

Diseño de Compiladores

# Documentación

---

# Raoul

**Fecha:**

6 de Junio de 2022

**Maestros:**

Héctor Gibrán Ceballos Cancino

Elda Guadalupe Quiroga González

**Autores:**

RicardoGlz

Ricardo Javier González Castillo

[A01282829]

# Índice

<b>Descripción del Proyecto</b>	<b>2</b>
Propósito, Objetivos y Alcance	2
Análisis de Requerimientos	2
Tabla de requerimientos	2
Casos de uso	3
Test Cases	15
Proceso de Desarrollo	17
Proceso	17
Bitácoras	18
Reflexión	19
<b>Descripción del lenguaje</b>	<b>21</b>
Nombre del lenguaje	21
Descripción de las principales características del lenguaje	21
Listado de errores	21
<b>Descripción del compilador</b>	<b>24</b>
Equipo de cómputo, lenguaje, utilerías	24
Análisis de Léxico	24
Expresiones Regulares	24
Tokens	25
Análisis de Sintaxis	25
Generación de Código Intermedio y Semántica	28
Generación de AST	28
Inferencia de tipado	36
Casts implícitos	37
Especificaciones extras	37
Cubo Semántico	37
Administración de Memoria	38
<b>Descripción de la Máquina Virtual</b>	<b>40</b>
Equipo de cómputo, lenguaje, utilerías	40
Administración de la memoria	41
<b>Pruebas del Funcionamiento del Lenguaje</b>	<b>43</b>
<b>Código</b>	<b>56</b>

# Descripción del Proyecto

## Propósito, Objetivos y Alcance

Con el actual auge de la necesidad de hacer ciencia de datos, es necesario actualizar nuestras herramientas que sean especializadas para este tema. Siendo uno de estos el lenguaje de programación que usamos para ello. Este lenguaje debe ser fácilmente entendible para alguien que recién se introduce a programar y que además proporcione las herramientas básicas que se podría necesitar.

El objetivo de *Raoul* es proveer de un lenguaje de programación especializado en ciencia de datos, que no esté limitado en cuestión de velocidad. Por ello este será desarrollado usando *Rust* como lenguaje principal, debido a la velocidad de ejecución y eficiencia que promete.

## Análisis de Requerimientos

Tabla de requerimientos

RF01	El lenguaje deberá poder resolver operaciones aritméticas y lógicas.
RF02	El lenguaje deberá tener estatutos para introducir datos.
RF03	El lenguaje deberá poder imprimir datos primitivos.
RF04	El lenguaje deberá soportar estatutos de ciclos. En este caso un for y un while.
RF05	El lenguaje deberá soportar estatutos de control. Esto incluye los estatutos condicionales if, else if y else.
RF06	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. Dichos parámetros sólo pueden ser de tipo primitivo.
RF07	El lenguaje deberá soportar arreglos con los tipos de datos primitivos.
RF08	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: media, desviación estándar, variación y mediana
RF09	El lenguaje deberá poder poder mostrar visualizaciones con dataframes. Siendo estas visualizaciones un “scatter plot” y un “histograma”

## Casos de uso

<b>Caso de uso:</b> Realizar suma de operandos		<b>ID:</b> UC001
<b>Descripción:</b> El código obtiene la suma de dos operandos		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe un operando (variable, constante)</li><li>2. Programador escribe '+'</li><li>3. Programador escribe otro operando (variable, constante)</li><li>4. Ejecutar código</li><li>5. Máquina Virtual obtiene la suma de los dos operandos</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>• Operandos son compatibles</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>• Se regresa el valor de la suma de los operandos</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>• Se arroja error si los tipos de datos no son compatibles</li></ul>		

<b>Caso de uso:</b> Realizar resta de operandos		<b>ID:</b> UC002
<b>Descripción:</b> El código obtiene la resta de dos operandos		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe un operando (variable, constante)</li><li>2. Programador escribe '-'</li><li>3. Programador escribe otro operando (variable, constante)</li><li>4. Ejecutar código</li><li>5. Máquina Virtual obtiene la resta de los dos operandos</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>• Operandos son compatibles</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>• Se regresa el valor de la resta de los operandos</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>• Se arroja error si los tipos de datos no son compatibles</li></ul>		

<b>Caso de uso:</b> Realizar producto de operandos		<b>ID:</b> UC003
<b>Descripción:</b> El código obtiene el producto de dos operandos		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe un operando (variable, constante)</li><li>2. Programador escribe '*'</li><li>3. Programador escribe otro operando (variable, constante)</li><li>4. Ejecutar código</li><li>5. Máquina Virtual obtiene el producto de los dos operandos</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>• Operandos son compatibles</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>• Se regresa el valor del producto de los operandos</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>• Se arroja error si los tipos de datos no son compatibles</li></ul>		

<b>Caso de uso:</b> Realizar división de operandos		<b>ID:</b> UC004
<b>Descripción:</b> El código obtiene la división de dos operandos		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe un operando (variable, constante)</li><li>2. Programador escribe '/'</li><li>3. Programador escribe otro operando (variable, constante)</li><li>4. Ejecutar código</li><li>5. Máquina Virtual obtiene el producto de los dos operandos</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>• Operandos son compatibles</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>• Se regresa el valor de la división de los operandos</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>• Se arroja error si los tipos de datos no son compatibles</li><li>• Si el denominador es cero, también se arroja error. Pero este durante ejecución.</li></ul>		

<b>Caso de uso:</b> Comparar igualdad de operandos	<b>ID:</b> UC005
--	------------------

<b>Descripción:</b> El código determina si dos operandos son iguales en valor	
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"> <li>1. Programador escribe un operando (variable, constante)</li> <li>2. Programador escribe '=='</li> <li>3. Programador escribe otro operando (variable, constante)</li> <li>4. Ejecutar código</li> <li>5. Máquina Virtual determina si los operandos son equivalentes</li> </ol>
<b>Condición de entrada:</b>	<ul style="list-style-type: none"> <li>• Operandos son compatibles</li> </ul>
<b>Condición de salida:</b>	<ul style="list-style-type: none"> <li>• Se regresa si los operandos son equivalentes</li> </ul>
<b>Flujo alternativo:</b> <ul style="list-style-type: none"> <li>• Se arroja error si los tipos de datos no son compatibles</li> </ul>	

<b>Caso de uso:</b> Comparar desigualdad de operandos		<b>ID:</b> UC006
<b>Descripción:</b> El código determina si dos operandos no son iguales en valor		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe un operando (variable, constante)</li><li>2. Programador escribe '!='</li><li>3. Programador escribe otro operando (variable, constante)</li><li>4. Ejecutar código</li><li>5. Máquina Virtual determina si los operandos no son equivalentes</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>● Operandos son compatibles</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>● Se regresa si los operandos no son equivalentes</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>● Se arroja error si los tipos de datos no son compatibles</li></ul>		

<b>Caso de uso:</b> Realizar NOT lógico		<b>ID:</b> UC007
<b>Descripción:</b> El código obtiene el opuesto lógico de un operando entero		

<b>Flujo de eventos:</b>	<ol style="list-style-type: none"> <li>1. Programador escribe 'NOT'</li> <li>2. Programador escribe un operando (variable, constante)</li> <li>3. Ejecutar código</li> <li>4. Máquina Virtual obtiene el opuesto lógico</li> </ol>
<b>Condición de entrada:</b>	<ul style="list-style-type: none"> <li>• El operando es de tipo entero</li> </ul>
<b>Condición de salida:</b>	<ul style="list-style-type: none"> <li>• Se obtiene opuesto lógico del operando</li> </ul>
<b>Flujo alternativo:</b> <ul style="list-style-type: none"> <li>• Se arroja error si el operando no es compatible</li> </ul>	

<b>Caso de uso:</b> Realizar menor o igual que		<b>ID:</b> UC008
<b>Descripción:</b> El código determina si un operando es menor o igual a otro		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe un operando (variable, constante)</li><li>2. Programador escribe '&lt;='</li><li>3. Programador escribe otro operando (variable, constante)</li><li>4. Ejecutar código</li><li>5. Máquina Virtual determina si primer operando es menor o igual que el segundo operando</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>● Operandos son compatibles</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>● Se obtiene determina si primer operando es menor o igual que el segundo operando</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>● Se arroja error si los tipos de datos no son compatibles</li></ul>		

<b>Caso de uso:</b> Realizar mayor o igual que		<b>ID:</b> UC009
<b>Descripción:</b> El código determina si un operando es mayor o igual a otro		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"> <li>1. Programador escribe un operando (variable, constante)</li> <li>2. Programador escribe '&gt;='</li> <li>3. Programador escribe otro operando (variable, constante)</li> </ol>	

	<ol style="list-style-type: none"> <li>Ejecutar código</li> <li>Máquina Virtual determina si primer operando es mayor o igual que el segundo operando</li> </ol>
<b>Condición de entrada:</b>	<ul style="list-style-type: none"> <li>Operandos son compatibles</li> </ul>
<b>Condición de salida:</b>	<ul style="list-style-type: none"> <li>Se obtiene determina si primer operando es mayor o igual que el segundo operando</li> </ul>
<b>Flujo alternativo:</b> <ul style="list-style-type: none"> <li>Se arroja error si los tipos de datos no son compatibles</li> </ul>	

<b>Caso de uso:</b> Realizar menor que		<b>ID:</b> UC010
<b>Descripción:</b> El código determina si un operando es menor que otro		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe un operando (variable, constante)</li><li>2. Programador escribe '&lt;'</li><li>3. Programador escribe otro operando (variable, constante)</li><li>4. Ejecutar código</li><li>5. Máquina Virtual determina si primer operando menor que el segundo operando</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>• Operandos son compatibles</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>• Se obtiene determina si primer operando menor que el segundo operando</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>• Se arroja error si los tipos de datos no son compatibles</li></ul>		

<b>Caso de uso:</b> Realizar mayor que		<b>ID:</b> UC011
<b>Descripción:</b> El código determina si un operando es mayor que otro		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"> <li>Programador escribe un operando (variable, constante)</li> <li>Programador escribe '&gt;'</li> <li>Programador escribe otro operando (variable, constante)</li> <li>Ejecutar código</li> </ol>	



	5. Máquina Virtual determina si primer operando mayor que el segundo operando
<b>Condición de entrada:</b>	<ul style="list-style-type: none"> <li>Operandos son compatibles</li> </ul>
<b>Condición de salida:</b>	<ul style="list-style-type: none"> <li>Se obtiene determina si primer operando mayor que el segundo operando</li> </ul>
<b>Flujo alternativo:</b> <ul style="list-style-type: none"> <li>Se arroja error si los tipos de datos no son compatibles</li> </ul>	

