j;
{
    j = bar * 100;
    printer(j);

    return(j+1);
}


main();
var float foo, baz;
{
    foo = 100;

    baz = func1(foo);

    printer(baz);
}
```

Salida:

```
Successful compilation!
10000.0
10001.0
```

Resultado de prueba:
Exitosa

Nombre: Dataframes	**ID:** CP04

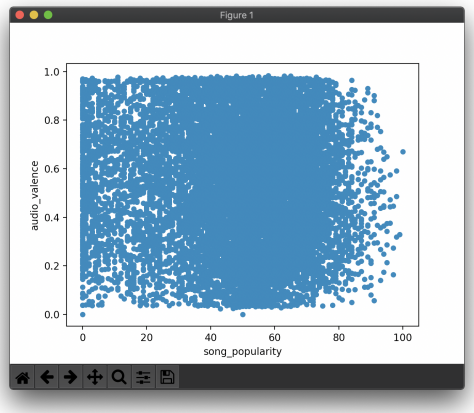
Objetivo:
Utilización de *dataframes* para cálculos estadísticos

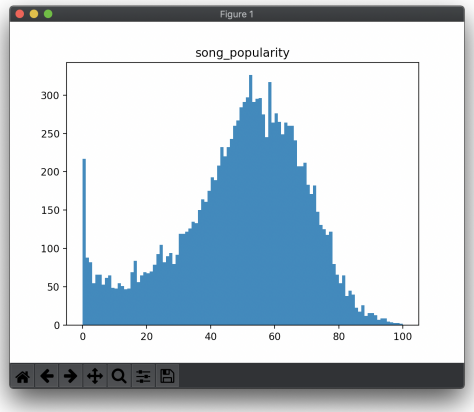
Entradas:

```
program Dataframe;
var dataframe data;

main();
var int rows, cols;
    float res;
{
    load_file("song_data_clean.csv");
    load_data(data, rows, cols);

    print("Number of rows: ", rows, "\n");
    print("Number of cols: ", cols, "\n");

    print("\nAverage: ", avg(data,
"song_popularity"), "\n");
    print("Mode: ", mode(data,
"song_popularity"), "\n");
    print("Range: ", range(data,
"song_popularity"), "\n");
    print("Variance: ", variance(data,
"song_popularity"), "\n");
    print("Std_Dev: ", std_dev(data,
"song_popularity"), "\n");
    print("Max: ", max(data, "song_popularity"),
"\n");
    print("Min: ", min(data, "song_popularity"),
"\n");
    print("Correl: ", correl(data,
"song_popularity", "audio_valence"), "\n");

    plot(data, "song_popularity",
"audio_valence");
    histogram(data, "song_popularity", 100);
}
```

Salida:

```
Successful compilation!
Number of rows: 13053
Number of cols: 15

Average: 48.48448632498276
Mode: 52.0
Range: 100.0
Variance: 404.52038832981987
Std_Dev: 20.11269221983521
Max: 100.0
Min: 0.0
Correl: -0.04976362384324907
```





Resultado de prueba:
Exitosa

Proceso de Desarrollo

Proceso

Para el desarrollo del compilador en cuestión a lo largo del documento el equipo decidió hacer uso de la técnica conocida como *pair programing*. Tomando en cuenta los beneficios esperados de dicha técnica, entre los cuales se incluye mayor calidad en el código, mejor difusión de conocimiento en el equipo, etc., se optó por esta. Dicho lo anterior, para llevar a cabo esta técnica el equipo hizo uso de llamadas con protocolo VoIP a través de la aplicación *Discord* para su comunicación. Paralelamente el editor de texto utilizado fue *Visual Studio Code* de la compañía *Microsoft*, este editor cuenta con un aditamento llamado *Live Share* cuya funcionalidad permite al equipo compartir un código en tiempo real y editarlo de manera paralela.

El uso de estas herramientas fue crucial para el proyecto, ya que las mismas permitieron al equipo llevar a cabo discusiones sobre el diseño del código de manera más eficientemente. A su vez la posterior implementación de dicho diseño también se vió afectada de manera positiva ya que al haber dos individuos trabajando sobre el mismo código con la misma idea, más código era escrito en menor tiempo.

Bitácoras

13/04/2020	**Avance:** Léxico y Sintaxis **Progreso:** Se genera analizador léxico y sintáctico, usando ANTLR sobre Python. Compilador marca cuando el código se pudo compilar de manera exitosa y cuando hay error de sintaxis marca el token recibido y el esperado. **Commits:** [Link] y [Link]
20/04/2020	**Avance:** DirFunciones y Tablas de Variables **Progreso:** Se creó una clase DirFunc para administrar las funciones del programa, sus tipos y sus variables. Dentro de cada función se definen las variables de ese alcance. **Commits:** [Link] y [Link]
27/04/2020	**Avance:** Cubo Semántico y Cuádruplos para Expresiones **Progreso:** Se creó el cubo semántico para definir compatibilidad de tipos al hacer operaciones. Se implementó una memoria dummy que arroja las direcciones pero no guarda valores. Se implementaron las tuplas para expresiones básicas (aritméticas, relacionales, lineales) y clase que las genera. Se preserva precedencia y asociatividad. Compatible con paréntesis. **Commits:** [Link], [Link], [Link], [Link] y [Link]

04/05/2020	**Avance:** Cuádruplos para Estatutos No-Lineales **Progreso:** Se generan instrucciones para if/else. Además se generan los cuádruplos para las estructuras de ciclo for y while. Funcionamiento de estas estructuras de forma anidada también fue comprobada y fue exitoso. **Commits:** [Link] y [Link]
11/05/2020	**Avance:** Funciones **Progreso:** DirFunc ahora obtiene información sobre espacio requerido por función para ERA. Se generan los cuádruplos necesarios para ejecutar subrutinas (ERA, PARAM, GOSUB, ENDFUNC). Se genera además cuádruplo para retornar valores. Asimismo funciones se dan de alta en tabla global para almacenar valores de retorno. **Commits:** [Link] y [Link]
18/05/2020	**Avance:** Máquina Virtual - parte 1 **Progreso:** Se refactoriza la memoria para que pueda almacenar valores y además se fragmentó por tipo de dato. Se creó clase Máquina Virtual para la ejecución de los cuádruplos. Por ahora, Máquina Virtual puede ejecutar estatutos lineales. **Commits:** [Link] y [Link]
25/05/2020	**Avance:** Traducción de Arreglos **Progreso:** Máquina Virtual puede procesar cuádruplos no lineales (ciclos, condicionales y llamado de funciones). Además, ahora el compilador soporta declaración y acceso a arreglos. **Commits:** [Link] y [Link]
31/05/2020	**Avance:** Último avance **Progreso:** Se agregaron funciones especializadas COVID, que incluyen operaciones estadísticas sobre un dataframe, generado de un archivo csv. Además, se refinan los mensajes de error y corrigen los últimos detalles del lenguaje. **Commits:** [Link] y [Link]

Reflexión

Juan Antonio Lizárraga Vizcarra

El proyecto es definitivamente uno de los más complejos de toda la carrera. Sin embargo, es muy enriquecedor al proveer una perspectiva sobre el funcionamiento de los componentes internos de un lenguaje de programación. Manejando todo desde un nivel alto del stack, uno no se preocupa sobre las implementaciones internas de la memoria y las instrucciones máquina,

gracias a la abstracción. No obstante, al codificar esta parte de la herramienta, adquirí la razón de muchos errores o de decisiones de diseño que frecuentan muchos lenguajes de programación. Esto ha resultado en una mayor apreciación de los lenguajes, así como una comprensión de instancias en las que un compilador o una máquina virtual arroja un error de ejecución. Finalmente, no sería una subestimación decir que el proyecto requirió de muchas horas de trabajo (que incluían diseño, codificación y pruebas), mas creo que al final de cuentas, valió mucho la pena por los conocimientos adquiridos y ver un lenguaje de programación funcionando.

Emilio Fernando Alonso Villa.

El proyecto en general me enseñó muchas cosas. Me atrevo a decir que en primera instancia subestimé la dificultad del proyecto, sobre todo la manera en la que se implementan arreglos, honestamente pensé que se iba a poder hacer en dos semanas. No obstante a lo largo del proyecto me dí cuenta de la dificultad real que hay detrás de un compilador y que las cosas que tomamos por dado en un lenguaje no son tan sencillas como las creemos. Una vez dicho esto, a lo largo del curso de matemáticas computacionales constantemente cuestionaba el uso de cosas como las gramáticas y este proyecto definitivamente me sirvió como un ejemplo claro y cristalino sobre el uso de las mismas. A su vez todos los pasos necesarios para generar una representación como un código de tres direcciones y su posterior uso para la representación interna en la máquina virtual.

Descripción del lenguaje

Nombre del lenguaje

El nombre del lenguaje descrito a lo largo del presente texto es Covid-19--

Descripción de las principales características del lenguaje

En primera instancia el lenguaje permite al programador llevar a cabo operaciones aritméticas básicas, las cuales incluyen: suma, resta, multiplicación y división. Además las operaciones lógicas y relacionales también se encuentran presentes en el lenguaje descrito, entre ellas: and, or y not, éstas funcionan con lógica aritmética al igual que C. Cabe hacer mención que el operador not(!) sólo funciona con variables de tipo entero. Del lado relacional se encuentran operaciones de: <, <=, >, >=, ==, !=, de nuevo, trabajan con lógica aritmética.

En cuanto a los tipos de datos manejados para el lenguaje, éste cuenta con tipos: *integer (int), float, string, char* y *dataframe*. Los tipos de dato *int* y *float* pueden interactuar entre sí. Si un float es asignado a un int, simplemente se llevará a cabo una truncación de decimales y se guardará la parte entera en la variable que recibe estos datos.

El lenguaje cuenta con algunas bondades para el programador, algunas de estas son:

- Implementación de ciclos en forma de *while* y *for* loops
- Implementación de condicionales en bloques *if, if-else*
- Implementación de módulos, estos últimos necesitan ser definidos por el usuario con base en la guía que se proporcionará más adelante en este texto.
- Uso de *dataframes* para cálculos estadísticos, estos permiten calcular descriptivos de un grupo de datos los cuales son alimentados al programa en forma de un archivo .csv.

Listado de errores

Ejecución:

Mensaje	**Descripción**
[Error] Division by zero	Este error se suscita cuando el usuario intenta hacer una división con divisor igual a 0.
[Error]: Argument type does not match parameter type	Este error se suscita cuando el usuario intenta pasar un parámetro de tipo diferente del definido en un módulo

[Error] Index out of range for array	Este error se suscita cuando el índice pasado a un arreglo es mayor al tamaño del mismo o negativo.
[Error] File should be .csv format	Este error se suscita cuando el tipo de archivo utilizado para un dataframe es de tipo diferente a .csv.
[Error] File could not be opened	Este error se suscita cuando el archivo utilizado para un dataframe no fue encontrado en el sistema o no se pudo abrir.
[Error]: Dataframe key not found in file	Este error se suscita en cualquier operación sobre dataframe cuando la llave, provista por el programador, no existe en el data frame.

Compilación:

Mensaje	**Descripción**
[Error] Memory stack exceeded for type and context	Este error se suscita cuando el programador crea más de 100 variables de un tipo ya sea en scope local, temporal o global o constante.
[Error] Uninitialized variable	Este error se suscita cuando el programador hace uso de una variable sin valor asignado en una operación distinta a asignación.
[Error: line#] Redefinition of function func_name	Este error se suscita cuando el programador nombra a dos funciones de la misma manera, redefiniendo así una función previamente declarada.
[Error: line#] Redefinition of variable var_name	Este error se suscita cuando el programador nombra a dos variables de la misma manera, redefiniendo así una variable previamente declarada.
[Error: line#] Only one dataframe allowed per program	Este error se suscita cuando el programador intenta declarar más de una variable de tipo *dataframe* en el programa.
[Error: line#] Non void function does not have a return statement	Este error se suscita cuando el programador no hace uso de un retorno en una función de tipo diferente a *void.*
[Error: line#] Number of arguments does not coincide with number of parameters in func_name	Este error se suscita cuando el programador pasa una cantidad diferente de parámetros a una función de los definidos en la función originalmente.
[Error: line#] Use of	Este error se suscita cuando el programador intenta utilizar una

undeclared variable var_name	variable cuya declaración no fue hecha previamente.
[Error: line#] Array index is not an int	Este error se suscita cuando el programador intenta inicializar el tamaño de un array con un tipo de dato diferente a un int.
[Error: line#] Type mismatch	Este error se suscita cuando el programador intenta realizar una operación no soportada por el cubo semántico entre distintos tipos.
[Error: line#] NOT operand must be of an INT	Este error se suscita cuando el programador hace uso del operador not (!) con un tipo de dato diferente a un int.
[Error: line#] Function func_name not defined	Este error se suscita cuando el programador intenta hacer uso de una función que no fue declarada previamente.
[Error: line#] Void type functions must not return	Este error se suscita cuando el programador intenta hacer uso de un retorno en una función de tipo void.
[Error: line#] Return expression type does not match function type. Must be func_type	Este error se suscita cuando el tipo de dato a retornar en una función no es del mismo tipo que el tipo de la función.
[Error: line#] Void function does not return value for expression	Este error se suscita cuando el programador intenta obtener un valor de una función de tipo void.
[Error: line#] Expected int type for conditional expression	Este error se suscita cuando el programador usa una variable de tipo diferente a int en una condicional.
[Error: line#] Parameter type mismatch. Expected param_type, got arg_type	Este error se suscita cuando el programador pasa una cantidad diferente de parámetros a una función de los definidos en la función originalmente.
[Error: line#] Argument must be of type string, got arg_type	Este error se suscita cuando el programador utiliza un argumento de tipo diferente a string para cargar datos a un *dataframe*
[Error: line#] load_data() first argument must be of type dataframe, got arg_type	Este error se suscita cuando el programador utiliza un argumento en posición 1 de tipo diferente a *dataframe* al usar load_data
[Error: line#] load_data()	Este error se suscita cuando el programador utiliza un argumento

second argument must be of type int, got arg_type	en posición 2 de tipo diferente a int al usar load_data
[Error: line#] load_data() second argument must be of type int, got arg_type	Este error se suscita cuando el programador utiliza un argumento en posición 3 de tipo diferente a int al usar load_data
[Error: line#] coivd_func() first argument must be of type dataframe, got arg_type	Este error se suscita cuando el programador utiliza un argumento en posición 1 de tipo diferente a *dataframe* en las funciones: average, mode, range, variance, std_dev, max, min, correl, plot
[Error: line#] coivd_func() second argument must be of type string, got arg_type	Este error se suscita cuando el programador utiliza un argumento en posición 1 de tipo diferente a *string* en las funciones: average, mode, range, variance, std_dev, max, min, correl, plot
[Error: line#] coivd_func() third argument must be of type string, got arg_type	Este error se suscita cuando el programador utiliza un argumento en posición 1 de tipo diferente a *string* en las funciones: correl, plot
[Error: line#] histogram() first argument must be of type dataframe, got arg_type	Este error se suscita cuando el programador utiliza un argumento en posición 1 de tipo diferente a *dataframe* en función histogram()
[Error: line#] histogram() second argument must be of type string, got arg_type	Este error se suscita cuando el programador utiliza un argumento en posición 2 de tipo diferente a *string* en función histogram()
[Error: line#] histogram() third argument must be of type int, got arg_type	Este error se suscita cuando el programador utiliza un argumento en posición 3 de tipo diferente a *int* en función histogram()

Descripción del compilador

Equipo de cómputo, lenguaje, utilerías

	Librerías utilizadas	Lenguaje
DirFunc	antlr4 os	Python 3.7
Quadruples	antlr4 os	Python 3.7
Utilities	enum	Python 3.7
SemanticCube	enum	Python 3.7

Equipo de Computo		
Component	**Equipo 1**	**Equipo 2**
OS	MacOS 10.15.4 Catalina	MacOS 10.15.4 Catalina
CPU	2.3 GHz Quad-Core Intel Core i5	2.6 GHz 6-Core Intel Core i7
Memoria	8 GB 2133 MHz LPDDR3	16 GB 2667 MHz DDR4
Gráficos	Intel Iris Plus Graphics 655 1536 MB	AMD Radeon Pro 5300M 4GB

Análisis de Léxico

Expresiones Regulares

INT_CTE	-?[0-9]+
FLOAT_CTE	-?[0-9]+\.[0-9]+
STRING_CTE	\"[^"]*\"
CHAR_CTE	\'.\'
ID	[a-zA-Z][a-zA-Z0-9_]*

Tokens

PLUS	+	**LTE**	<=	**CHAR**	char	**MAX**	max
MINUS	-	**GTE**	>=	**STRING**	string	**MIN**	min
MULT	*	**LT**	<	**DATAFRAME**	dataframe	**PLOT**	plot
DIVIDE	/	**GT**	>	**VOID**	void	**HISTOGRAM**	histogram
SEMI	;	**OR**	||	**FUNC**	func	**CORREL**	correl
COLON	:	**AND**	&&	**PRINT**	print	**WS**	[\t\n]+
COMMA	,	**ASGN**	=	**MAIN**	main	**INT_CTE**	regex
CURLY_L	{	**PROGRAM**	program	**RETURN**	return	**FLOAT_CTE**	regex
CURLY_R	}	**VAR**	var	**INPUT**	input	**STRING_CTE**	regex
PARENS_L	(	**IF**	if	**LOAD_FILE**	load_file	**CHAR_CTE**	regex
PARENS_R	)	**WHILE**	while	**LOAD_DATA**	load_data	**ID**	regex
SQUARE_L	[	**FOR**	for	**AVG**	avg	**LINE_CMT**	'//' ~[\r\n]*
SQUARE_R	]	**TO**	to	**MODE**	mode	**BLOCK_CMT**	'/*' .*? '*/'
EQ	==	**ELSE**	else	**RANGE**	range		
NE	!=	**INT**	int	**VARIANCE**	variance		
NOT	!	**FLOAT**	float	**STD_DEV**	std_dev		

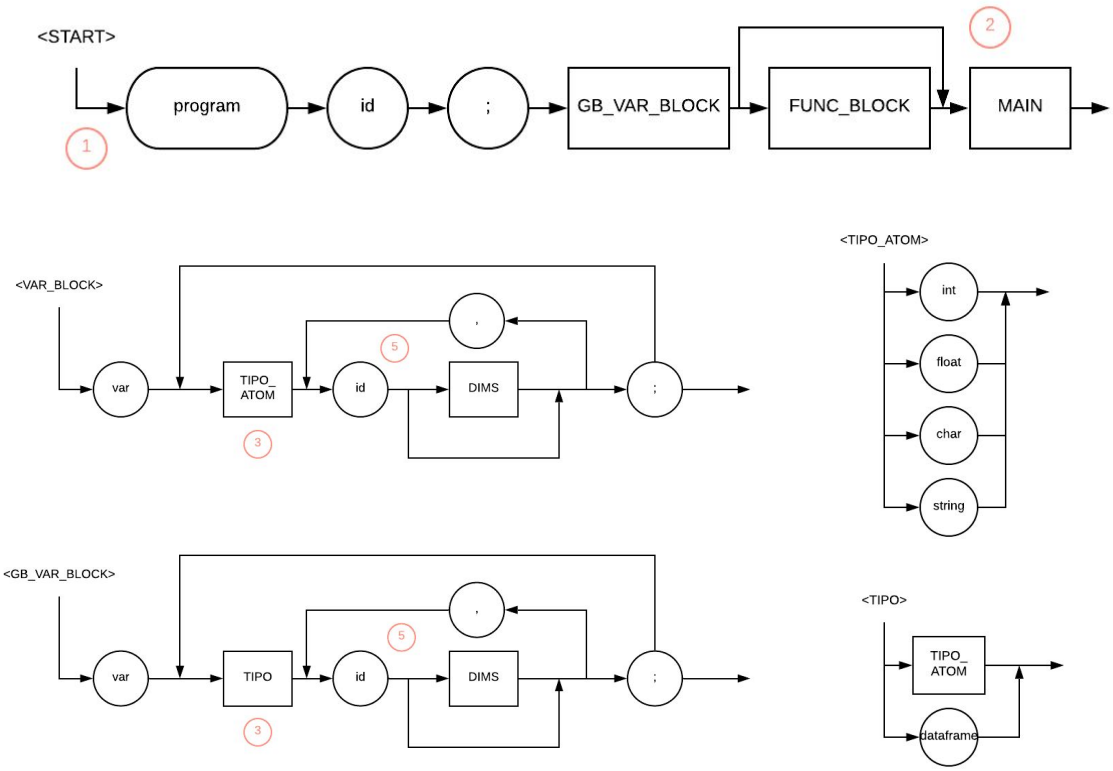
Análisis de Sintaxis

```
START           -> program <id> ; VAR_BLOCK FUNC_BLOCK MAIN

VAR_BLOCK       -> var VARS | empty
VARS            -> TIPO <id> DIMS IDS ; VARS | TIPO <id> DIMS IDS ;
TIPO            -> TIPO | dataframe
TIPO_ATOM       -> int | float | char | string
DIMS            -> [ <cte-int> ] | [ <cte-int> ] [ <cte-int> ] |
empty
IDS             -> , <id> DIMS IDS | empty

FUNC_BLOCK      -> FUNCION FUNC_BLOCK | empty
FUNCION         -> func TIPO_RETORNO <id> ( PARAM_LIST ) ; VAR_BLOCK
{
                   ESTATUTOS }
```

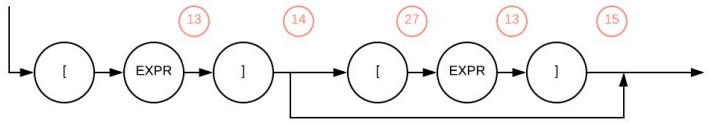
```
TIPO_RETORNO    -> TIPO | void
PARAM_LIST      -> empty | PARAMS
PARAMS          -> TIPO <id> , PARAMS | TIPO <id>

MAIN            -> main ( ) ; { ESTATUTOS }

ESTATUTOS       -> ESTATUTO ESTATUTOS | empty
ESTATUTO        -> ASIGNACION | LLAMADA ; | RETORNO | LECTURA |
                   ESCRITURA | DECISION | MIENTRAS | DESDE |
                EXPRESION
                   | COVID ;

ASIGNACION      -> <id> DIMS = EXPRESION ;

LLAMADA         -> <id> ( ARG_LIST ) ;
ARG_LIST        -> empty | ARGS
ARGS            -> EXPRESION , ARGS | EXPRESION

RETORNO         -> return ( EXPRESION ) ;

LECTURA         -> input ( <id> DIMS IDS ) ;
ESCRITURA       -> print ( IMPRS ) ;
IMPRS           -> EXPRESION , IMPRS | EXPRESION | <cte-string>

DECISION        -> if ( EXPRESION ) { ESTATUTOS } ELSE
ELSE            -> else { ESTATUTOS } | empty

WHILE           -> while ( EXPRESION ) { ESTATUTOS }
DESDE           -> from <id> = EXPRESION to EXPRESION { ESTATUTOS }

EXPRESION       -> OR_TERM OR_TERMS

OR_TERMS        -> || OR_TERM OR_TERMS | empty
OR_TERM         -> AND_TERM AND_TERMS
AND_TERMS       -> && AND_TERM AND_TERMS | empty
AND_TERM        -> EXP > EXP | EXP < EXP | EXP >= EXP | EXP <= EXP |
                   EXP == EXP | EXP != EXP | EXP

EXP             -> TERMINO EXP'
EXP'            -> + TERMINO EXP' | - TERMINO EXP' | empty
TERMINO         -> FACTOR TERMINO'
TERMINO'        -> * FACTOR TERMINO' | / FACTOR TERMINO' | empty
FACTOR          -> ( EXPRESION ) | CTE | + CTE | - CTE | LLAMADA |
```

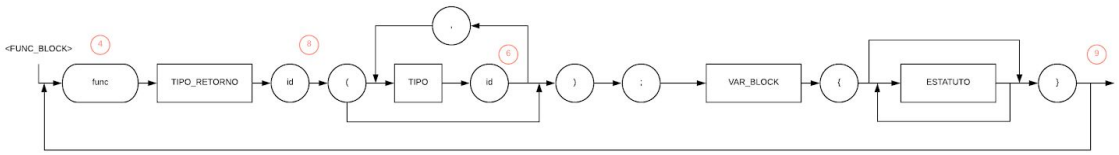
```
                    <id> DIMS