<b>Caso de uso:</b> Realizar OR lógico		<b>ID:</b> UC012
<b>Descripción:</b> El código realiza un OR lógico entre dos operandos		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe un operando (variable, constante)</li><li>2. Programador escribe 'OR'</li><li>3. Programador escribe otro operando (variable, constante)</li><li>4. Ejecutar código</li><li>5. Máquina Virtual realiza OR lógico entre los operandos</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>● Operandos son compatibles</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>● Se obtiene el resultado del OR lógico</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>● Se arroja error si los tipos de datos no son compatibles</li></ul>		

<b>Caso de uso:</b> Realizar AND lógico		<b>ID:</b> UC013
<b>Descripción:</b> El código realiza un AND lógico entre dos operandos		
<b>Flujo de eventos:</b>	1. Programador escribe un operando (variable, constante) 2. Programador escribe 'AND' 3. Programador escribe otro operando (variable, constante) 4. Ejecutar código 5. Máquina Virtual realiza AND lógico entre los operandos	

<b>Condición de entrada:</b>	<ul style="list-style-type: none"> <li>• Operandos son compatibles</li> </ul>
<b>Condición de salida:</b>	<ul style="list-style-type: none"> <li>• Se obtiene el resultado del AND lógico</li> </ul>
<b>Flujo alternativo:</b> <ul style="list-style-type: none"> <li>• Se arroja error si los tipos de datos no son compatibles</li> </ul>	

<b>Caso de uso:</b> Realizar asignación		<b>ID:</b> UC014
<b>Descripción:</b> El código realiza una asignación a una variable		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe una variable</li><li>2. Programador escribe '='</li><li>3. Programador escribe una expresión</li><li>4. Ejecutar código</li><li>5. Máquina Virtual asigna el valor de la expresión a la variable</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>● Operandos son compatibles</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>● El valor de la variable es el resultado de la expresión</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>● Se arroja error si los tipos de datos no son compatibles</li><li>● La variable fue anteriormente asignada a otro tipo no compatible con el de la expresión</li></ul>		

<b>Caso de uso:</b> Realizar asignación global		<b>ID:</b> UC014
<b>Descripción:</b> El código realiza una asignación a una variable		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"> <li>1. Programador escribe 'global'</li> <li>2. Programador escribe una variable</li> <li>3. Programador escribe '='</li> <li>4. Programador escribe una expresión</li> <li>5. Ejecutar código</li> <li>6. Máquina Virtual asigna el valor de la expresión a la variable</li> </ol>	
<b>Condición de</b>	<ul style="list-style-type: none"> <li>• Operandos son compatibles</li> </ul>	

<b>entrada:</b>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"> <li>El valor de la variable es el resultado de la expresión</li> </ul>
<b>Flujo alternativo:</b> <ul style="list-style-type: none"> <li>Se arroja error si los tipos de datos no son compatibles</li> <li>La variable fue anteriormente asignada a otro tipo no compatible con el de la expresión</li> </ul>	

<b>Caso de uso:</b> Realizar condicional		<b>ID:</b> UC015
<b>Descripción:</b> El código realiza control de flujo mediante una condicional		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe 'if'</li><li>2. Programador escribe una expresión condicional entre paréntesis</li><li>3. Programador escribe un bloque de código</li><li>4. Ejecutar código</li><li>5. Máquina Virtual evalúa expresión</li><li>6. Máquina Virtual decide si se ejecuta bloque de código</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>• Expresión dentro de condicional es de tipo entera</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>• Máquina virtual ejecuta los cuádruplos según la evaluación de la expresión en la condicional.</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>• Se agrega un bloque "else if", que realiza la misma lógica que con el bloque "if" en caso de que la evaluación del "if" haya sido falsa.</li><li>• Se agrega un bloque else, que contiene código a ejecutar en caso de que la expresión no evalúa a verdadero (0 o falso)</li><li>• Se arroja error cuando expresión no es de tipo lógico (entero o booleano)</li><li>• Se arroja un error si se regresa en el bloque del if y no se agrega otro en el else o en el resto de la función</li></ul>		

<b>Caso de uso:</b> Realizar ciclo delimitado		<b>ID:</b> UC016
<b>Descripción:</b> El código repite código dentro del ciclo de acuerdo a límites		
<b>Flujo de</b>	<ol style="list-style-type: none"> <li>Programador escribe 'for'</li> </ol>	

<b>eventos:</b>	<ol style="list-style-type: none"> <li>Programador inicializa una variable previamente definida</li> <li>Programador escribe 'to'</li> <li>Programador define límite superior inclusivo como expresión</li> <li>Programador escribe un bloque de código</li> <li>Programador ejecuta código</li> <li>Máquina virtual inicializa la variable</li> <li>Máquina virtual evalúa condición de límite superior inclusivo</li> <li>En éxito, máquina virtual ejecuta código</li> <li>Máquina virtual incrementa iterador</li> <li>Máquina virtual regresa al paso 8</li> </ol>
<b>Condición de entrada:</b>	<ul style="list-style-type: none"> <li>Iterador debe ser numérico</li> <li>Expresión límite debe ser numérica</li> </ul>
<b>Condición de salida:</b>	<ul style="list-style-type: none"> <li>Máquina virtual ejecuta los cuádruplos según la evaluación de la expresión en la condicional.</li> </ul>
<b>Flujo alternativo:</b> <ul style="list-style-type: none"> <li>Se arroja error cuando no se cumplen las condiciones de tipo de dato</li> <li>Se arroja error cuando se usa el statement return, y no se agrega otro en otra parte de la función</li> </ul>	

<b>Caso de uso:</b> Realizar ciclo condicional		<b>ID:</b> UC017
<b>Descripción:</b> El código repite código dentro del ciclo de acuerdo a la condición		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"> <li>Programador escribe 'while'</li> <li>Programador escribe una expresión entre paréntesis</li> <li>Programador escribe un bloque de código</li> <li>Programador ejecuta código</li> <li>Máquina virtual evalúa condición</li> <li>En éxito, máquina virtual ejecuta código</li> <li>Máquina virtual regresa al paso 5</li> </ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"> <li>Expresión en la condición debe ser tipo entera</li> </ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"> <li>Máquina virtual ejecuta los cuádruplos según la evaluación de la expresión en la condicional.</li> </ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"> <li>Se arroja error cuando no se cumplen las condiciones de tipo de dato</li> <li>Se arroja error cuando se usa el statement return, y no se agrega otro en otra parte de la función</li> </ul>		

<b>Caso de uso:</b> Declaración de una función		<b>ID:</b> UC018
<b>Descripción:</b> El código genera los cuádruplos para una subrutina del código		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe 'func'</li><li>2. Programador escribe tipo de retorno o void</li><li>3. Programador escribe nombre de la función</li><li>4. Programador escribe parámetros entre paréntesis</li><li>5. Programador escribe bloque de código</li><li>6. Programador ejecuta código</li><li>7. Se generan los cuádruplos necesarios</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>• Nombre de la función es único</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>• Compilador genera los cuádruplos para la función</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>• Se arroja error cuando el nombre ya existe (función redefinida)</li></ul>		

<b>Caso de uso:</b> Llamada de una función		<b>ID:</b> UC019
<b>Descripción:</b> El código hace una llamada a una subrutina		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe el nombre de la subrutina</li><li>2. Programador escribe los argumentos a enviar entre paréntesis</li><li>3. Programador ejecuta código</li><li>4. Máquina virtual ejecuta método</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>• Argumentos coinciden en número y tipo con parámetros</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>• Máquina virtual ejecuta la función</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>• Se arroja error cuando existe algún problema con los argumentos</li></ul>		

<b>Caso de uso:</b> Realizar escritura		<b>ID:</b> UC020
<b>Descripción:</b> El código imprime los contenidos del llamado		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe 'print'</li><li>2. Programador escribe expresiones o letreros</li><li>3. Programador ejecuta el código</li><li>4. Máquina virtual imprime los contenidos del llamado</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>• N/A</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>• Máquina virtual escribe los contenidos del llamado</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>• Máquina virtual manda error si el valor no ha sido inicializado</li></ul>		

<b>Caso de uso:</b> Realizar lectura		<b>ID:</b> UC021
<b>Descripción:</b> El código pide una entrada al usuario y la escribe en alguna variable		
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"><li>1. Programador escribe 'input'</li><li>2. Programador escribe variables</li><li>3. Programador ejecuta el código</li><li>4. Máquina virtual pide entrada a usuario</li><li>5. Programador escribe algún valor</li><li>6. Máquina virtual asigna ese valor a la variable</li></ol>	
<b>Condición de entrada:</b>	<ul style="list-style-type: none"><li>• Tipos de datos son compatibles</li></ul>	
<b>Condición de salida:</b>	<ul style="list-style-type: none"><li>• Entrada de usuario se le asigna a una variable</li></ul>	
<b>Flujo alternativo:</b> <ul style="list-style-type: none"><li>• Si los tipos de datos no son compatibles arroja error</li></ul>		

<b>Caso de uso:</b> Operaciones Data Science		<b>ID:</b> UC022
<b>Descripción:</b>		

El código hace operaciones en un dataframe	
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"> <li>1. Programador abre un archivo con read_csv()</li> <li>2. Programador usa algún método de Data Science (ex. average)</li> <li>3. Programador ejecuta código</li> <li>4. Máquina virtual carga archivo y datos</li> <li>5. Máquina virtual ejecuta método Data Science</li> </ol>
<b>Condición de entrada:</b>	<ul style="list-style-type: none"> <li>• Archivo existe y se puede abrir</li> <li>• Parámetros para llamados coinciden con su definición</li> </ul>
<b>Condición de salida:</b>	<ul style="list-style-type: none"> <li>• Máquina virtual realiza operaciones con el dataframe</li> </ul>
<b>Flujo alternativo:</b> <ul style="list-style-type: none"> <li>• Si el archivo no existe el lenguaje arroja un error</li> <li>• Si no se mandan los parámetros de manera correcta, se arroja error</li> </ul>	