CTE                 -> <cte-int> | <cte-float> | <cte-char> |
<cte-string>

COVID               -> CARGA_ARCH | CARGA_DATOS | MEDIA | MODA | RANGO |
                       VARIANZA | STD_DEV | MAX | MIN | MOMENTO | GRAFICA
                       | HISTOGRAMA | CORRELACIONA

CARGA_ARCH          -> load_file (OPERAND) ; | load_file (<cte_string>) ;
CARGA_DATOS         -> load_data ( <id> , <id>, <id> ) ;
MEDIA               -> avg ( <id>, OPERAND ) ;
MODA                -> mode ( <id>, OPERAND ) ;
RANGO               -> range ( <id>, OPERAND ) ;
VARIANZA            -> variance ( <id>, OPERAND ) ;
STD_DEV             -> std_dev ( <id>, OPERAND ) ;
MAX                 -> max ( <id>, OPERAND ) ;
MIN                 -> min ( <id>, OPERAND ) ;
GRAFICA             -> plot ( <id>, OPERAND , OPERAND ) ;
HISTOGRAMA          -> histogram ( <id>, OPERAND , EXPRESION ) ;
CORRELACIONA        -> correl ( <id>, OPERAND , OPERAND ) ;
```

Generación de Código Intermedio y Semántica

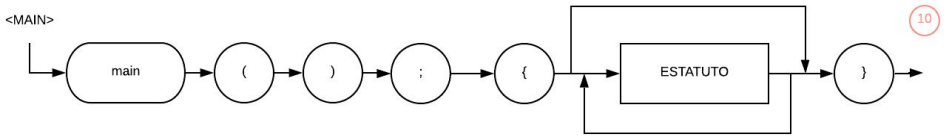


<DIMS>

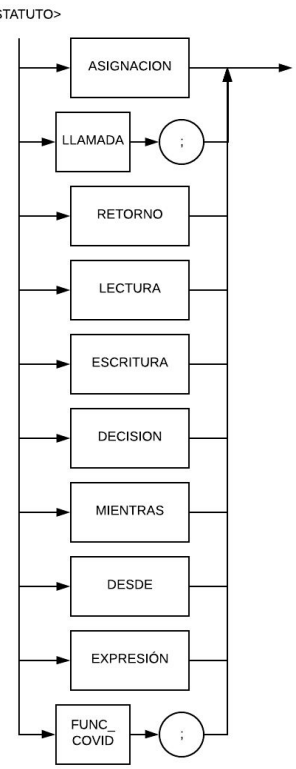




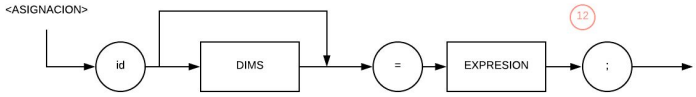
<MAIN>



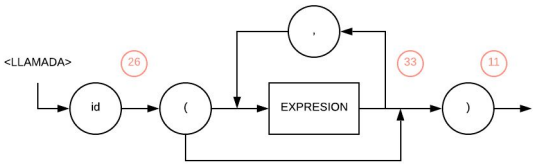
<ESTATUTO>



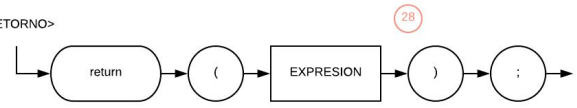
<ASIGNACION>

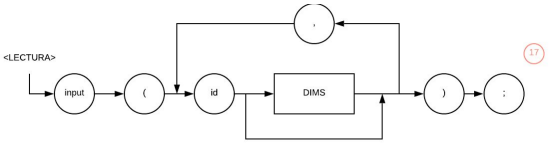


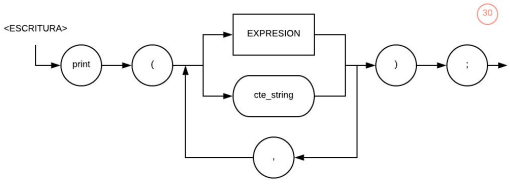
<LLAMADA>

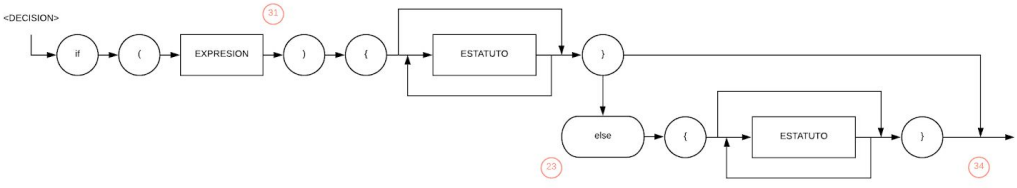


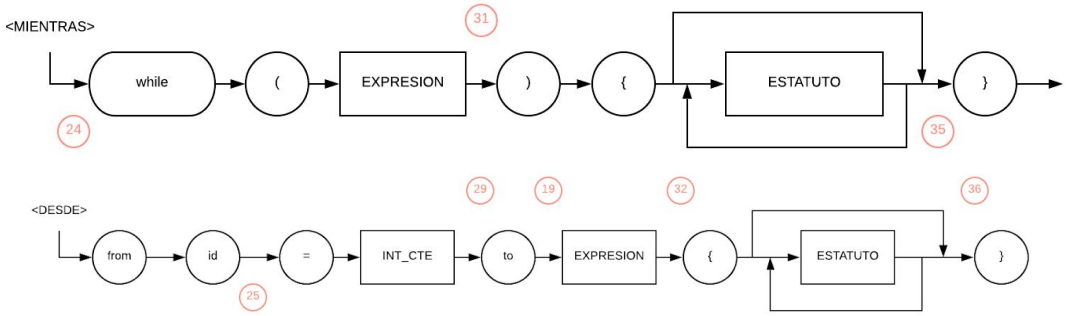
<RETORNO>

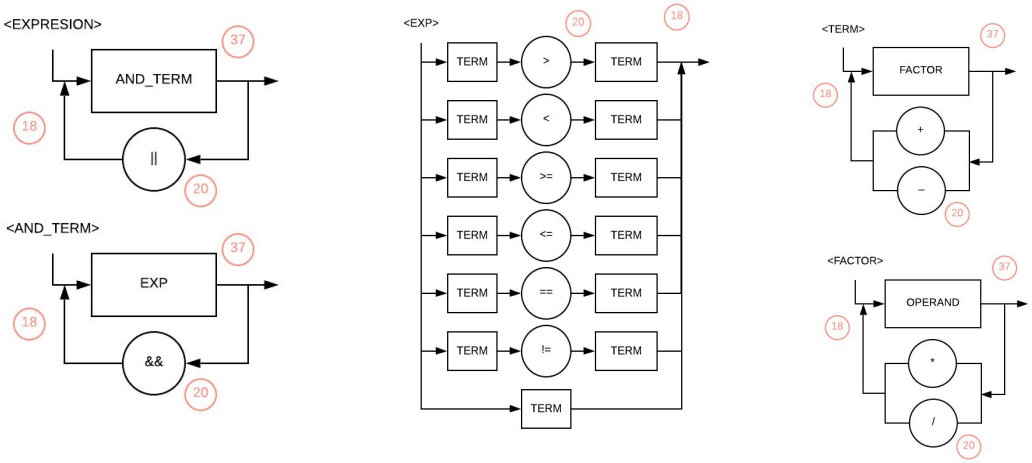


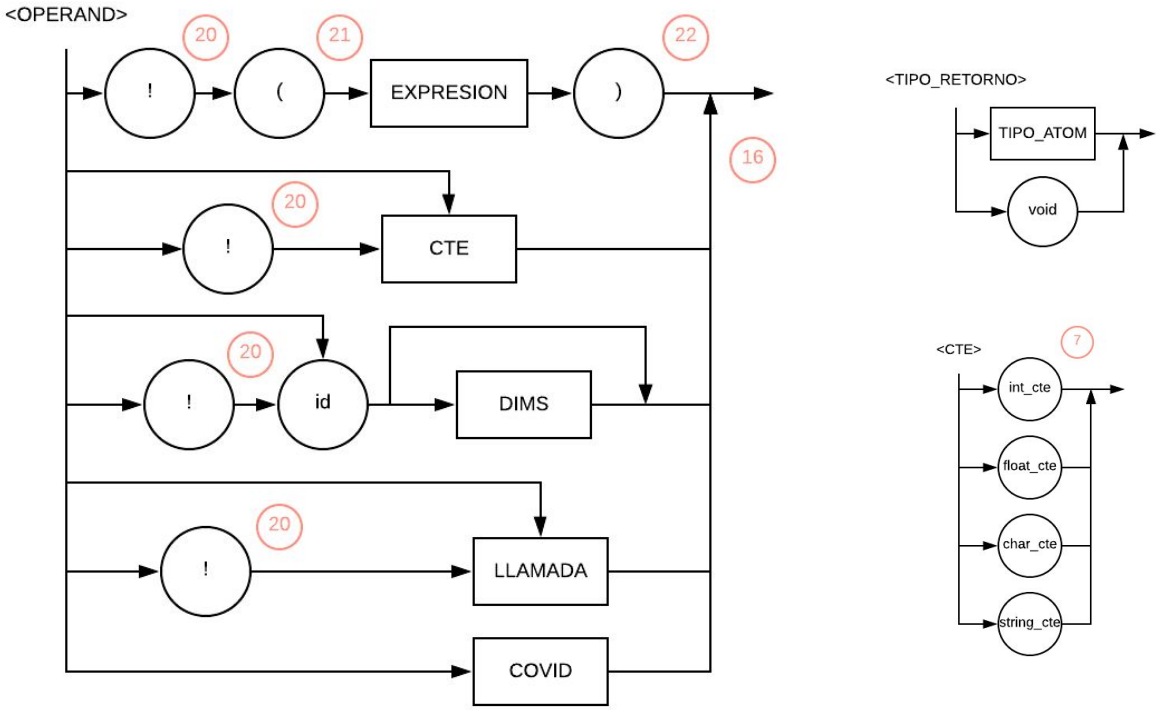


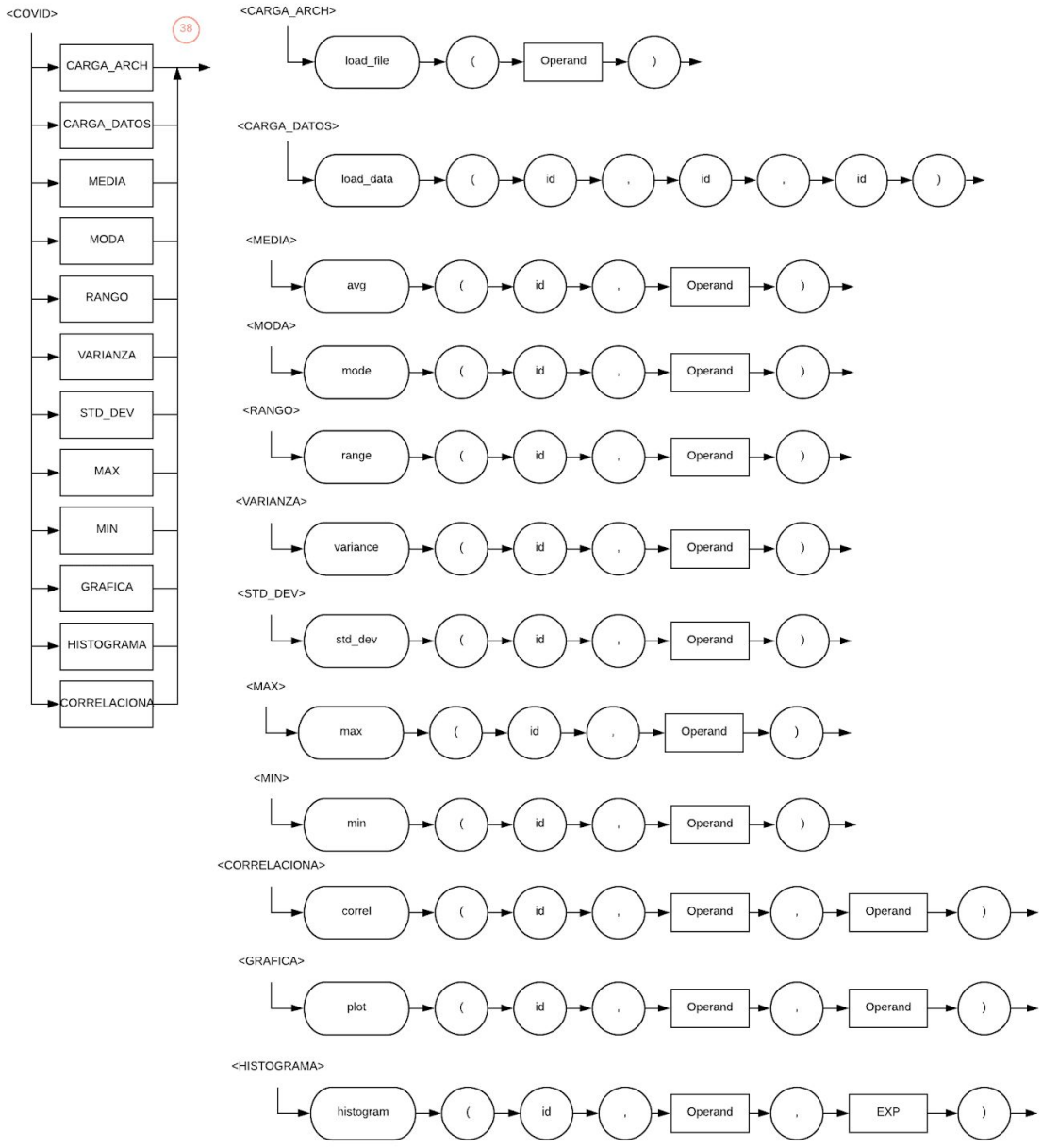





<MIENTRAS>
while
(
EXPRESION
)
{
ESTATUTO
}
<DESDE>
from
id
=
INT_CTE
to
EXPRESION
{
ESTATUTO
}


<EXPRESION>
AND_TERM
||
<AND_TERM>
EXP
&&
<EXP>
TERM
>
<
>=
<=
==
!=
<TERM>
FACTOR
+
-
<FACTOR>
OPERAND
*
/


<OPERAND>
!
(
EXPRESION
)
CTE
id
DIMS
LLAMADA
COVID
<TIPO_RETORNO>
TIPO_ATOM
void
<CTE>
int_cte
float_cte
char_cte
string_cte



1	Dar de alta tabla de funciones
2	- Dar de alta main en tabla de funciones - Cambiar contexto a main
3	Actualizar current type
4	Dar de alta función en la tabla y sus atributos

5	Verificar si id existe y dar de alta en la tabla de variables
6	Dar de alta parámetro en tabla de función
7	Dar de alta constante en tabla de constantes
8	- Cambiar contexto de función - Crear flag para ver si función debe retornar
9	Crear cuádruplo de “ENDPROC”, dar de baja memoria de función
10	Crear cuádruplo de “END”, liberar memoria
11	- Revisar que el número de parámetros coincida con el número de argumentos - Generar cuádruplo GOSUB - Si función retorna generar parche guadalupano
12	Crear cuádruplo de “ASGN”
13	- Crear cuádruplo “VER” - Quitar fondo falso
14	Sumarle a base address
15	- Hacer s2 * m2 - Sumar desplazamiento anterior
16	Agregar a stack de operandos
17	Crear cuádruplo de “INPUT”
18	Revisar compatibilidad de operadores
19	Agregar salto pendiente al stack para for-loop
20	Agregar operador a stack de operadores
21	Agregar fondo falso al stack
22	Quitar fondo false
23	- Generar GOTO para cuando se entra en al true - Resolver GOTO de IF falso - Meter a stack de saltos para resolver el GOTO
24	Meter a stack de saltos inicio de WHILE
25	Agregar iterador 3 veces al stack de operandos (para INCR, COMP y ASGN)
26	- Agregar llamada actual a stack de llamadas por resolver

	- Generar cuádruplo ERA
27	Meter fondo falso
28	- Checar tipo de retorno contra función - Generar cuádruplo RETURN - Validar si función debía retornar
29	Asignar valor a iterador
30	Generar cuádruplo de “PRINT”
31	Generar cuádruplo para salto en falso de WHILE e IF
32	Verificar límite superior de for-loop
33	Validar tipo de argumento contra parámetro
34	Resolver salto anterior de regla de decisión
35	- Generar GOTO de while-loop - Resolver GOTOF de while-loop
36	- Incrementar iterador de for-loop - Crear GOTO de for-loop - Resolver GOTOF de for-loop
37	Resolver expresión
38	- Generar cuádruplo COVID correspondiente - Validación de parámetros

Cubo Semántico

INT	**INT**
=> INT: *, /, -, +, <, >, <=, >=, ==, !=, &&, ||, !, =	

INT	**FLOAT**
=> FLOAT: *, /, -, + **=> INT:** <, >, <=, >=, ==, !=, =	

FLOAT	**INT**
=> FLOAT: *, /, -, +, =	

=> INT: <, >, <=, >=, ==, !=

FLOAT	**FLOAT**
=> FLOAT: *, /, -, +, = **=> INT:** <, >, <=, >=, ==, !=	

CHAR	**CHAR**
=> INT: <, >, <=, >=, ==, != **=> CHAR:** =	

STRING	**CHAR**
=> STRING: =	

STRING	**STRING**
=> INT: <, >, <=, >=, ==, != **=> STRING:** =	

Administración de Memoria

Dir Func (diccionario)

func_name	→	Function				
		name	return_type	var_table	first_quad	param_list
...						
func_name	→	Function				
		name	return_type	var_table	first_quad	param_list

name: string con nombre de variable
return_type: enumerador con el tipo de retorno de la función
var_table: diccionario anidado con las variables en el alcance de la función
first_quad: entero con dirección de primer cuádruplo
param_list: lista con tuplas (dirección, tipo) para especificar los parámetros de la función

Además de esto, cada función tiene un manejador de direcciones que gestiona la asignación y liberación de memoria según tipo de variable.

Var Table (diccionario)

var_name	→	Variable					
		name	data_type	address	dims	d1	d2
...							
var_name	→	Variable					
		name	data_type	address	dims	d1	d2

name: string con nombre de variable
data_type: enumerador con el tipo de la variable
address: dirección virtual de la variable en memoria
dims: número de dimensiones en la variable
d1: tamaño de la primera dimensión de la variable (si la tiene)
d2: tamaño de la segunda dimensión de la variable (si la tiene)

Manejador de direcciones (clase)

Contexto (fijo):

GLOBAL	0
LOCAL	1000
TEMPORAL	2000
CONSTANTE	3000
POINTER	4000

Tipo de dato (diccionario; incrementa cada vez que se asigna memoria):

INT	0
FLOAT	100
CHAR	200
STRING	300
DATAFRAME	900

Primero se define el contexto de un manejador de direcciones (asignándole un valor en la posición de los 1000s, según su alcance). Cada vez que se solicita a memoria una variable, el manejador de direcciones incrementa el contador de acuerdo al espacio de memoria solicitado y al tipo de dato. Finalmente, se suma el valor del contexto con el contador de tipo de dato para generar la dirección (ej: 2012 es una variable temporal, entera).

Liberación de memoria temporal
Adicionalmente, el manejador de direcciones de memoria temporal tiene un diccionario que toma como llave el tipo de dato y que regresa una lista de direcciones. En esta lista de direcciones se almacenan direcciones liberadas de memoria. Cuando se realiza una operación donde los operandos son temporales y no se vuelven a usar, la dirección de estos se agrega a las listas para que sean reutilizados. Cuando se solicita un espacio de memoria temporal, primero se analiza si hay direcciones liberadas previamente. Si no es el caso, se genera una nueva dirección, incrementando el contador. De lo contrario, se elimina la primera dirección de la lista de direcciones disponibles y esta es la que se retorna al pedir un espacio temporal.

Cuádruplos (clase)

oper	op1	op2	res

oper: enumerador con el tipo de operación a realizar
op1: dirección virtual del primer operando de la operación
op2: dirección virtual del segundo operando de la operación
res: dirección virtual para escribir el resultado de la operación

Para la máquina virtual, estos cuádruplos se guardan en una lista. A continuación se presenta un ejemplo de dicha estructura:

```
Quad List:
0.      (GOTO, None, None, 24)
1.      (ASGN, 3000, None, 1002)
2.      (ASGN, 3001, None, 1001)
3.      (SUM, 1000, 3000, 2000)
4.      (LT, 1001, 2000, 2001)
5.      (GOTOF, 2001, None, 10)
6.      (MULT, 1002, 1001, 2000)
7.      (ASGN, 2000, None, 1002)
8.      (INCR, 1001, None, None)
9.      (GOTO, None, None, 3)
10.     (RETURN, None, None, 1002)
11.     (ENDPROC, None, None, None)
...
```

Descripción de la Máquina Virtual

Equipo de cómputo, lenguaje, utilerías

	Librerías utilizadas	**Lenguaje**
VirtualMachine	pandas matplotlib os operator	Python 3.7
Utilities	enum	Python 3.7
SemanticCube	enum	Python 3.7

Se utilizó el mismo equipo de cómputo que en la sección de Compilador

Administración de la memoria

La máquina virtual instancia un objeto de tipo memoria para el contexto global, local, temporal (que toman como argumento el manejador de direcciones que genera la clase de cuadruplas). Además, recibe los bloques de memoria constante y de apuntadores ya creados.

La máquina virtual tiene además una pila de contexto:

...			
fibo	2	local_mem	temp_mem
fibo	2	local_mem	temp_mem
main	13	local_mem	temp_mem

Cada elemento de la pila guarda el nombre del contexto, la posición del último cuádruplo ejecutado o el próximo (en el caso del contexto activo), así como un bloque de memoria local y temporal.

Memoria (clase)
La memoria para la máquina virtual es una lista de espacios contiguos. Ya que se instancia con el manejador de directorios, el bloque de memoria ya sabe la cantidad de espacios necesarios. Además, sabe cómo se distribuye internamente el espacio según los tipos, ya que se cuenta con el espacio que cada tipo requiere. Con esto se colocan pointers de inicio por cada tipo. Cuando se recibe una dirección, se resuelve su tipo de acuerdo al esquema definido en las especificaciones de la memoria virtual. Se obtiene el "desplazamiento" de la casilla según su tipo y se accede al arreglo en la posición tipo_ptr + desplazamiento.

int_ptr			float_ptr		char_ptr string_ptr		
1	24	-3	2.0	3.14	"hello"	"world"	"\n"

Ej: se recibe request para 1301. El contexto (1000) nos indica a cuál instancia de memoria acceder, que en este caso es la local activa. El tipo (300) nos indica que el desplazamiento se debe aplicar al inicio del string_ptr. El desplazamiento (01) nos indica que se requiere acceder al segundo elemento, después del tipo_ptr. Por ende la dirección virtual 1301 se traduce a la séptima casilla de la memoria local.

Arreglos
Dado que la declaración de arreglos del lenguaje es estilo C y solo toma como máximo dos dimensiones, las ecuaciones para acceder un casilla de un arreglo quedan como:

$Address(id[s_1]) = BaseAddress(id) + s_1$
$Address(id[s_1][s_2]) = BaseAddress(id) + s_1 * d_2 + s_2$

En la generación de cuádruplos, se codifican estas sumas y multiplicaciones según sean necesarias para la cantidad de dimensiones del arreglo.

Pruebas del Funcionamiento del Lenguaje