<b>Caso de uso:</b> Operaciones Visualizaciones	<b>ID:</b> UC022
<b>Descripción:</b> El código hace operaciones de visualización en un dataframe	
<b>Flujo de eventos:</b>	<ol style="list-style-type: none"> <li>1. Programador abre un archivo con read_csv()</li> <li>2. Programador usa algún método de visualización (ex. plot)</li> <li>3. Programador ejecuta código</li> <li>4. Máquina virtual carga archivo y datos</li> <li>5. Máquina virtual ejecuta método de visualización</li> </ol>
<b>Condición de entrada:</b>	<ul style="list-style-type: none"> <li>• Archivo existe y se puede abrir</li> <li>• Parámetros para llamados coinciden con su definición</li> </ul>
<b>Condición de salida:</b>	<ul style="list-style-type: none"> <li>• Máquina virtual muestra visualizaciones del dataframe</li> </ul>
<b>Flujo alternativo:</b> <ul style="list-style-type: none"> <li>• Si el archivo no existe el lenguaje arroja un error</li> <li>• Si no se mandan los parámetros de manera correcta, se arroja error</li> </ul>	

## Test Cases

<b>Nombre:</b> Expresiones lineales		<b>ID:</b> CP01
<b>Objetivo:</b> Realizar operaciones aritméticas, lógicas y relacionales.		
<b>Entradas:</b> <pre>func main(): void {     a = true OR false;     b = 1 AND true;     c = 1 &lt;= 2;     d = "1" &lt; 2;     e = 2 &gt;= 1;     f = 2 &gt; 1;     e = 3 == 3;     f = 3 != 2;     g = "2.1" + 2.0;     h = 2 - 2.0;     i = 4 * 4;     j = 4 / 4;     k = NOT false;     print(a, b, c, d, e, f, g, h, i, j, k); }</pre>		<b>Salida:</b> <pre>true true true true true true 4.1 0 16 1 true</pre>
<b>Resultado de prueba:</b> Exitosa		

<b>Nombre:</b> Expresiones no lineales		<b>ID:</b> CP02
<b>Objetivo:</b> Realizar condicionales y ciclos		
<b>Entradas:</b> <pre>func main(): void {     d = 10;     a = 1;     while (a &lt; d) {         print(a);         a = a + 1;     }     for (i = 0 to a) {         print(i);     }     if (i &gt;= a) {         c = 9001;     }     if (c &gt;= 9001) {         print("It's over 9000!");     } else if (c &gt;= 1000) {         print("It's over 1000!");     } else {</pre>		<b>Salida:</b> <pre>1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8</pre>

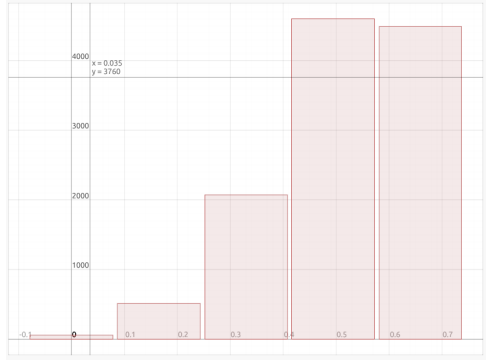


<pre>         print("It's something!");     } }</pre>	9 10 It's over 9000!
<b>Resultado de prueba:</b> Exitosa	

<b>Nombre:</b> Modulos		<b>ID:</b> CP03
<b>Objetivo:</b> Utilización de código modular		
<b>Entradas:</b> <pre> func printer(text: float): void {     print(text); }  func func1(bar: float): float{     j = bar * 100;     printer(j);      return(j+1); }  func main(): void{     foo = 100;     baz = func1(foo);     printer(baz); }</pre>		<b>Salida:</b> 10000 10001
<b>Resultado de prueba:</b> Exitosa		

<b>Nombre:</b> Dataframes		<b>ID:</b> CP04
<b>Objetivo:</b> Utilización de <i>dataframes</i> para cálculos estadísticos		
<b>Entradas:</b> <pre> func main(): void {     dataframe = read_csv("song_data_clean.csv");     avg = average(dataframe, "song_duration_ms");     col = "danceability";     danceability_std = std(dataframe, col);     danceability_mode = median(dataframe, col);     danceability_var = variance(dataframe, col);     corr = correlation(dataframe, col, "song_duration_ms");</pre>		<b>Salida:</b> 218645.22944916878 0.15916777771148127 0.637 0.025334381461611516 -0.08547008157088816

<pre>print(avg, danceability_std, danceability_mode, danceability_var, corr); }</pre>	
<b>Resultado de prueba:</b> Exitosa	

<b>Nombre:</b> Visualización		<b>ID:</b> CP05
<b>Objetivo:</b> Utilización de <i>dataframes</i> para visualizaciones		
<b>Entradas:</b> <pre>func main(): void {     dataframe = read_csv("song_data_clean.csv");     histogram(dataframe, "danceability", 5); }</pre>		<b>Salida:</b> 
<b>Resultado de prueba:</b> Exitosa		

## Proceso de Desarrollo

### Proceso

Para el desarrollo del compilador se realizó un proceso de TDD, en el que se agregaban diferentes ejemplos al folder de “examples” en la carpeta del proyecto. Y cada uno de estos ejemplos como los inputs para probar los que ya se sabían que eran ejemplos válidos/correctos y los que no. Para los que no lo eran también se dividían en aquellos ejemplos que arrojaba un error desde el análisis sintáctico/léxico, en el análisis semántico estático o en el análisis semántico dinámico. Al mismo tiempo se configuró el repositorio de Github para que alertara cuando ya no pasaban todas las pruebas.

## Bitácoras

08/04/2022	<p><b>Avance:</b> Léxico y Sintaxis</p> <p><b>Progreso:</b> Se genera analizador léxico y sintáctico, usando pest.rs en Rust. Compilador marca cuando el código se pudo compilar de manera exitosa y cuando hay error de sintaxis marca el token recibido y los posibles siguientes.</p> <p><b>MRs:</b> <a href="#">[#1]</a> y <a href="#">[#2]</a></p> <p><b>Release:</b> <a href="#">[#1]</a></p>
10/04/2022	<p><b>Avance:</b> DirFunciones y Tablas de Variables</p> <p><b>Progreso:</b> Se implementó la estructura base para el AST y la clase que ayuda a generarlo. 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. Y se empezaron a añadir pruebas.</p> <p><b>MRs:</b> <a href="#">[#5]</a></p> <p><b>Release:</b> <a href="#">[#2]</a></p>
18/04/2022	<p><b>Avance:</b> Cubo Semántico y Cuádruplos para Expresiones</p> <p><b>Progreso:</b> Se crearon manejadores para variables de los 4 “scopes”: constantes, temporales, locales y globales. Se crearon los cuádruplos y los nodos del AST para operaciones lineales. Se agregaron más ejemplos inválidos</p> <p><b>MRs:</b> <a href="#">[#10]</a></p> <p><b>Release:</b> <a href="#">[#3]</a></p>
25/04/2022	<p><b>Avance:</b> Cuádruplos para Estatutos No-Lineales</p> <p><b>Progreso:</b> Se crea el AST para los estatutos no lineales. Se generan instrucciones para if/else if/else. Al mismo tiempo se agrega para las instrucciones de ciclo: for y while. Se creó una función para probar que la compilación falla en la evaluación semántica</p> <p><b>MRs:</b> <a href="#">[#17]</a></p> <p><b>Release:</b> <a href="#">[#4]</a></p>
26/04/2022	<p><b>Avance:</b> Funciones</p> <p><b>Progreso:</b> Se generan los cuádruplos y nodos del AST necesarios para las operaciones de módulos. Los manejadores de funciones y las funciones tal cual calculan el tamaño que ocupan. Se agregaron más casos de prueba. Se empieza a registrar el primer quad de la función en la lista de quads.</p> <p><b>MRs:</b> <a href="#">[#20]</a></p>

	<b>Release:</b> <a href="#">[#5]</a>
28/04/2022	<p><b>Avance:</b> Máquina Virtual - parte 1</p> <p><b>Progreso:</b> Se creó una clase para Memoria virtual que toma un manejador de direcciones y crea slots vacíos por cada una de las direcciones manejadas por la memoria, la cual maneja su lectura y escritura. También se crea una clase para la Máquina Virtual la cual ya puede ejecutar estatutos lineales.</p> <p><b>MRs:</b> <a href="#">[#21]</a></p> <p><b>Release:</b> <a href="#">[#6]</a></p>
04/05/2022	<p><b>Avance:</b> Traducción de Arreglos</p> <p><b>Progreso:</b> Se creo un manejador de punteros para el manejo de arreglos. A la estructura de Variables se le añadió un atributo de dimensiones, para determinar si es una lista o matriz. Se modificó que los manejadores de direcciones, tomasen en cuenta las dimensiones de la variable que está solicitando dirección/direcciones. También se agregaron los nodos, cuádruplos relacionados con los arreglos. Además la VM ya puede realizar todas las operaciones posibles (lineales, no lineales, módulos y con arreglos). Se agregaron pruebas para garantizar que el AST, los cuádruplos, los mensajes de la VM y cualquier error que hubiese, fuesen los correctos.</p> <p><b>MRs:</b> <a href="#">[#22]</a></p> <p><b>Release:</b> <a href="#">[#7]</a></p>
31/05/2022	<p><b>Avance:</b> Último avance</p> <p><b>Progreso:</b> Se agregaron funciones especializadas para Data Science, que incluyen operaciones estadísticas y visualizaciones sobre un dataframe, generado de un archivo csv. Además, se refinan los mensajes de error y corrigen los últimos detalles del lenguaje.</p> <p><b>MRs:</b> <a href="#">[#24]</a></p> <p><b>Release:</b> <a href="#">[#8]</a></p>

## Reflexión

### Ricardo Javier González Castillo

Si algo aprendí de este proyecto es que debería dejar de complicarme la vida. Empecé complicandome la vida cuando escogí Rust como lenguaje, después me compliqué aún más la vida cuando decidí que mis variables se iba a declarar y asignar al mismo tiempo y por último me compliqué aún más la vida cuando después de averiguar que no habían muchas librerías *estables* para visualización para Rust aún así dije: “No hay problema, sí se arma hacerlo de Data Science con visualización”. Sin embargo, creo que definitivamente aprendí mucho más de lo que hubiese hecho de otra forma. No sólo del lenguaje que escogí, con el cual nunca había

trabajado y ahora tengo una relación amor-odio, sino con todo lo que involucra un lenguaje. Lo que más me acuerdo por el momento, fue el gran reto de crear una forma de asignar arreglos "constantes", los cuales no parecieran la gran cosa, pero una vez que empiezas a hacerlo sí que se empieza a complicar todo.