```
program BinarySearch;

var int a[20];

func int binSearch(int left, int right, int target);
var int mid;
{
    if (right >= left) {
        mid = left + (right - left) / 2;

        if(a[mid] == target){
            return (mid);
        }

        if(a[mid] > target){
            return (binSearch(left, mid - 1, target));
        }

        return (binSearch(mid + 1, right, target));
    }

    return (-1);
}

main();
var int num, i, limit, result;
{
    limit = 21;
    while (limit > 20 || limit <= 0) {
        print("Size of array (<20): ");
        input(limit);
    }

    // Read array
    for i = 0 to limit {
        print("a[", i, "]: ");
        input(a[i]);
    }

    print("Number to search: ");
    input(num);
    result = binSearch(0, limit - 1, num);
    print("Result from binSearch: ", result, "\n");
}
```

```
➜  COVID-19-- git:(master) ✗ python3 Covid.py
ExampleFiles/binarysearch.cov -q
Operator Stack:
[]

Operand Stack:
[]

Quad List:
0.      (GOTO, None, None, 34)
1.      (GTE, 1001, 1000, 2000)
2.      (GOTOF, 2000, None, 32)
3.      (SUB, 1001, 1000, 2001)
4.      (DIV, 2001, 3003, 2002)
5.      (SUM, 1000, 2002, 2003)
6.      (ASGN, 2003, None, 1003)
7.      (VER, 1003, 3001, None)
8.      (SUM, 3000, 1003, 4000)
9.      (EQ, 4000, 1002, 2004)
10.     (GOTOF, 2004, None, 12)
11.     (RETURN, None, None, 1003)
12.     (VER, 1003, 3001, None)
13.     (SUM, 3000, 1003, 4001)
14.     (GT, 4001, 1002, 2005)
15.     (GOTOF, 2005, None, 24)
16.     (ERA, binSearch, None, None)
17.     (PARAM, 1000, None, par0)
18.     (SUB, 1003, 3002, 2006)
19.     (PARAM, 2006, None, par1)
20.     (PARAM, 1002, None, par2)
21.     (GOSUB, binSearch, 1, None)
22.     (ASGN, 20, None, 2007)
23.     (RETURN, None, None, 2007)
24.     (ERA, binSearch, None, None)
25.     (SUM, 1003, 3002, 2008)
26.     (PARAM, 2008, None, par0)
27.     (PARAM, 1001, None, par1)
28.     (PARAM, 1002, None, par2)
29.     (GOSUB, binSearch, 1, None)
30.     (ASGN, 20, None, 2009)
31.     (RETURN, None, None, 2009)
32.     (RETURN, None, None, 3004)
33.     (ENDPROC, None, None, None)
34.     (ASGN, 3005, None, 1002)
35.     (GT, 1002, 3001, 2000)
36.     (LTE, 1002, 3000, 2001)
37.     (OR, 2000, 2001, 2002)
38.     (GOTOF, 2002, None, 42)
39.     (PRINT, 3300, None, None)
40.     (INPUT, 1002, None, None)
41.     (GOTO, None, None, 35)
42.     (ASGN, 3000, None, 1001)
43.     (LT, 1001, 1002, 2000)
44.     (GOTOF, 2000, None, 53)
45.     (PRINT, 3301, None, None)
46.     (PRINT, 1001, None, None)
47.     (PRINT, 3302, None, None)
48.     (VER, 1001, 3001, None)
49.     (SUM, 3000, 1001, 4002)
```

```
50.      (INPUT, 4002, None, None)
51.      (INCR, 1001, None, None)
52.      (GOTO, None, None, 43)
53.      (PRINT, 3303, None, None)
54.      (INPUT, 1000, None, None)
55.      (ERA, binSearch, None, None)
56.      (PARAM, 3000, None, par0)
57.      (SUB, 1002, 3002, 2003)
58.      (PARAM, 2003, None, par1)
59.      (PARAM, 1000, None, par2)
60.      (GOSUB, binSearch, 1, None)
61.      (ASGN, 20, None, 2004)
62.      (ASGN, 2004, None, 1003)
63.      (PRINT, 3304, None, None)
64.      (PRINT, 1003, None, None)
65.      (PRINT, 3305, None, None)
66.      (END, None, None, None)


Successful compilation!
Size of array (<20): 5
a[0]: 1
a[1]: 2
a[2]: 3
a[3]: 4
a[4]: 5
Number to search: 2
Result from binSearch: 1
```

```
program Factorial;

func int fact(int n);
var int i, res;
{
    res = 1;
    for i = 2 to n + 1 {
        res = res * i;
    }

    return(res);
}


func int fact_recursive(int n);
{
         if (n == 0) {
                  return(1);
         } else {
                  return(n * fact_recursive(n - 1) );
         }
}

main();
var int i;
{
    print("Input number to get Factorial: ");
    input(i);

    print("Fact Iterative: ", fact(i), "\n");
    print("Fact Recursive: ", fact_recursive(i), "\n");
}
```

```
➜  COVID-19-- git:(master) ✗ python3 Covid.py
ExampleFiles/factorial.cov -q
Operator Stack:
[]

Operand Stack:
[]

Quad List:
0.       (GOTO, None, None, 24)
1.       (ASGN, 3000, None, 1002)
2.       (ASGN, 3001, None, 1001)
3.       (SUM, 1000, 3000, 2000)
4.       (LT, 1001, 2000, 2001)
5.       (GOTOF, 2001, None, 10)
6.       (MULT, 1002, 1001, 2000)
7.       (ASGN, 2000, None, 1002)
8.       (INCR, 1001, None, None)
9.       (GOTO, None, None, 3)
10.      (RETURN, None, None, 1002)
11.      (ENDPROC, None, None, None)
12.      (EQ, 1000, 3002, 2000)
13.      (GOTOF, 2000, None, 16)
14.      (RETURN, None, None, 3000)
15.      (GOTO, None, None, 23)
16.      (ERA, fact_recursive, None, None)
17.      (SUB, 1000, 3000, 2001)
18.      (PARAM, 2001, None, par0)
19.      (GOSUB, fact_recursive, 12, None)
20.      (ASGN, 1, None, 2002)
21.      (MULT, 1000, 2002, 2003)
22.      (RETURN, None, None, 2003)
23.      (ENDPROC, None, None, None)
24.      (PRINT, 3300, None, None)
```

```
25.       (INPUT, 1000, None, None)
26.       (PRINT, 3301, None, None)
27.       (ERA, fact, None, None)
28.       (PARAM, 1000, None, par0)
29.       (GOSUB, fact, 1, None)
30.       (ASGN, 0, None, 2000)
31.       (PRINT, 2000, None, None)
32.       (PRINT, 3302, None, None)
33.       (PRINT, 3303, None, None)
34.       (ERA, fact_recursive, None, None)
35.       (PARAM, 1000, None, par0)
36.       (GOSUB, fact_recursive, 12, None)
37.       (ASGN, 1, None, 2000)
38.       (PRINT, 2000, None, None)
39.       (PRINT, 3302, None, None)
40.       (END, None, None, None)


Successful compilation!
Input number to get Factorial: 7
Fact Iterative: 5040
Fact Recursive: 5040
```

```
program Fibo;

var
         int A, i;

func void fib(int n);
var int t1, t2, nextTerm;
{
    t1 = 0;
    t2 = 1;
    nextTerm = 0;
    for i = 1 to n + 1 {
        if (i == 1) {
           print(t1, "\n");
        }

        else {
            if (i == 2) {
                print (t2, "\n");
            }
            else {
                nextTerm = t1 + t2;
                t1 = t2;
                t2 = nextTerm;
                print(nextTerm, "\n");
            }
        }
    }
}

func int fib_recursive(int n);
{
    if(n <= 1) {
        return (n);
    }
    else {
        return (fib_recursive(n - 1) + fib_recursive(n - 2));
    }
}
```

```
➜  COVID-19-- git:(master) ✗ python3 Covid.py
ExampleFiles/fibo.cov -q
Operator Stack:
[]

Operand Stack:
[]

Quad List:
0.       (GOTO, None, None, 45)
1.       (ASGN, 3001, None, 1001)
2.       (ASGN, 3000, None, 1002)
3.       (ASGN, 3001, None, 1003)
4.       (ASGN, 3000, None, 1)
5.       (SUM, 1000, 3000, 2000)
6.       (LT, 1, 2000, 2001)
7.       (GOTOF, 2001, None, 26)
8.       (EQ, 1, 3000, 2000)
9.       (GOTOF, 2000, None, 13)
10.      (PRINT, 1001, None, None)
11.      (PRINT, 3300, None, None)
12.      (GOTO, None, None, 24)
13.      (EQ, 1, 3002, 2002)
14.      (GOTOF, 2002, None, 18)
15.      (PRINT, 1002, None, None)
16.      (PRINT, 3300, None, None)
17.      (GOTO, None, None, 24)
18.      (SUM, 1001, 1002, 2003)
19.      (ASGN, 2003, None, 1003)
20.      (ASGN, 1002, None, 1001)
21.      (ASGN, 1003, None, 1002)
22.      (PRINT, 1003, None, None)
23.      (PRINT, 3300, None, None)
24.      (INCR, 1, None, None)
25.      (GOTO, None, None, 5)
26.      (PRINT, 3301, None, None)
27.      (ENDPROC, None, None, None)
28.      (LTE, 1000, 3000, 2000)
29.      (GOTOF, 2000, None, 32)
30.      (RETURN, None, None, 1000)
```

```
main();
var int j;
{
    print("A value: ");
    input(A);

    print("Fib iterativo: \n");
    fib(A);

    print("Fib recursivo: \n");
    for j = 0 to A {
        print(fib_recursive(j), "\n");
    }
}
```

```
31.     (GOTO, None, None, 44)
32.     (ERA, fib_recursive, None, None)
33.     (SUB, 1000, 3000, 2001)
34.     (PARAM, 2001, None, par0)
35.     (GOSUB, fib_recursive, 28, None)
36.     (ASGN, 2, None, 2002)
37.     (ERA, fib_recursive, None, None)
38.     (SUB, 1000, 3002, 2003)
39.     (PARAM, 2003, None, par0)
40.     (GOSUB, fib_recursive, 28, None)
41.     (ASGN, 2, None, 2004)
42.     (SUM, 2002, 2004, 2005)
43.     (RETURN, None, None, 2005)
44.     (ENDPROC, None, None, None)
45.     (PRINT, 3302, None, None)
46.     (INPUT, 0, None, None)
47.     (PRINT, 3303, None, None)
48.     (ERA, fib, None, None)
49.     (PARAM, 0, None, par0)
50.     (GOSUB, fib, 1, None)
51.     (PRINT, 3304, None, None)
52.     (ASGN, 3001, None, 1000)
53.     (LT, 1000, 0, 2000)
54.     (GOTOF, 2000, None, 63)
55.     (ERA, fib_recursive, None, None)
56.     (PARAM, 1000, None, par0)
57.     (GOSUB, fib_recursive, 28, None)
58.     (ASGN, 2, None, 2001)
59.     (PRINT, 2001, None, None)
60.     (PRINT, 3300, None, None)
61.     (INCR, 1000, None, None)
62.     (GOTO, None, None, 53)
63.     (END, None, None, None)

Successful compilation!
A value: 10
Fib iterativo:
0 1 1 2 3 5 8 13 21 34
Fib recursivo:
0 1 1 2 3 5 8 13 21 34
```

```
program MergeSort;

var int a[20], b[20], limit;

func void printArr();
var int i;
{
    for i = 0 to limit {
        print(a[i], " ");
    }
    print("\n");
}

func void merge(int low, int mid, int high);
var int l1, l2, i;
{
    l1 = low;
    l2 = mid + 1;
    i = low;

    while (l1 <= mid && l2 <= high) {
        if(a[l1] <= a[l2]) {
```

```
➜  COVID-19-- git:(master) ✗ python3 Covid.py
ExampleFiles/mergesort.cov -q
Operator Stack:
[]

Operand Stack:
[]

Quad List:
0.      (GOTO, None, None, 100)
1.      (ASGN, 3000, None, 1000)
2.      (LT, 1000, 40, 2000)
3.      (GOTOF, 2000, None, 10)
4.      (VER, 1000, 3001, None)
5.      (SUM, 3000, 1000, 4000)
6.      (PRINT, 4000, None, None)
7.      (PRINT, 3300, None, None)
8.      (INCR, 1000, None, None)
9.      (GOTO, None, None, 2)
10.     (PRINT, 3301, None, None)
11.     (ENDPROC, None, None, None)
12.     (ASGN, 1000, None, 1003)
```

```
            b[i] = a[l1];
            l1 = l1 + 1;
        }
        else {
            b[i] = a[l2];
            l2 = l2 + 1;
        }
        i = i + 1;
    }

    while(l1 <= mid) {
        b[i] = a[l1];
        i = i + 1;
        l1 = l1 + 1;
    }

    while(l2 <= high) {
        b[i] = a[l2];
        i = i + 1;
        l2 = l2 + 1;
    }

    for i = low to (high + 1) {
        a[i] = b[i];
    }
}

func void sort(int low, int high);
var int mid;
{
    if( low < high) {
        mid = (low + high) / 2;
        sort(low, mid);
        sort(mid + 1, high);
        merge(low, mid, high);
    }
}

main();
var int i;
{
    limit = 21;
    while (limit > 20 || limit <= 0) {
        print("Size of array (<20): ");
        input(limit);
    }

    // Read array
    for i = 0 to limit {
        print("a[", i, "]: ");
        input(a[i]);
    }

    print("List before sorting \n");
    printArr();

    sort(0, limit - 1);

    print("List after sorting \n");
    printArr();
}
```

```
13.      (SUM, 1001, 3002, 2000)
14.      (ASGN, 2000, None, 1004)
15.      (ASGN, 1000, None, 1005)
16.      (LTE, 1003, 1001, 2000)
17.      (LTE, 1004, 1002, 2001)
18.      (AND, 2000, 2001, 2002)
19.      (GOTOF, 2002, None, 44)
20.      (VER, 1003, 3001, None)
21.      (SUM, 3000, 1003, 4001)
22.      (VER, 1004, 3001, None)
23.      (SUM, 3000, 1004, 4002)
24.      (LTE, 4001, 4002, 2000)
25.      (GOTOF, 2000, None, 34)
26.      (VER, 1005, 3001, None)
27.      (SUM, 3001, 1005, 4003)
28.      (VER, 1003, 3001, None)
29.      (SUM, 3000, 1003, 4004)
30.      (ASGN, 4004, None, 4003)
31.      (SUM, 1003, 3002, 2003)
32.      (ASGN, 2003, None, 1003)
33.      (GOTO, None, None, 41)
34.      (VER, 1005, 3001, None)
35.      (SUM, 3001, 1005, 4005)
36.      (VER, 1004, 3001, None)
37.      (SUM, 3000, 1004, 4006)
38.      (ASGN, 4006, None, 4005)
39.      (SUM, 1004, 3002, 2004)
40.      (ASGN, 2004, None, 1004)
41.      (SUM, 1005, 3002, 2005)
42.      (ASGN, 2005, None, 1005)
43.      (GOTO, None, None, 16)
44.      (LTE, 1003, 1001, 2006)
45.      (GOTOF, 2006, None, 56)
46.      (VER, 1005, 3001, None)
47.      (SUM, 3001, 1005, 4007)
48.      (VER, 1003, 3001, None)
49.      (SUM, 3000, 1003, 4008)
50.      (ASGN, 4008, None, 4007)
51.      (SUM, 1005, 3002, 2007)
52.      (ASGN, 2007, None, 1005)
53.      (SUM, 1003, 3002, 2008)
54.      (ASGN, 2008, None, 1003)
55.      (GOTO, None, None, 44)
56.      (LTE, 1004, 1002, 2009)
57.      (GOTOF, 2009, None, 68)
58.      (VER, 1005, 3001, None)
59.      (SUM, 3001, 1005, 4009)
60.      (VER, 1004, 3001, None)
61.      (SUM, 3000, 1004, 4010)
62.      (ASGN, 4010, None, 4009)
63.      (SUM, 1005, 3002, 2010)
64.      (ASGN, 2010, None, 1005)
65.      (SUM, 1004, 3002, 2011)
66.      (ASGN, 2011, None, 1004)
67.      (GOTO, None, None, 56)
68.      (ASGN, 1000, None, 1005)
69.      (SUM, 1002, 3002, 2012)
70.      (LT, 1005, 2012, 2013)
71.      (GOTOF, 2013, None, 79)
72.      (VER, 1005, 3001, None)
73.      (SUM, 3000, 1005, 4011)
74.      (VER, 1005, 3001, None)
75.      (SUM, 3001, 1005, 4012)
76.      (ASGN, 4012, None, 4011)
77.      (INCR, 1005, None, None)
```

```
78.      (GOTO, None, None, 69)
79.      (ENDPROC, None, None, None)
80.      (LT, 1000, 1001, 2000)
81.      (GOTOF, 2000, None, 99)
82.      (SUM, 1000, 1001, 2001)
83.      (DIV, 2001, 3003, 2002)
84.      (ASGN, 2002, None, 1002)
85.      (ERA, sort, None, None)
86.      (PARAM, 1000, None, par0)
87.      (PARAM, 1002, None, par1)
88.      (GOSUB, sort, 80, None)
89.      (ERA, sort, None, None)
90.      (SUM, 1002, 3002, 2003)
91.      (PARAM, 2003, None, par0)
92.      (PARAM, 1001, None, par1)
93.      (GOSUB, sort, 80, None)
94.      (ERA, merge, None, None)
95.      (PARAM, 1000, None, par0)
96.      (PARAM, 1002, None, par1)
97.      (PARAM, 1001, None, par2)
98.      (GOSUB, merge, 12, None)
99.      (ENDPROC, None, None, None)
100.     (ASGN, 3004, None, 40)
101.     (GT, 40, 3001, 2000)
102.     (LTE, 40, 3000, 2001)
103.     (OR, 2000, 2001, 2002)
104.     (GOTOF, 2002, None, 108)
105.     (PRINT, 3302, None, None)
106.     (INPUT, 40, None, None)
107.     (GOTO, None, None, 101)
108.     (ASGN, 3000, None, 1000)
109.     (LT, 1000, 40, 2000)
110.     (GOTOF, 2000, None, 119)
111.     (PRINT, 3303, None, None)
112.     (PRINT, 1000, None, None)
113.     (PRINT, 3304, None, None)
114.     (VER, 1000, 3001, None)
115.     (SUM, 3000, 1000, 4013)
116.     (INPUT, 4013, None, None)
117.     (INCR, 1000, None, None)
118.     (GOTO, None, None, 109)
119.     (PRINT, 3305, None, None)
120.     (ERA, printArr, None, None)
121.     (GOSUB, printArr, 1, None)
122.     (ERA, sort, None, None)
123.     (PARAM, 3000, None, par0)
124.     (SUB, 40, 3002, 2003)
125.     (PARAM, 2003, None, par1)
126.     (GOSUB, sort, 80, None)
127.     (PRINT, 3306, None, None)
128.     (ERA, printArr, None, None)
129.     (GOSUB, printArr, 1, None)
130.     (END, None, None, None)

Successful compilation!
Size of array (<20): 5
a[0]: 43
a[1]: 1
a[2]: 24
a[3]: 65
a[4]: 3
List before sorting
43 1 24 65 3
List after sorting
1 3 24 43 65
```

```
program QuickSort;

var int a[20], limit;

func void printArr();
var int i;
{
    for i = 0 to limit {
        print(a[i], " ");
    }
    print("\n");
}

func void swap(int x, int y);
var int temp;
{
    temp = a[x];
    a[x] = a[y];
    a[y] = temp;
}

func int partition(int lo, int hi);
var int pivot, i, j;
{
    pivot = a[hi];
    i = lo - 1;

    for j = lo to hi {
        if (a[j] < pivot) {
            i = i + 1;
            swap(i, j);
        }
    }

    swap(i + 1, hi);
    return(i + 1);
}

func void quicksort(int lo, int hi);
var int pi;
{
    if (lo < hi) {
        pi = partition(lo, hi);

        quicksort(lo, pi - 1);
        quicksort(pi + 1, hi);
    }
}

main();
var int i;
{
    limit = 21;
    while (limit > 20 || limit <= 0) {
        print("Size of array (<20): ");
        input(limit);
    }

    // Read array
    for i = 0 to limit {
        print("a[", i, "]: ");
        input(a[i]);
    }
```

```
➜  COVID-19-- git:(master) ✗ python3 Covid.py
ExampleFiles/quicksort.cov -q
Operator Stack:
[]

Operand Stack:
[]

Quad List:
0.      (GOTO, None, None, 71)
1.      (ASGN, 3000, None, 1000)
2.      (LT, 1000, 20, 2000)
3.      (GOTOF, 2000, None, 10)
4.      (VER, 1000, 3001, None)
5.      (SUM, 3000, 1000, 4000)
6.      (PRINT, 4000, None, None)
7.      (PRINT, 3300, None, None)
8.      (INCR, 1000, None, None)
9.      (GOTO, None, None, 2)
10.     (PRINT, 3301, None, None)
11.     (ENDPROC, None, None, None)
12.     (VER, 1000, 3001, None)
13.     (SUM, 3000, 1000, 4001)
14.     (ASGN, 4001, None, 1002)
15.     (VER, 1000, 3001, None)
16.     (SUM, 3000, 1000, 4002)
17.     (VER, 1001, 3001, None)
18.     (SUM, 3000, 1001, 4003)
19.     (ASGN, 4003, None, 4002)
20.     (VER, 1001, 3001, None)
21.     (SUM, 3000, 1001, 4004)
22.     (ASGN, 1002, None, 4004)
23.     (ENDPROC, None, None, None)
24.     (VER, 1001, 3001, None)
25.     (SUM, 3000, 1001, 4005)
26.     (ASGN, 4005, None, 1002)
27.     (SUB, 1000, 3002, 2000)
28.     (ASGN, 2000, None, 1003)
29.     (ASGN, 1000, None, 1004)
30.     (LT, 1004, 1001, 2000)
31.     (GOTOF, 2000, None, 44)
32.     (VER, 1004, 3001, None)
33.     (SUM, 3000, 1004, 4006)
34.     (LT, 4006, 1002, 2001)
35.     (GOTOF, 2001, None, 42)
36.     (SUM, 1003, 3002, 2002)
37.     (ASGN, 2002, None, 1003)
38.     (ERA, swap, None, None)
39.     (PARAM, 1003, None, par0)
40.     (PARAM, 1004, None, par1)
41.     (GOSUB, swap, 12, None)
42.     (INCR, 1004, None, None)
43.     (GOTO, None, None, 30)
44.     (ERA, swap, None, None)
45.     (SUM, 1003, 3002, 2003)
46.     (PARAM, 2003, None, par0)
47.     (PARAM, 1001, None, par1)
48.     (GOSUB, swap, 12, None)
49.     (SUM, 1003, 3002, 2004)
50.     (RETURN, None, None, 2004)
51.     (ENDPROC, None, None, None)
52.     (LT, 1000, 1001, 2000)
53.     (GOTOF, 2000, None, 70)
54.     (ERA, partition, None, None)
```