Ricardo Glz

# Descripción del lenguaje

## Nombre del lenguaje

El nombre del lenguaje descrito a lo largo del presente texto es *Raoul*

## 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 y booleana. Por eso mismo, el operador *NOT* sólo funciona para valores booleanos y enteros. 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*, *bool* 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. Además de permitir visualizaciones

## Listado de errores

### Ejecución:

Mensaje	Descripción
[Error] Stack Overflow!	Este error se suscita cuando se acaba la memoria del programa, determinado por una cantidad de 1024 variables locales y temporales de cualquier tipo.
[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] Found initialized value	Este error se suscita cuando se intenta acceder a un espacio de memoria que existe, pero cuyo valor no ha sido inicializado
[Error] File is not a valid CSV	Este error se suscita cuando el tipo de archivo utilizado para un dataframe es de tipo diferente a .csv.
[Error] No data frame was created. You need to create one using `read_csv` before using an operation	Este error pasa cuando se intenta usar alguna de las operaciones para dataframes, antes de inicializar el dataframe
[Error] Could not read the file	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.
[Error]: The amount of bins should be positive	Este error se suscita cuando se hace la operación `histogram` y el valor de `bins` es 0 negativa

### Compilación:

**Nota:** Para estos tipos de error además del mensaje de error a continuación se muestra el fragmento código que causó el error.

Mensaje	Descripción
Memory was exceeded	Este error se suscita cuando el programador crea más de 250 variables de un tipo ya sea en scope local, temporal o global o constante.
Variable "name" was not declared	Este error se suscita cuando el programador intenta utilizar una variable "name" cuya declaración no fue hecha previamente.
Function "name" was not declared or does not return a non-void value"	Este error se suscita cuando el programador intenta hacer uso de una función "name" que no fue declarada previamente. Este mensaje se muestra específicamente cuando se usa la llamada como parte de una expresión
Function "name" was not declared	Este error se suscita cuando el programador intenta hacer uso de una función "name" que no fue declarada previamente. Este mensaje se muestra específicamente cuando se usa la llamada

	como un estatuto.
Function "name" was already declared before	Este error se suscita cuando el programador nombra a dos funciones con el identificador "name", redefiniendo así una función previamente declarada. Y eso no se puede porque aquí somos de buenas familias
"name" was originally defined as A and you're attempting to redefined it as a B	Este error se suscita cuando el programador intenta en su programa asignarle a una variable un valor de tipo B, a una que ya se había definido como tipo A. Y que además, no permitimos el "cast" implícito de B a A.
Cannot cast from A to B	Este error se suscita cuando el programador intenta realizar una operación o dar un argumento a una función que espera un tipo B usando un valor de tipo A, pero no es posible hacer un "cast" implícito de A a B.
Wrong args amount: Expected {expected}, but were given {given}	Este error se suscita cuando el programador manda un total de <i>given</i> argumentos, pero la función espera una cantidad <i>expected</i>
In function "name" not all branches return a value	Este error se suscita cuando el programador crea una función "name" que debería regresar un valor. Pero no se regresa ningún valor o en algunos casos puede que no regresa. Ex. Cuando sólo se regresa el valor en un if, pero no en un o else o en el resto de la función
`name` is not a list	Este error se suscita cuando el programador intenta usar la variable "name" como si fuese una lista, cuando no lo es.
`name` is not a matrix	Este error se suscita cuando el programador intenta usar la variable "name" como si fuese una matriz o arreglo 2D, cuando no lo es.
Expecting matrix with second dimension being <i>expected</i> but received <i>given</i>	Este error se suscita cuando el programador intenta crear una matriz con valores constantes, pero las listas adentro de esta no son del mismo tamaño. Ex: a = [[1], [1, 2]];
Only one dataframe is allowed per program	Este error se suscita cuando el programador intenta declarar más de un dataframe.



# Descripción del compilador

Equipo de cómputo, lenguaje, utilerías

	Librerías utilizadas	Lenguaje
<b>address</b>	- std	Rust (1.6.0)
<b>ast</b>	- std - pest	Rust (1.6.0)
<b>enums</b>	- std	Rust (1.6.0)
<b>error</b>	- std - pest	Rust (1.6.0)
<b>parser</b>	- std - pest - pest_consume	Rust (1.6.0)
<b>quadruple</b>	- std	Rust (1.6.0)

Equipo de Computo	
Component	Equipo 1
<b>OS</b>	MacOS 10.15.7 Catalina
<b>CPU</b>	2.5 GHz Dual-Core Intel Core i5
<b>Memoria</b>	8 GB 1600 MHz DDR3
<b>Gráficos</b>	Intel HD Graphics 4000 1536 MB

## Análisis de Léxico

Expresiones Regulares

<b>INT_CTE</b>	-?[0-9]+
<b>FLOAT_CTE</b>	-?[0-9]+\.[0-9]+

<b>STRING_CTE</b>	<code>\"[^"]*"   \'[^\']*\'</code>
<b>BOOL_CTE</b>	<code>true   false</code>
<b>ID</b>	<code>[a-zA-Z][a-zA-Z0-9_]*</code>

## Tokens

<b>SUM</b>	<code>+</code>	<b>LTE</b>	<code>&lt;=</code>	<b>CHAR</b>	<code>char</code>	<b>global</b>	<code>global</code>
<b>MINUS</b>	<code>-</code>	<b>GTE</b>	<code>&gt;=</code>	<b>STRING</b>	<code>string</code>	<b>UNDERSCORE</b>	<code>_</code>
<b>TIMES</b>	<code>*</code>	<b>LT</b>	<code>&lt;</code>	<b>FUNC</b>	<code>func</code>	<b>MIN</b>	<code>min</code>
<b>DIV</b>	<code>/</code>	<b>GT</b>	<code>&gt;</code>	<b>PRINT</b>	<code>print</code>	<b>MAX</b>	<code>max</code>
<b>SEMI_COLON</b>	<code>;</code>	<b>OR</b>	<code>OR</code>	<b>MAIN</b>	<code>main</code>	<b>RANGE</b>	<code>range</code>
<b>COLON</b>	<code>:</code>	<b>AND</b>	<code>AND</code>	<b>RETURN</b>	<code>return</code>	<b>GET_ROWS</b>	<code>get_rows</code>
<b>COMMA</b>	<code>,</code>	<b>ASGN</b>	<code>=</code>	<b>INPUT</b>	<code>input</code>	<b>GET_COLUMNS</b>	<code>get_columns</code>
<b>L_BRACKET</b>	<code>{</code>	<b>IF</b>	<code>if</code>	<b>LOAD_FILE</b>	<code>read_csv</code>	<b>BOOL_CTE</b>	<code>regex</code>
<b>R_BRACKET</b>	<code>}</code>	<b>WHILE</b>	<code>while</code>	<b>AVERAGE</b>	<code>average</code>	<b>INT_CTE</b>	<code>regex</code>
<b>L_PAREN</b>	<code>(</code>	<b>FOR</b>	<code>for</code>	<b>MEDIAN</b>	<code>median</code>	<b>FLOAT_CTE</b>	<code>regex</code>
<b>R_PAREN</b>	<code>)</code>	<b>TO</b>	<code>to</code>	<b>VARIANCE</b>	<code>variance</code>	<b>STRING_CTE</b>	<code>regex</code>
<b>L_SQUARE</b>	<code>[</code>	<b>ELSE</b>	<code>else</code>	<b>STD</b>	<code>std</code>	<b>ID</b>	<code>regex</code>
<b>R_SQUARE</b>	<code>]</code>	<b>INT</b>	<code>int</code>	<b>CORREL</b>	<code>correlation</code>		
<b>EQ</b>	<code>==</code>	<b>FLOAT</b>	<code>float</code>	<b>PLOT</b>	<code>plot</code>		
<b>NE</b>	<code>!=</code>	<b>STRING</b>	<code>string</code>	<b>HISTOGRAM</b>	<code>histogram</code>		
<b>NOT</b>	<code>NOT</code>	<b>VOID</b>	<code>void</code>	<b>DECLARE_KEY</b>	<code>declare_arr</code>		

## Análisis de Sintaxis

```

ATOM_CTE  -> <bool_cte> | <float_cte> | <int_cte> | <STRING_CTE>
arr_index -> [ expr ]
arr_val   -> <id> arr_index arr_val'
arr_val'  -> arr_index | empty
NON_CTE   -> dataframe_value_ops | func_call | arr_val | <id>
VAR_VAL   -> ATOM_CTE | NON_CTE

```

```

comp_op -> == | !=
rel_op  -> < | > | <= | >=

```

```

art_op -> + | -
fact_op -> * | /

expr          -> and_term expr'
expr'         -> OR and_term expr' | empty
and_term      -> comp_term and_term'
and_term'     -> AND comp_term and_term' | empty
comp_term     -> rel_term comp_term'
comp_term'    -> comp_op rel_term comp_term' | empty
rel_term      -> art_term rel_term'
rel_term'     -> rel_op art_term rel_term' | empty
art_term      -> fact_term art_term'
art_term'     -> art_op fact_term art_term' | empty
fact_term     -> operand fact_term'
fact_term'    -> fact_op operand fact_term' | empty
operand       -> NOT operand_value | operand_value
operand_value -> VAR_VAL | ( expr )
exprs         -> expr exprs'
exprs'        -> , expr exprs' | empty

atomic_types  -> bool | float | int | string
types         -> atomic_types | void

read -> input ( )

declare_arr_type -> < atomic_types >
declare_arr      -> declare_arr declare_arr_type ( int_cte
                    declare_arr' )
declare_arr'     -> , int_cte | empty

list_cte -> [ exprs ]
mat_cte  -> [ list_cte mat_cte' ]
mat_cte' -> , list_cte mat_cte' | empty
arr_cte  -> list_cte | mat_cte

assignment_exp -> read | read_csv | expr | declare_arr | arr_cte
assignee       -> arr_val | <id>
assignment_base -> assignee = assignment_exp
assignment     -> assignment' assignment_base
assignment'    -> global | empty
global_assignment -> assignment_base ;

block          -> { block' }