```
    print("List before sorting \n");
    printArr();

    quicksort(0, limit - 1);

    print("List after sorting \n");
    printArr();
}
```

```
55.      (PARAM, 1000, None, par0)
56.      (PARAM, 1001, None, par1)
57.      (GOSUB, partition, 24, None)
58.      (ASGN, 21, None, 2001)
59.      (ASGN, 2001, None, 1002)
60.      (ERA, quicksort, None, None)
61.      (PARAM, 1000, None, par0)
62.      (SUB, 1002, 3002, 2002)
63.      (PARAM, 2002, None, par1)
64.      (GOSUB, quicksort, 52, None)
65.      (ERA, quicksort, None, None)
66.      (SUM, 1002, 3002, 2003)
67.      (PARAM, 2003, None, par0)
68.      (PARAM, 1001, None, par1)
69.      (GOSUB, quicksort, 52, None)
70.      (ENDPROC, None, None, None)
71.      (ASGN, 3003, None, 20)
72.      (GT, 20, 3001, 2000)
73.      (LTE, 20, 3000, 2001)
74.      (OR, 2000, 2001, 2002)
75.      (GOTOF, 2002, None, 79)
76.      (PRINT, 3302, None, None)
77.      (INPUT, 20, None, None)
78.      (GOTO, None, None, 72)
79.      (ASGN, 3000, None, 1000)
80.      (LT, 1000, 20, 2000)
81.      (GOTOF, 2000, None, 90)
82.      (PRINT, 3303, None, None)
83.      (PRINT, 1000, None, None)
84.      (PRINT, 3304, None, None)
85.      (VER, 1000, 3001, None)
86.      (SUM, 3000, 1000, 4007)
87.      (INPUT, 4007, None, None)
88.      (INCR, 1000, None, None)
89.      (GOTO, None, None, 80)
90.      (PRINT, 3305, None, None)
91.      (ERA, printArr, None, None)
92.      (GOSUB, printArr, 1, None)
93.      (ERA, quicksort, None, None)
94.      (PARAM, 3000, None, par0)
95.      (SUB, 20, 3002, 2003)
96.      (PARAM, 2003, None, par1)
97.      (GOSUB, quicksort, 52, None)
98.      (PRINT, 3306, None, None)
99.      (ERA, printArr, None, None)
100.     (GOSUB, printArr, 1, None)
101.     (END, None, None, None)


Successful compilation!
Size of array (<20): 5
a[0]: 23
a[1]: 23
a[2]: 5
a[3]: 1
a[4]: 4
List before sorting
23 23 5 1 4
List after sorting
1 4 5 23 23
```

```
program Dataframe;
var dataframe data;
```

```
➜  COVID-19-- git:(master) ✗ python3 Covid.py
ExampleFiles/dataframe.cov -q
Operator Stack:
```

```
main();
var int rows, cols;
    float res;
{
    load_file("song_data_clean.csv");
    load_data(data, rows, cols);

    print("Number of rows: ", rows, "\n");
    print("Number of cols: ", cols, "\n");

    print("\nAverage: ", avg(data, "song_popularity"), "\n");
    print("Mode: ", mode(data, "song_popularity"), "\n");
    print("Range: ", range(data, "song_popularity"), "\n");
    print("Variance: ", variance(data, "song_popularity"),
"\n");
    print("Std_Dev: ", std_dev(data, "song_popularity"),
"\n");
    print("Max: ", max(data, "song_popularity"), "\n");
    print("Min: ", min(data, "song_popularity"), "\n");
    print("Correl: ", correl(data, "song_popularity",
"audio_valence"), "\n");

    plot(data, "song_popularity", "audio_valence");
    histogram(data, "song_popularity", 100);
}
```

```
[]

Operand Stack:
[]

Quad List:
0.      (GOTO, None, None, 1)
1.      (FILE, 3300, None, None)
2.      (DATA, 1000, 1001, None)
3.      (PRINT, 3301, None, None)
4.      (PRINT, 1000, None, None)
5.      (PRINT, 3302, None, None)
6.      (PRINT, 3303, None, None)
7.      (PRINT, 1001, None, None)
8.      (PRINT, 3302, None, None)
9.      (PRINT, 3304, None, None)
10.     (AVG, 3305, None, 2100)
11.     (PRINT, 2100, None, None)
12.     (PRINT, 3302, None, None)
13.     (PRINT, 3306, None, None)
14.     (MODE, 3305, None, 2101)
15.     (PRINT, 2101, None, None)
16.     (PRINT, 3302, None, None)
17.     (PRINT, 3307, None, None)
18.     (RANGE, 3305, None, 2102)
19.     (PRINT, 2102, None, None)
20.     (PRINT, 3302, None, None)
21.     (PRINT, 3308, None, None)
22.     (VAR, 3305, None, 2103)
23.     (PRINT, 2103, None, None)
24.     (PRINT, 3302, None, None)
25.     (PRINT, 3309, None, None)
26.     (STD_DEV, 3305, None, 2104)
27.     (PRINT, 2104, None, None)
28.     (PRINT, 3302, None, None)
29.     (PRINT, 3310, None, None)
30.     (MAX, 3305, None, 2105)
31.     (PRINT, 2105, None, None)
32.     (PRINT, 3302, None, None)
33.     (PRINT, 3311, None, None)
34.     (MIN, 3305, None, 2106)
35.     (PRINT, 2106, None, None)
36.     (PRINT, 3302, None, None)
37.     (PRINT, 3312, None, None)
38.     (CORREL, 3305, 3313, 2107)
39.     (PRINT, 2107, None, None)
40.     (PRINT, 3302, None, None)
41.     (PLOT, 3305, 3313, None)
42.     (HIST, 3305, 3001, None)
43.     (END, None, None, None)


Successful compilation!
Number of rows: 13053
Number of cols: 15

Average: 48.48448632498276
Mode: 52.0
Range: 100.0
Variance: 404.52038832981987
Std_Dev: 20.11269221983521
Max: 100.0
Min: 0.0
Correl: -0.04976362384324907
```

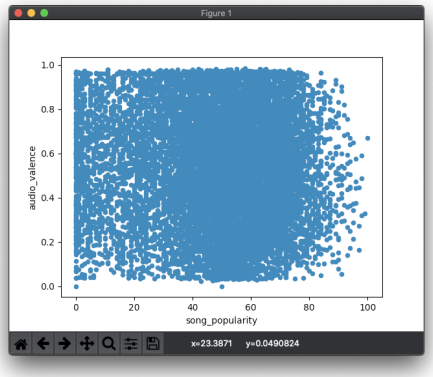


Figure 1
audio_valence
song_popularity
x=23.3871 y=0.0490824

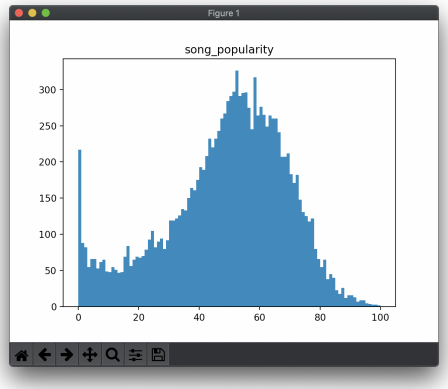


Figure 1
song_popularity

Código

Lifecycle			
Driver Covid.py	**Tabla de var y funcs** DirFunc.py	**Código Intermedio** Quadruples.py	**Máquina Virtual** VirtualMachine.py
Manda a llamar a todas las clases involucradas en el proceso de compilación y ejecución.	Llena las tablas de funciones y de variables. Además, da de alta las constantes en el espacio de memoria. Y genera los directorios para cada función del programa.	Genera la lista de cuadruplos que funcionan como representación intermedia para el proceso de compilación. Toma en cuenta las acciones semánticas necesarias.	Convierte la representación intermedia de un 3AC a representación interna la cuál es implementación de python.

Covid.py

```
import sys
from antlr4 import *
from antlr.CovidLexer import CovidLexer
from antlr.CovidParser import CovidParser
from antlr.CovidListener import CovidListener
from DirFunc import DirFunc
from Quadruples import QuadrupleList
from VirtualMachine import VirtualMachine
from antlr4.tree.Trees import Trees


def main(argv):
    input_stream = FileStream(argv[1])
    lexer = CovidLexer(input_stream)
    stream = CommonTokenStream(lexer)
    parser = CovidParser(stream)
    tree = parser.start()

    if parser.getNumberOfSyntaxErrors() != 0:
        print("Compilation unsuccessful: Syntax
Error(s)")
        sys.exit()

    # Create DirFunc using tree walkers
    dir_func = DirFunc()
    walker = ParseTreeWalker()
    walker.walk(dir_func, tree)

    # Generate list of quads
    quad_list = QuadrupleList(dir_func)
    walker = ParseTreeWalker()
    walker.walk(quad_list, tree)

    # Instantiate virtual machine
    virtual_machine = VirtualMachine(
        dir_func.func_table,
        quad_list.quad_list,
        quad_list.cte_address_dir,
        quad_list.pointer_mem,
        dir_func.global_address_dir
    )

    # Debug flags
    if len(argv) >= 3:
        if '-q' in argv: # prints quadruples
            print(quad_list)
        if '-d' in argv: # prints dir func
            print(dir_func)

    # If no errors and successful compilation, run VM
    if parser.getNumberOfSyntaxErrors() == 0:
        print("Successful compilation!")
        virtual_machine.run()

if __name__ == '__main__':
    # Catch Keyboard interrupt
    try:
        main(sys.argv)
    except KeyboardInterrupt:
        print("\nEnded program due to keyboard
interrupt.\n")
        sys.exit()
```

DirFunc.py

```
class DirFunc(CovidListener):
    """
    Used to create the function directory, and
variable table, instantiate constant memory
    """
    func_table = {}
    curr_scope = ""
    curr_type = None

    constants = []

    global_address_dir = GlobalAddressDir()
    cte_address_dir = CteMemory()

    dataframe_mem = None
    dataframe_exists = False

def enterFunc(self, ctx):
        # Get function name and type, update scope
        func_name = ctx.ID().getText()
        func_type =
Type[ctx.getChild(1).getText().upper()]

        self.curr_scope = func_name

        # Add function to func_table
        if not func_name in self.func_table:
            self.func_table[func_name] =
Function(func_name, func_type, {})

            # Insert function into global table, to
store return value
```

```
            if func_type != Type.VOID:
                func_address =
self.global_address_dir.getAddress(func_type)

self.func_table["global"].var_table[func_name] =
Variable(func_name, func_type, func_address)
        else:
            # Throw error if function is redefined
            print(f"[Error: {ctx.start.line}]
Redefinition of function {func_name}")
            sys.exit()

def addVariable(self, ctx, index):
        var_name = ctx.ID().getText()
        var_table =
self.func_table[self.curr_scope].var_table

        d1 = 1
        d2 = 1
        dim = 0
        address_pointer = None

        # Determine if var has dimensions
        if isinstance(ctx,
CovidParser.Ident_declContext):
            if ctx.getChildCount() == 4:
                d1 = int(ctx.getChild(2).getText())
                dim = 1
            elif ctx.getChildCount() == 7:
                d1 = int(ctx.getChild(2).getText())
                d2 = int(ctx.getChild(5).getText())
                dim = 2

        d1_str = str(d1)
        d2_str = str(d2)

        # Is a parameter
        if index >= 0:
            self.curr_type =
Type[ctx.getChild(index).getText().upper()]

        # Add variable to var_table
        if not var_name in var_table:
            # Choose scope to insert table
            if self.curr_scope == "global":
                address =
self.global_address_dir.getAddress(self.curr_type, d1
* d2)
            else:
                address =
self.func_table[self.curr_scope].address_dir.addLocal
(self.curr_type, d1 * d2)

            # Is a parameter, fill details on
func_table for param list
            if index >= 0:

self.func_table[self.curr_scope].params.append((addre
ss, self.curr_type))

            # If it has dimensions, gets starting
address, appends dimensions to constant memory
            if dim > 0:
                if str(address) not in
self.constants:
                    address_pointer =
self.cte_address_dir.addConstant(Type.INT,
str(address))

self.constants.append(str(address))
                else:
                    address_pointer =
self.cte_address_dir.address_table[str(address)]

            # Solve address for first dimension
            if d1_str not in self.constants:
                address_d1 =
self.cte_address_dir.addConstant(Type.INT, d1_str)
                self.constants.append(d1_str)
            else:
                address_d1 =
self.cte_address_dir.address_table[str(d1)]

            # Solve address for second dimension
            if d2_str not in self.constants:
                address_d2 =
self.cte_address_dir.addConstant(Type.INT, d2_str)
                self.constants.append(d2_str)
            else:
                address_d2 =
self.cte_address_dir.address_table[d2_str]

            var_table[var_name] = Variable(var_name,
self.curr_type, address, dim, address_d1, address_d2,
address_pointer)

        else:
            print(f"[Error: {ctx.start.line}]
Redefinition of variable {var_name}")
            sys.exit()
```

Memory.py