```

```

block'                -> statement block' | empty
block_or_statement -> block | inline_statement

func_arg              -> <id> : atomic_types
func_args             -> func_arg func_args'
func_args'           -> , func_arg func_args' | empty
FUNC_HEADER           -> func <id> FUNC_HEADER' : types
FUNC_HEADER'         -> func_args | empty
function              -> FUNC_HEADER block
MAIN_FUNCTION         -> func main ( ) : void block
func_call             -> <id> func_call'
func_call'           -> exprs | empty

COND_EXPR             -> ( expr )
if_block              -> if COND_EXPR block_or_statement
else_block            -> else else_block'
else_block'          -> block_or_statement | decision
decision              -> if_block decision'
decision'            -> else_block | empty

write -> PRINT ( write' )
write' -> exprs | empty

while_loop -> while COND_EXPR block_or_statement

for_loop -> for ( assignment to expr ) block_or_statement

possible_str          -> <STRING_CTE> | NON_CTE
read_csv              -> READ_CSV_KEY ( possible_str )
pure_dataframe_key    -> get_rows | get_columns
pure_dataframe_op     -> pure_dataframe_key ( <id> )
unary_dataframe_key   -> average | std | median | variance | min |
                        max | range
unary_dataframe_op    -> unary_dataframe_key ( <id> , possible_str )
TWO_COLUMNS_FUNC      -> ( <id> , possible_str , possible_str )
CORRELATION           -> correlation TWO_COLUMNS_FUNC
dataframe_value_ops   -> pure_dataframe_op | unary_dataframe_op |
                        CORRELATION
plot                  -> PLOT_KEY TWO_COLUMNS_FUNC
histogram             -> HISTOGRAM_KEY ( <id> , possible_str , expr )
DATAFRAME_VOID_OPS    -> plot | histogram

return_statement -> RETURN_KEY expr

```

```

BLOCK_STATEMENT -> decision | while_loop | for_loop
INLINE_STATEMENT -> DATAFRAME_VOID_OPS | assignment | write |
return_statement | func_call
inline_statement' -> INLINE_STATEMENT ;
statement        -> inline_statement | BLOCK_STATEMENT

global_assignments -> global_assignment global_assignments | empty
program            -> SOI global_assignments program' MAIN_FUNCTION
EOI
program' -> function program' | empty

```

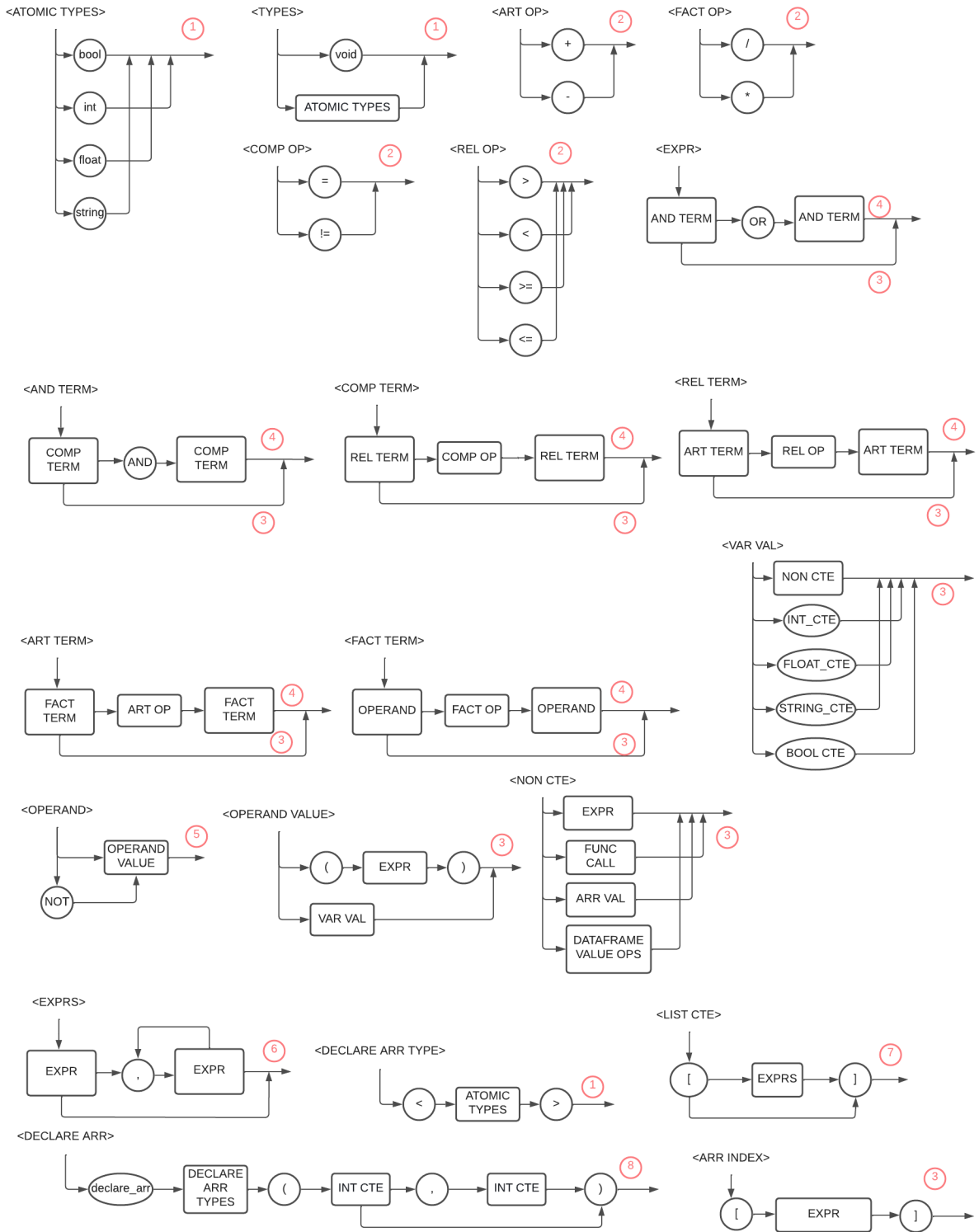
## Generación de Código Intermedio y Semántica