```
class AddressDir:
    def __init__(self, context):
        """
        Class stores the space needed for variables
per type
        Provides functions to get address for a
variable and get the size of a function
        """
        self.context = context

        self.addresses = {
            Type.INT: 0,
            Type.FLOAT: 0,
            Type.CHAR: 0,
            Type.STRING: 0,
            Type.DATAFRAME: 0
        }

    def getAddress(self, data_type, size = 1):
        # Return next address
        pointer_val = self.addresses[data_type]
        self.addresses[data_type] += size

        # Check if stack has been exceeded
        if self.addresses[data_type] > 100:
            print("[Error] Memory stack exceeded for
type and context")
            sys.exit()

        context_val = self.context * 1000

        if data_type == Type.INT:
            data_type_val = 0
        elif data_type == Type.FLOAT:
            data_type_val = 100
        elif data_type == Type.CHAR:
            data_type_val = 200
        elif data_type == Type.STRING:
            data_type_val = 300

        return context_val + data_type_val +
pointer_val

    def getSize(self):
        return sum(self.addresses.values())

    def __repr__(self):
        result = f"Context {self.context}\n
Addresses: {self.addresses}\n"
        return result

class Memory:
    def __init__(self, directory):
```

```
        """
        Generates a array of the size needed to
represent the function memory
        Creates pointers to access the space of each
memory
        Provides functionality to get values given an
address and write in addresses given address and
value
        """
        self.int_pointer = 0
        self.float_pointer = self.int_pointer +
directory.addresses[Type.INT]
        self.char_pointer = self.float_pointer +
directory.addresses[Type.FLOAT]
        self.string_pointer = self.char_pointer +
directory.addresses[Type.CHAR]
        self.df_pointer = self.string_pointer +
directory.addresses[Type.STRING]

        self.size = self.df_pointer +
directory.addresses[Type.DATAFRAME]
        self.space = [None for x in range(self.size)]

    def getValue(self, address):
        # Obtain the value for a var, given an
address
        data_type_val = (address % 1000) // 100
        pointer_val = address % 100

        try:
            # bloque a intentar
            if data_type_val == 0:
                return
int(self.space[self.int_pointer + pointer_val])
            elif data_type_val == 1:
                return
float(self.space[self.float_pointer + pointer_val])
            elif data_type_val == 2:
                return self.space[self.char_pointer +
pointer_val]
            elif data_type_val == 3:
                return self.space[self.string_pointer
+ pointer_val]
        except:
            print("[Error] uninitialized variable")
            sys.exit()

    def writeAddress(self, address, value):
        # Write a value for a var, given an address
and the value
        data_type_val = (address % 1000) // 100
        pointer_val = address % 100

        if data_type_val == 0:
            self.space[self.int_pointer +
pointer_val] = value
        elif data_type_val == 1:
            self.space[self.float_pointer +
pointer_val] = value
        elif data_type_val == 2:
            self.space[self.char_pointer +
pointer_val] = value
        elif data_type_val == 3:
            self.space[self.string_pointer +
pointer_val] = value
```

Quadruples.py

```
class QuadrupleList(CovidListener):
    """
    Generates a list of quads for intermediate code
repr. and does semantic checks
    """

    quad_list = []
    operand_stack = []
    operator_stack = []
    jump_stack = []
    curr_scope = ""
    call_stack = []
    quad_counter = 0
    k = 0
    has_return = None
    pointer_mem = PointerMemory()

    def __init__(self, dir_func):
        self.func_table = dir_func.func_table
        self.cte_address_dir =
dir_func.cte_address_dir


self.createQuadruple(Quadruple(Operator.GOTO,None,Non
e,None))

def addOperandToStack(self, var_name, line_num):
        local_table =
self.func_table[self.curr_scope].var_table
        global_table =
self.func_table["global"].var_table

        # Check where the variable belongs (global or
local)
        if var_name in local_table:
            table = local_table
        elif var_name in global_table:
            table = global_table
        else:
            print(f"[Error: {line_num}] use of
undeclared variable: {var_name}")
            sys.exit()

        # Get relevant data for the var
        dims = table[var_name].dims
        start = table[var_name].address
        address_start =
table[var_name].address_pointer
        data_type = table[var_name].data_type
        d1_addr = table[var_name].d1
        d2_addr = table[var_name].d2

        if dims == 1:
            # Get index and throw error if the type
is not int
            s1, s1_type = self.operand_stack.pop()

            if s1_type != Type.INT:
                print(f"[Error: {line_num}] Array
index is not an int")
                sys.exit()

            # Verify that index is within bounds
            ver_quad = Quadruple(Operator.VER, None,
s1, d1_addr)
            self.createQuadruple(ver_quad)

            # Obtain pointer and push to operand
stack
            pointer = self.pointer_mem.getPointer()

            sum_quad = Quadruple(Operator.SUM,
pointer, address_start, s1)
            self.createQuadruple(sum_quad)
            self.operand_stack.append((pointer,
data_type))
```

```
        elif dims == 2:
            # Get indices
            s2, s2_type = self.operand_stack.pop()
            s1, s1_type = self.operand_stack.pop()

            if s1_type != Type.INT or s2_type !=
Type.INT:
                print(f"[Error: {line_num}] Array
index is not an int")
                sys.exit()

            # Verify first index is within bounds
            ver_quad = Quadruple(Operator.VER, None,
s1, d1_addr)
            self.createQuadruple(ver_quad)

            # Shift to proper row
            mult_res =
self.func_table[self.curr_scope].address_dir.addTemp(
Type.INT)
            mult_quad = Quadruple(Operator.MULT,
mult_res, s1, d2_addr)
            self.createQuadruple(mult_quad)

            # Verify second index is within bounds
            ver_quad = Quadruple(Operator.VER, None,
s2, d2_addr)
            self.createQuadruple(ver_quad)

            # Apply row shift to start of array
            sum_res =
self.func_table[self.curr_scope].address_dir.addTemp(
Type.INT)
            sum_quad = Quadruple(Operator.SUM,
sum_res, address_start, mult_res)
            self.createQuadruple(sum_quad)

            # Obtain pointer and push to operand
stack
            pointer = self.pointer_mem.getPointer()
            sum_quad = Quadruple(Operator.SUM,
pointer, sum_res, s2)
            self.createQuadruple(sum_quad)
            self.operand_stack.append((pointer,
data_type))

        else:
            self.operand_stack.append((start,
data_type))

def parseFourTupleQuad(self, operands, line_num):
        # Parse most four tuple quads

        if not self.operator_stack:
            return

        if self.operator_stack[-1] in operands:

            r_operand, r_type =
self.operand_stack.pop()
            l_operand, l_type =
self.operand_stack.pop()
            operator = self.operator_stack.pop()

            # Verify semantics for both types of
operands with a given operator
            result_type =
semantic_cube[l_type][r_type][operator]

            if result_type != None:
                result_addr =
self.func_table[self.curr_scope].address_dir.addTemp(
result_type)
                # Create respective quad
                quad = Quadruple(operator,
result_addr, l_operand, r_operand)
                self.createQuadruple(quad)
self.operand_stack.append((result_addr, result_type))

                # Check if addresses must be released
(if temp)
                if (r_operand / 1000) == 2:

self.func_table[self.curr_scope].address_dir.temp.rel
easeAddress(r_type, r_operand)
                if (l_operand / 1000) == 2:

self.func_table[self.curr_scope].address_dir.temp.rel
easeAddress(l_type, l_operand)

            else:
                print(f'[Error: {line_num}] Type
mismatch')
                sys.exit()
```

VirtualMachine.py

```
class VirtualMachine:
    """
    Class simulate virtual machine, is in charge of
runtime for the language
    """
    def __init__(self, func_table, quad_list,
cte_mem, pointer_mem, global_addr_dir):
        self.func_table = func_table
        self.quad_list = quad_list
        self.cte_mem = cte_mem
        self.ctx_stack = []
        self.call_stack = []
        self.file_path = None
        self.dataframe = None

        # Init global memory
        self.global_mem = Memory(global_addr_dir)

        # Init pointer memory
        self.pointer_mem = pointer_mem

        # Init main memory and context
        local_mem =
Memory(self.func_table["main"].address_dir.local)
        temp_mem =
Memory(self.func_table["main"].address_dir.temp)
        self.ctx_stack.append(Cache(0, "main",
local_mem, temp_mem))

def resolveMem(self, address):
        # Obtains the value by searching in different
contexts
        if address >= 4000:
            real_address =
self.pointer_mem.getAddress(address)
            return self.resolveMem(real_address)
        if address >= 3000:
            return self.cte_mem.getConstant(address)
        if address >= 2000:
            return
self.ctx_stack[-1].temp_mem.getValue(address)
        if address >= 1000:
            return
self.ctx_stack[-1].local_mem.getValue(address)
        else:
            return self.global_mem.getValue(address)

def writeResult(self, address, result):
        # Writes a value by searching in different
contexts
        if address >= 4000:
            self.pointer_mem.writePointer(address,
result)
        elif address >= 2000:
```

```
self.ctx_stack[-1].temp_mem.writeAddress(address,
result)
        elif address >= 1000:

self.ctx_stack[-1].local_mem.writeAddress(address,
result)
        else:
            self.global_mem.writeAddress(address,
result)

def performArithmetic(self, quad, oper):
        # Common function for all arithmetic
operations
        try:
            result = oper(self.resolveMem(quad.op1),
self.resolveMem(quad.op2))
            self.writeResult(quad.res, result)
        except ZeroDivisionError:
            # Throw error if there is a division by
zero
            print("[Error] Division by zero")
            sys.exit()

def performPARAM(self, quad):
        # Writes a value for a parameter, given an
argument
        from_address = quad.op1

        # Out of bounds (deprecated, but won't touch)
        if from_address >= 5000:
            return

        # Get the type of the argument, param number
and param data
        from_type = getDataType(from_address)

        param_num = int(quad.res[3:])
        dest_addr, dest_type =
self.func_table[self.call_stack[-1].ctx].params[param
_num]

        # Get value of parameter
        if from_address >= 4000:
            quad = Quadruple(Operator.PARAM,
quad.res, self.pointer_mem.getAddress(from_address))
            self.performPARAM(quad)
            return
        elif from_address >= 3000:
            val =
self.cte_mem.getConstant(from_address)
        elif from_address >= 2000:
            val =
self.ctx_stack[-1].temp_mem.getValue(from_address)
        elif from_address >= 1000:
            val =
self.ctx_stack[-1].local_mem.getValue(from_address)
        else:
            val =
self.global_mem.getValue(from_address)

        # Check if it is valid
        result_type =
semantic_cube[dest_type][from_type][Operator.ASGN]

        if result_type != None:
            if result_type == Type.INT:
                val = int(val)
            elif result_type == Type.FLOAT:
                val = float(val)


self.call_stack[-1].local_mem.writeAddress(dest_addr,
val)
        else:
            print("[Error]: Argument type does not
match parameter type")
            sys.exit()

def performERA(self, quad):
        # Instantiate memory of next context and
appends it to the call stack
        next_local_mem =
Memory(self.func_table[quad.op1].address_dir.local)
        next_temp_mem =
Memory(self.func_table[quad.op1].address_dir.temp)
        next_context = quad.op1
        self.call_stack.append(Cache(None,
next_context, next_local_mem, next_temp_mem))

def performGOSUB(self, quad):
        # When returning from func, return to next
quad
        self.ctx_stack[-1].quad_pos += 1

        # Save where we're going and change context
from current one to next call
        self.call_stack[-1].quad_pos = quad.op2
        self.ctx_stack.append(self.call_stack.pop())

def performENDPROC(self):
        # Remove current context
        self.ctx_stack.pop()

def performDATA(self,quad):
        # Create a dataframe, from file path
previously defined
        self.dataframe = pd.read_csv(self.file_path)

        self.writeResult(quad.op1,
len(self.dataframe.index))
        self.writeResult(quad.op2,
len(self.dataframe.columns))

def performPLOT(self, quad):
        try:
            self.dataframe.plot(x =
self.resolveMem(quad.op1), y =
self.resolveMem(quad.op2), kind='scatter')
            plt.show()
        except TypeError:
            print("[Error] Dataframe key not found in
file.")
            sys.exit()
```