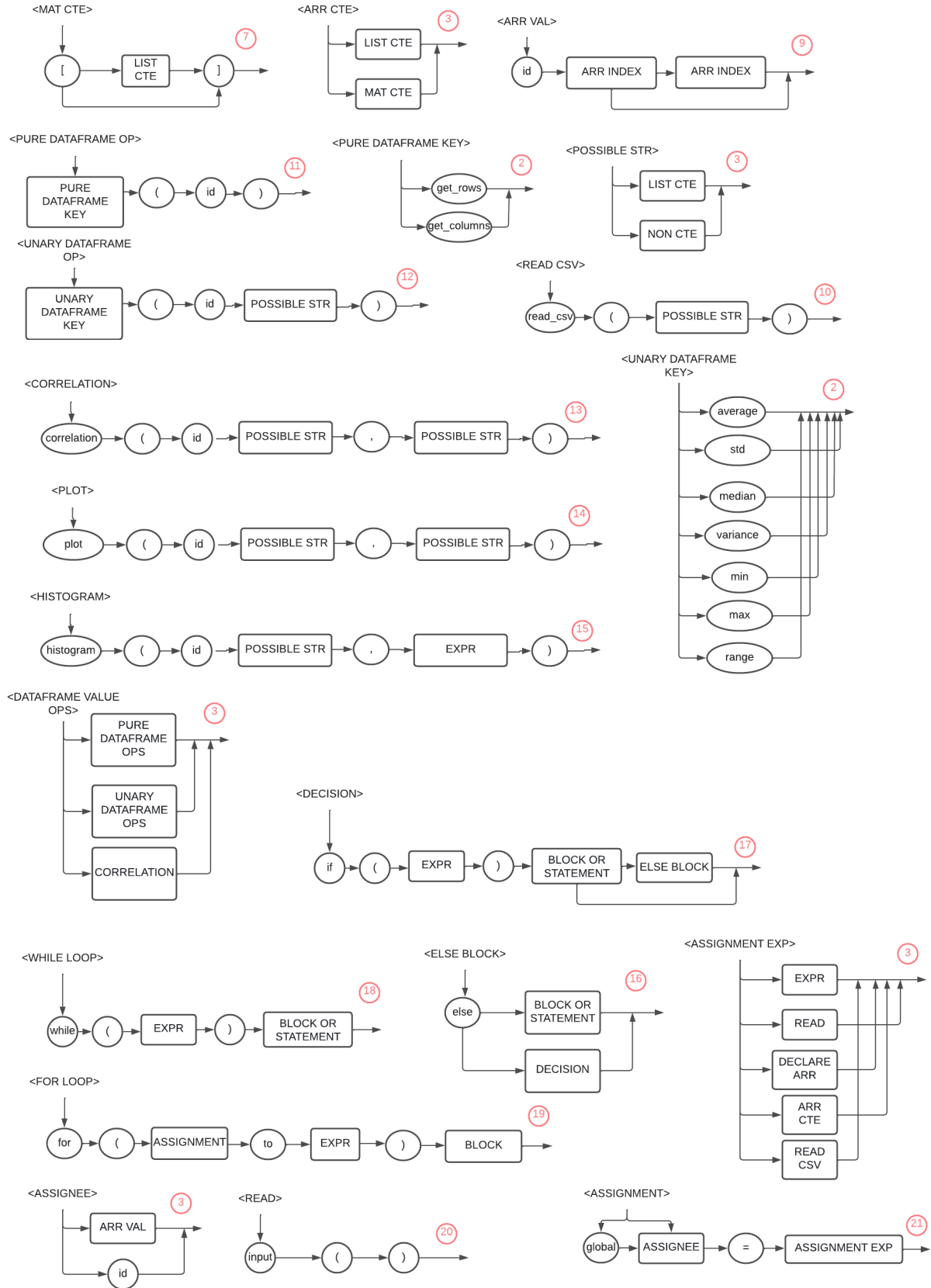
### Generación de AST

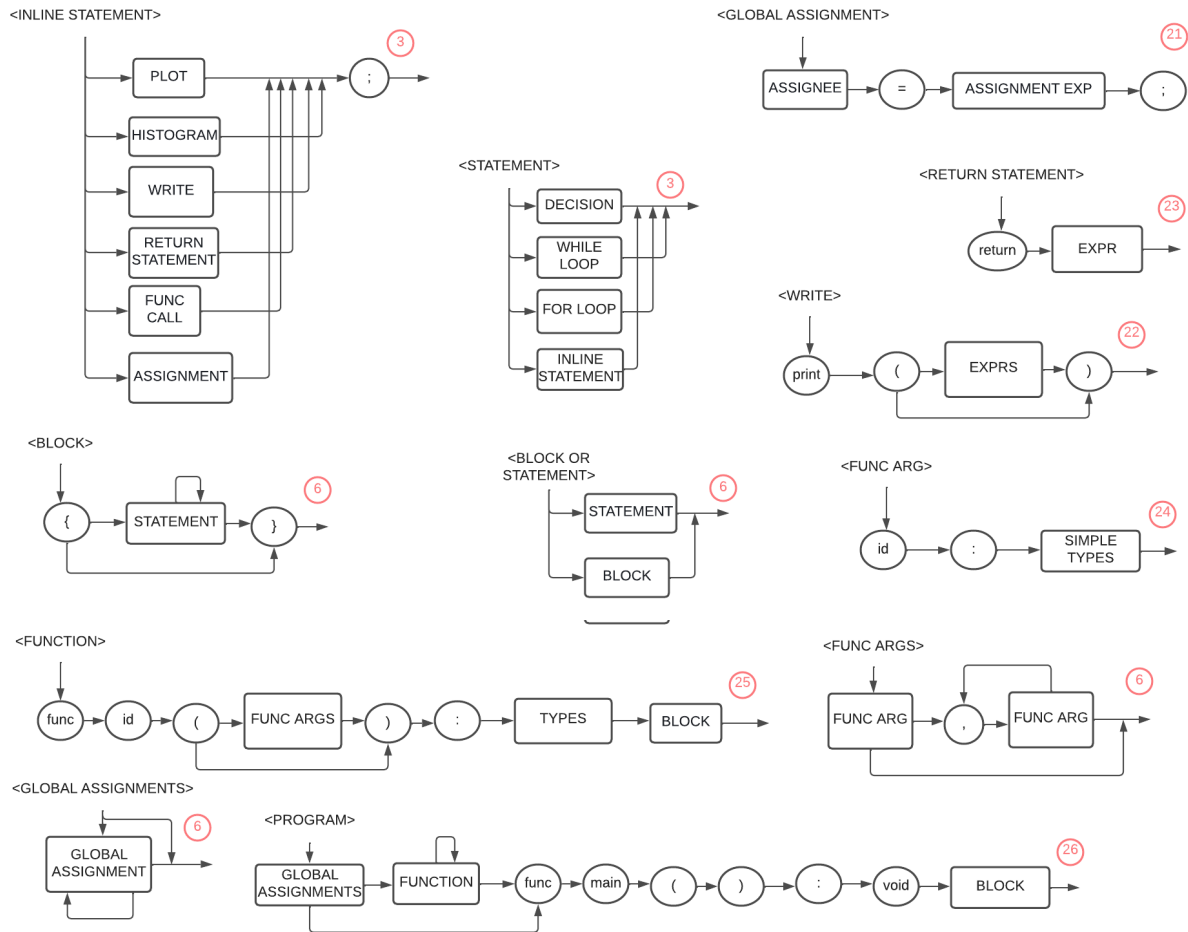
Durante el proceso de compilación, para poder implementar fácilmente las funcionalidades del lenguaje se implementaría la construcción de un Abstract Syntax Tree (AST).

La decisión de por qué un AST era la herramienta correcta en lugar de una compilación basada en puntos neurálgicos, es que un AST permite recordar el contexto en el que uno se encuentra más fácilmente. Al mismo tiempo, las herramientas usadas para la construcción y exploración de la gramática (*pest* y *pest\_consume*), utilizan un acercamiento funcional para “consumir” el árbol sintáctico inicial. Por lo que la creación de AST también fue demasiado intuitiva y directa.

A continuación se muestran los diagramas de la gramática y los puntos neurálgicos en el código (Siendo todos estos al final de la regla gramatical)







1.	Se crea una variable tipo Type (enum)
2.	Se crea una variable tipo Operator (enum)
3.	Se regresa nodo de AST
4.	Se crea un AST node de tipo BinaryOperator
5.	Si pasa por NOT crea un UnaryOperator, de lo contrario sólo regresa el AST ya creado
6.	Se crea lista de nodos de AST
7.	Se crea un AST node de tipo Array
8.	Se crea un AST node de tipo ArrayDeclaration
9.	Se crea un AST node de tipo ArrayVal
10.	Se crea un AST node de tipo ReadCsv



11.	Se crea un AST node de tipo PureDataframeOp
12.	Se crea un AST node de tipo UnaryDataframeOp
13.	Se crea un AST node de tipo Correlation
14.	Se crea un AST node de tipo Plot
15.	Se crea un AST node de tipo Histogram
16.	Se crea un AST node de tipo ElseBlock
17.	Se crea un AST node de tipo Decision
18.	Se crea un AST node de tipo While
19.	Se crea un AST node de tipo For
20.	Se crea un AST node de tipo Read
21.	Se crea un AST node de tipo Assignment
22.	Se crea un AST node de tipo Write
23.	Se crea un AST node de tipo Return
24.	Se crea un AST node de tipo Argument
25.	Se crea un AST node de tipo PureDataframeOp
26.	Se crea un AST node de tipo Main

Además de los tipos de ASTs mencionados con anterioridad, hay 5 tipos más creados a partir de los tokens léxicos:

- Id a partir de un id
- Integer a partir de un INT\_CTE
- Float a partir de un FLOAT\_CTE
- Bool a partir de un BOOL\_CTE
- String a partir de un STRING\_CTE

De tal manera que el AST de Raoul tiene un total de 28 tipos diferentes de nodos. A continuación se describirán cada uno de ellos:

1. **Id**: Nodo que representa un id, ya sea una variable o una función. El valor de este es el id
2. **Integer**: Nodo que representa un número entero. Su valor es dicho número
3. **Float**: Nodo que representa un número flotante. Su valor es dicho número
4. **Bool**: Nodo que representa un valor booleano. Su valor es ese valor booleano

5. **String**: Nodo que representa una cadena de caracteres. Su valor es dicha cadena
6. **Array**: Nodo que representa un arreglo constante de valores. Tiene como valor nodos de AST que representa cada uno de sus elementos
7. **ArrayDeclaration**: Opción que ofrece Raoul por si se quiere hacer un arreglo con valores no inicializados desde un inicio
  - a. **data\_type**: Representa el tipo de los elementos del arreglo
  - b. **dim1**: Cantidad de elementos en la primera dimensión
  - c. **dim2**: Valor opcional que representa la cantidad de elementos para la segunda dimensión (Matriz)
8. **ArrayVal**: Nodo que representa cuando el programador quiere acceder a una casilla del arreglo
  - a. **name**: Nombre de la variable
  - b. **idx\_1**: Índice para la primera dimensión
  - c. **idx\_2**: Valor opcional que representa el índice para la segunda dimensión
9. **Assignment**: Nodo que representa cuando se hace una asignación a una variable
  - a. **assignee**: Nodo del ast que representa a quien le va a asignar. Este puede llegar a ser **Id** o **ArrayVal**
  - b. **global**: Si la asignación es a una variable de scope global
  - c. **value**: El valor que se le va a asignar. Siendo esta también un nodo del AST
10. **UnaryOperation**: Nodo que representa una operación unaria
  - a. **operator**: Valor de tipo Operator que representa la operación a ejecutar
  - b. **operand**: Operando al cual se le aplicará la operación
11. **BinaryOperation**: Nodo que representa una operación binaria
  - a. **operator**: Valor de tipo Operator que representa la operación a ejecutar
  - b. **lhs**: Operando izquierdo de la operación
  - c. **rhs**: Operando derecho de la operación
12. **Main**: Nodo que representa el programa completo, junto con main
  - a. **assignments**: Asignaciones globales
  - b. **body**: Estatutos de la función main
  - c. **functions**: Funciones del programa
13. **Argument**: Nodo que representa un argumento de una función
  - a. **arg\_type**: El tipo del argumento
  - b. **name**: El nombre del argumento
14. **Function**: Nodo que representa una función
  - a. **arguments**: Lista de argumentos representados por nodos del AST
  - b. **body**: Los estatutos de la función
  - c. **name**: El nombre de la función
  - d. **return\_type**: El tipo de regreso de la función
15. **Write**: Nodo que representa una impresión al usuario. Tiene como valor las expresiones que va a imprimir
16. **Read**: Nodo que representa lectura
17. **Decision**: Nodo que representa un if
  - a. **expr**: Nodo que representa la condición del if

- b. **statements**: Estatutos que se corren si se acepta la condición
  - c. **else\_block**: Nodo opcional que representa el branch si acaso no llegase a pasar la condición
- 18. **ElseBlock**: Nodo que representa el *else* de un *if*. Tiene como valor los estatutos del *else*
- 19. **While**: Nodo que representa un *while*
  - a. **expr**: Nodo que representa la condición del *while*
  - b. **statements**: Estatutos que se corren si se acepta la condición
- 20. **For**: Nodo que representa un *for*
  - a. **assignment**: Nodo que representa la asignación de un *for*
  - b. **expr**: Nodo que representa la condición del *for*
  - c. **statements**: Estatutos que se corren si se acepta la condición
- 21. **FuncCall**: Nodo que representa una llamada a una función
  - a. **name**: El nombre de la función a la que se está llamando
  - b. **exprs**: Lista de expresiones representando los argumentos que se le mandan a la función
- 22. **Return**: Nodo que representa un estatuto de regreso. Tiene como valor un nodo de AST que representa el valor por regresar
- 23. **ReadCSV**: Nodo que representa leer un csv para crear un dataframe con los datos de ese csv. Como valor tiene un nodo de AST que representa el nombre del archivo
- 24. **PureDataframeOp**: Nodo que representa una operación para un dataframe, sin más argumentos
  - a. **name**: El nombre del dataframe
  - b. **operator**: Valor de tipo Operator que representa la operación a ejecutar
- 25. **UnaryDataframeOp**: Nodo que representa una operación para un dataframe que tiene como argumento el nombre de una columna
  - a. **column**: Nodo que representa la columna para la operación
  - b. **name**: El nombre del dataframe
  - c. **operator**: Valor de tipo Operator que representa la operación a ejecutar
- 26. **Correlation**: Nodo que representa la operación de calcular la correlación entre 2 columnas de un dataframe
  - a. **name**: El nombre del dataframe
  - b. **column\_1**: Nodo que representa la 1º columna para la operación
  - c. **column\_2**: Nodo que representa la 2º columna para la operación
- 27. **Plot**: Nodo que representa una operación para mostrar un *Scatter Plot* entre dos columnas de un dataframe.
  - a. **name**: El nombre del dataframe
  - b. **column\_1**: Nodo que representa la 1º columna para la operación
  - c. **column\_2**: Nodo que representa la 2º columna para la operación
- 28. **Histogram**: Nodo que representa una operación para mostrar un histograma de una columna y unos bins, de un dataframe.
  - a. **name**: El nombre del dataframe
  - b. **bins**: Nodo que representa la cantidad de bins en el histograma
  - c. **column**: Nodo que representa la columna para la operación

Por último, se mostrará un ejemplo de un AST creado a partir de un código

### Código

```
func main(): void {
    a = true OR false;
    b = 1 AND true;
    c = 1 <= 2;
    d = "1" < 2;
    e = 2 >= 1;
    f = 2 > 1;
    e = 3 == 3;
    f = 3 != 2;
    g = "2.1" + 2.0;
    h = 2 - 2.0;
    i = 4 * 4;
    j = 4 / 4;
    k = NOT false;
    print(a, b, c, d, e, f, g, h, i, j, k);
}
```

### AST generado

```
Main([[], [], [
    Assignment(false, Id(a), BinaryOperation(Or, Bool(true), Bool(false))),
    Assignment(false, Id(b), BinaryOperation(And, Integer(1), Bool(true))),
    Assignment(false, Id(c), BinaryOperation(Lte, Integer(1), Integer(2))),
    Assignment(false, Id(d), BinaryOperation(Lt, String(1), Integer(2))),
    Assignment(false, Id(e), BinaryOperation(Gte, Integer(2), Integer(1))),
    Assignment(false, Id(f), BinaryOperation(Gt, Integer(2), Integer(1))),
    Assignment(false, Id(e), BinaryOperation(Eq, Integer(3), Integer(3))),
    Assignment(false, Id(f), BinaryOperation(Ne, Integer(3), Integer(2))),
    Assignment(false, Id(g), BinaryOperation(Sum, String(2.1), Float(2))),
    Assignment(false, Id(h), BinaryOperation(Minus, Integer(2), Float(2))),
    Assignment(false, Id(i), BinaryOperation(Times, Integer(4), Integer(4))),
    Assignment(false, Id(j), BinaryOperation(Div, Integer(4), Integer(4))),
    Assignment(false, Id(k), Unary(Not, Bool(false))),
    Write([Id(a), Id(b), Id(c), Id(d), Id(e), Id(f), Id(g), Id(h), Id(i),
Id(j), Id(k)]),
]))
```

## Inferencia de tipado

En *Raoul*, las variables no se definen en conjunto a su tipo de variable. Sino, que la variable infiere su propio tipo a partir del valor que se le asigna. Y se sabe el tipo del valor a partir del tipo del nodo del AST que lo representa.

Por ejemplo en el caso de la siguiente asignación:

```
a = 1
```

El ast que va a representar la asignación, es el siguiente

```
Assignment(false, Id(a), Integer(1))
```

Como se mencionó anteriormente, el tercer valor del assignment es el valor. En este caso, es un `Integer(1)`, por lo tanto, la semántica implementada, le dice que ese valor, es de tipo *Int* o entero.

A continuación se muestran todas las asociaciones que se tienen:

	Tipo asociado
<b>Integer</b>	Int
<b>PureDataframeOp</b>	Int
<b>Float</b>	Float
<b>UnaryDataframeOp</b>	Float
<b>Correlation</b>	Float
<b>String</b>	String
<b>Read</b>	String
<b>Bool</b>	Bool

Para los otros nodos, se basa en el cubo semántico como en el caso de las operaciones binarias y unarias. O, se determinan a partir de las variables ya declaradas.

## Casts implícitos

Debido a que no se tiene un control tan fácil del tipado, el lenguaje implementa una variedad de casts implícitos. En la siguiente tabla se muestra cuales son dichos casts

	Puede ser casteado a
<b>Int</b>	Float, String y Bool
<b>Float</b>	Int, String
<b>String</b>	Int, Float
<b>Bool</b>	Int

En el caso de que se caste de String, este cast puede fallar, en caso de que el string no se pueda representar como número. Y en caso de que fallo, arroja un error al usuario

## Especificaciones extras

Otra peculiaridad del lenguaje, es que se decidió que los estatutos de visualizaciones sean los últimos que se corran del programa. De tal forma que si hay más estatutos después de alguno de visualización, esos nunca se correran

## Cubo Semántico

*Operaciones asociativas (Sin importar el orden siguen siendo el mismo resultado)*

<b>INT</b>	
<b>=&gt; BOOL:</b> NOT	

<b>INT</b>	<b>INT</b>
<b>=&gt; INT:</b> *, /, -, + <b>=&gt; BOOL:</b> <, >, <=, >=, ==, !=, AND, OR	

<b>INT</b>	<b>FLOAT / STRING</b>
<b>=&gt; FLOAT:</b> *, /, -, + <b>=&gt; BOOL:</b> <, >, <=, >=, ==, !=	

<b>INT</b>	<b>BOOL</b>
<b>=&gt; BOOL: ==, !=, AND, OR</b>	

<b>FLOAT</b>	<b>FLOAT / STRING</b>
<b>=&gt; FLOAT: *, /, -, +</b>	

<b>BOOL</b>	
<b>=&gt; BOOL: NOT</b>	

<b>BOOL</b>	<b>BOOL</b>
<b>=&gt; BOOL: &lt;, &gt;, &lt;=, &gt;=, ==, !=, AND, OR</b>	

## Administración de Memoria

### Dir Func (diccionario)

func_name	→	Function				
		name	return_type	variables	first_quad	args
...						
func_name	→	Function				
		name	return_type	variables	first_quad	args

**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

**args:** lista de tuplas conteniendo una dirección y 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	dimensions
...					
var_name	→	Variable			
		name	data_type	address	dimensions

**name:** string con nombre de variable

**data\_type:** enumerador con el tipo de la variable

**address:** dirección virtual de la variable en memoria

**dimensions:** tupla que contiene dos enteros *nullable* representando las dimensiones

### 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	250
CHAR	500
STRING	750

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 una operación se termina



se revisa si alguna de las direcciones fue temporal, de ser así se “libera” que en código representa agregarlo a la lista de direcciones en espera. 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)

operator	op_1	op_2	res
----------	------	------	-----

**oper:** enumerador con el tipo de operación a realizar

**op\_1:** dirección virtual del primer operando de la operación

**op\_2:** 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      -      -      99
1    - Assignment 3000 -      1000
2    - Minus     22    3001  2000
3    - Lte       1000  2000  2750
4    - GotoF     2750 -      11
5    - Ver       1000  3002 -
6    - Sum       3000  1000  4000
7    - Print     4000 -      -
8    - PrintNl   -      -      -
9    - Inc       -      -      1000
10   - Goto      -      -      2
11   - EndProc   -      -      -
...

```

## Descripción de la Máquina Virtual

Equipo de cómputo, lenguaje, utilerías

	Librerías utilizadas	Lenguaje
vm	polars eframe std	Rust (1.60.0)

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. Esta clase memoria toma como argumentos un manejador de direcciones. En estos casos al generar los cuádruplos también se administran las direcciones constantes, y los punteros. Mientras que las locales y temporales, están asociadas a su respectiva función. Y las direcciones globales se generan junto al director de funciones.

La máquina virtual tiene además una pila de contexto:

...					
args	temp	local	1000	2	0
args	temp	local	1020	4	1
args	temp	local	1030	8	10

Cada contexto tiene como información la lista de direcciones de los argumentos de la función, sus memorias temporales y locales, el nombre de la función, el ip actual en ese context, y el *size* que ocupan sus recursos.

### Memoria (clase)

La memoria para la máquina virtual es una lista de espacios contiguos. Ya que se instancia con el manejador de directorios, la estructura ya sabe cuántos espacios va a necesitar “reservar” para todas las variables de este tipo. Aunado a eso, también nos permite crear un sistema de punteros, para señalar a partir de qué espacio empieza qué tipo de datos en la memoria

int_ptr		float_ptr		string_ptr		bool_ptr	
1	24	-3	2.0	3.14	“hello”	“world”	false

Ej: se recibe request para 1251. El contexto (1000) nos indica a cuál instancia de memoria acceder, que en este caso es la local activa. El tipo (250) nos indica que el desplazamiento se debe aplicar al inicio del float\_ptr. El desplazamiento (1) nos indica que se requiere acceder al segundo elemento, después de su puntero. Por ende la dirección virtual 1251 se traduce a la quinta 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:

$$\text{Address}(\text{id}[s_1]) = \text{BaseAddress}(\text{id}) + s_1$$

$$\text{Address}(\text{id}[s_1][s_2]) = \text{BaseAddress}(\text{id}) + s_1 * d_2 + s_2$$

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

```
func factorial(n: int): int {
    accum = 1;
    for (i = 2 to n) accum = accum * i;
    return accum;
}

func recursiveFactorial(n: int): int {
    if (n == 0) return 1;
    return n * recursiveFactorial(n - 1);
}

func main(): void {
    option = -1;
    while (option != 0 AND option != 1) {
        print("What option you want to use");
        print("  0 = Iterative");
        print("  1 = Recursive");
        option = input();
    }
    print("What factorial number you want to calculate?");
    n = input();
    if (option == 0) print(factorial(n));
    else if (option == 1) print(recursiveFactorial(n));
    else print("This should not happen");
}
```

```
Quads created sucessfully
0  - Goto      -      -      22
1  - Assignment 3000 -      1001
2  - Assignment 3001 -      1002
3  - Lte       1002 1000 2750
4  - GotoF     2750 -      9
5  - Times     1001 1002 2000
6  - Assignment 2000 -      1001
7  - Inc       -      -      1002
8  - Goto      -      -      3
9  - Return    1001 -      -
10 - EndProc   -      -      -
11 - Eq        1000 3002 2750
12 - GotoF     2750 -      14
13 - Return    3000 -      -
14 - Era       2      11     -
15 - Minus     1000 3000 2000
16 - Param     2000 -      0
17 - GoSub     11     -      -
18 - Assignment 1      -      2001
19 - Times     1000 2001 2002
20 - Return    2002 -      -
21 - EndProc   -      -      -
22 - Assignment 3003 -      1000
23 - Ne        1000 3002 2750
24 - Ne        1000 3000 2751
25 - And       2750 2751 2752
26 - GotoF     2752 -      36
27 - Print     3500 -      -
28 - PrintNl   -      -      -
29 - Print     3501 -      -
30 - PrintNl   -      -      -
31 - Print     3502 -      -
32 - PrintNl   -      -      -
33 - Read      -      -      2500
34 - Assignment 2500 -      1000
35 - Goto      -      -      23
36 - Print     3503 -      -
37 - PrintNl   -      -      -
38 - Read      -      -      2500
39 - Assignment 2500 -      1500
40 - Eq        1000 3002 2752
41 - GotoF     2752 -      49
42 - Era       5      1      -
43 - Param     1500 -      0
44 - GoSub     1      -      -
45 - Assignment 0      -      2000
46 - Print     2000 -      -
47 - PrintNl   -      -      -
48 - Goto      -      -      60
49 - Eq        1000 3000 2752
50 - GotoF     2752 -      58
51 - Era       5      11     -
52 - Param     1500 -      0
53 - GoSub     11     -      -
54 - Assignment 1      -      2001
55 - Print     2001 -      -
56 - PrintNl   -      -      -
57 - Goto      -      -      60
```

	58 - Print 3504 - -
	59 - PrintNl - - -
	60 - End - - -
	What option you want to use
	0 = Iterative
	1 = Recursive
	0
	What factorial number you want to calculate?
	5
	120

func fibo(n: int): int { if (n <= 1) return n; a = 0; b = 1; for (i = 2 to n) { next = a + b; a = b; b = next; } return next; }	Quads created sucessfully
func recursiveFibo(n: int): int { if (n <= 1) return n; return recursiveFibo(n - 2) + recursiveFibo(n - 1); }	0 - Goto - - 33
func main(): void { option = -1; while (option != 0 AND option != 1) { print("What option you want to use"); print(" 0 = Iterative"); print(" 1 = Recursive"); option = input(); } print("What fibonacci number you want to calculate?"); n = input(); if (option == 0) print(fibo(n)); else if (option == 1) print(recursiveFibo(n)); else print("This should not happen"); }	1 - Lte 1000 3000 2750
	2 - GotoF 2750 - 4
	3 - Return 1000 - -
	4 - Assignment 3001 - 1001
	5 - Assignment 3000 - 1002
	6 - Assignment 3002 - 1003
	7 - Lte 1003 1000 2750
	8 - GotoF 2750 - 15
	9 - Sum 1001 1002 2000
	10 - Assignment 2000 - 1004
	11 - Assignment 1002 - 1001
	12 - Assignment 1004 - 1002
	13 - Inc - - 1003
	14 - Goto - - 7
	15 - Return 1004 - -
	16 - EndProc - - -
	17 - Lte 1000 3000 2750
	18 - GotoF 2750 - 20
	19 - Return 1000 - -
	20 - Era 2 17 -
	21 - Minus 1000 3002 2000
	22 - Param 2000 - 0
	23 - GoSub 17 - -
	24 - Assignment 1 - 2001
	25 - Era 4 17 -
	26 - Minus 1000 3000 2002
	27 - Param 2002 - 0
	28 - GoSub 17 - -
	29 - Assignment 1 - 2002
	30 - Sum 2001 2002 2003
	31 - Return 2003 - -
	32 - EndProc - - -
	33 - Assignment 3003 - 1000
	34 - Ne 1000 3001 2750
	35 - Ne 1000 3000 2751
	36 - And 2750 2751 2752
	37 - GotoF 2752 - 47
	38 - Print 3500 - -
	39 - PrintNl - - -
	40 - Print 3501 - -
	41 - PrintNl - - -
	42 - Print 3502 - -
	43 - PrintNl - - -
	44 - Read - - 2500
	45 - Assignment 2500 - 1000
	46 - Goto - - 34
	47 - Print 3503 - -
	48 - PrintNl - - -

	49	-	Read	-	-	2500
	50	-	Assignment	2500	-	1500
	51	-	Eq	1000	3001	2752
	52	-	GotoF	2752	-	60
	53	-	Era	7	1	-
	54	-	Param	1500	-	0
	55	-	GoSub	1	-	-
	56	-	Assignment	0	-	2000
	57	-	Print	2000	-	-
	58	-	PrintNl	-	-	-
	59	-	Goto	-	-	71
	60	-	Eq	1000	3000	2752
	61	-	GotoF	2752	-	69
	62	-	Era	6	17	-
	63	-	Param	1500	-	0
	64	-	GoSub	17	-	-
	65	-	Assignment	1	-	2001
	66	-	Print	2001	-	-
	67	-	PrintNl	-	-	-
	68	-	Goto	-	-	71
	69	-	Print	3504	-	-
	70	-	PrintNl	-	-	-
	71	-	End	-	-	-
	What option you want to use					
	0 = Iterative					
	1 = Recursive					
	0					
	What fibonacci number you want to calculate?					
	5					
	5					

func main(): void {	Quads created sucessfully
dataframe = read_csv("song_data_clean.csv");	0 - Goto - - 1
rows = get_rows(dataframe);	1 - ReadCSV 3500 - -
columns = get_columns(dataframe);	2 - Rows - - 2000
print(rows, columns);	3 - Assignment 2000 - 1000
	4 - Columns - - 2001
col = "danceability";	5 - Assignment 2001 - 1001
	6 - Print 1000 - -
print(average(dataframe, "song_duration_ms"));	7 - Print 1001 - -
print(std(dataframe, col));	8 - PrintNl - - -
print(median(dataframe, col));	9 - Assignment 3501 - 1500
print(variance(dataframe, col));	10 - Average 3502 - 2250
print(min(dataframe, col));	11 - Print 2250 - -
print(max(dataframe, col));	12 - PrintNl - - -
print(range(dataframe, col));	13 - Std 1500 - 2250
	14 - Print 2250 - -
print(correlation(dataframe, col, "song_duration_ms"));	15 - PrintNl - - -
histogram(dataframe, "danceability", 5);	16 - Median 1500 - 2250
}	17 - Print 2250 - -
	18 - PrintNl - - -
	19 - Variance 1500 - 2250
	20 - Print 2250 - -
	21 - PrintNl - - -
	22 - Min 1500 - 2250
	23 - Print 2250 - -
	24 - PrintNl - - -
	25 - Max 1500 - 2250
	26 - Print 2250 - -
	27 - PrintNl - - -
	28 - Range 1500 - 2250

	29 - Print 2250 - - 30 - PrintNl - - - 31 - Corr 1500 3502 2250 32 - Print 2250 - - 33 - PrintNl - - - 34 - Histogram 3501 3000 - 35 - End - - -  13053 15 218645.22944916878 0.15916777771148127 0.637 0.025334381461611516 0 0.987 0.987 -0.08547008157088816
--	---

<pre> m1 = [[1, 2, 3], [1, 2, 3], [1, 2, 3]]; m2 = [[3, 2, 1], [3, 2, 1], [3, 2, 1]]; m3 = declare_arr&lt;int&gt;(3, 3);  func multiply(): void {     for (i = 0 to 2) {         for (j = 0 to 2) {             for (k = 0 to 2) global m3[i][j] = m1[i][k] * m2[k][j];         }     } }  func main(): void {     multiply();     for (i = 0 to 2) print(m3[i][0], m3[i][1], m3[i][2]); } </pre>	Quads created sucessfully 0 - Goto - - 34 1 - Assignment 3000 - 1000 2 - Lte 1000 3001 2750 3 - GotoF 2750 - 33 4 - Assignment 3000 - 1001 5 - Lte 1001 3001 2750 6 - GotoF 2750 - 3g 7 - Assignment 3000 - 1002 8 - Lte 1002 3001 2750 9 - GotoF 2750 - 29 10 - Ver 1000 3003 - 11 - Times 1000 3003 2000 12 - Ver 1001 3003 - 13 - Sum 3002 2000 2001 14 - Sum 2001 1001 4000 15 - Ver 1000 3003 - 16 - Times 1000 3003 2001 17 - Ver 1002 3003 - 18 - Sum 3000 2001 2002 19 - Sum 2002 1002 4001 20 - Ver 1002 3003 - 21 - Times 1002 3003 2002 22 - Ver 1001 3003 - 23 - Sum 3004 2002 2001 24 - Sum 2001 1001 4002 25 - Times 4001 4002 2001 26 - Assignment 2001 - 4000 27 - Inc - - 1002 28 - Goto - - 8 29 - Inc - - 1001 30 - Goto - - 5 31 - Inc - - 1000 32 - Goto - - 2 33 - EndProc - - - 34 - Ver 3000 3003 - 35 - Times 3000 3003 2000 36 - Ver 3000 3003 - 37 - Sum 3000 2000 2001 38 - Sum 2001 3000 4003 39 - Assignment 3005 - 4003 40 - Ver 3000 3003 - 41 - Times 3000 3003 2001
---	---

42	- Ver	3005	3003	-
43	- Sum	3000	2001	2002
44	- Sum	2002	3005	4004
45	- Assignment	3001	-	4004
46	- Ver	3000	3003	-
47	- Times	3000	3003	2002
48	- Ver	3001	3003	-
49	- Sum	3000	2002	2001
50	- Sum	2001	3001	4005
51	- Assignment	3003	-	4005
52	- Ver	3005	3003	-
53	- Times	3005	3003	2001
54	- Ver	3000	3003	-
55	- Sum	3000	2001	2002
56	- Sum	2002	3000	4006
57	- Assignment	3005	-	4006
58	- Ver	3005	3003	-
59	- Times	3005	3003	2002
60	- Ver	3005	3003	-
61	- Sum	3000	2002	2001
62	- Sum	2001	3005	4007
63	- Assignment	3001	-	4007
64	- Ver	3005	3003	-
65	- Times	3005	3003	2001
66	- Ver	3001	3003	-
67	- Sum	3000	2001	2002
68	- Sum	2002	3001	4008
69	- Assignment	3003	-	4008
70	- Ver	3001	3003	-
71	- Times	3001	3003	2002
72	- Ver	3000	3003	-
73	- Sum	3000	2002	2001
74	- Sum	2001	3000	4009
75	- Assignment	3005	-	4009
76	- Ver	3001	3003	-
77	- Times	3001	3003	2001
78	- Ver	3005	3003	-
79	- Sum	3000	2001	2002
80	- Sum	2002	3005	4010
81	- Assignment	3001	-	4010
82	- Ver	3001	3003	-
83	- Times	3001	3003	2002
84	- Ver	3001	3003	-
85	- Sum	3000	2002	2001
86	- Sum	2001	3001	4011
87	- Assignment	3003	-	4011
88	- Ver	3000	3003	-
89	- Times	3000	3003	2001
90	- Ver	3000	3003	-
91	- Sum	3004	2001	2002
92	- Sum	2002	3000	4012
93	- Assignment	3003	-	4012
94	- Ver	3000	3003	-
95	- Times	3000	3003	2002
96	- Ver	3005	3003	-
97	- Sum	3004	2002	2001
98	- Sum	2001	3005	4013
99	- Assignment	3001	-	4013
100	- Ver	3000	3003	-
101	- Times	3000	3003	2001
102	- Ver	3001	3003	-
103	- Sum	3004	2001	2002
104	- Sum	2002	3001	4014
105	- Assignment	3005	-	4014
106	- Ver	3005	3003	-



107	- Times	3005	3003	2002
108	- Ver	3000	3003	-
109	- Sum	3004	2002	2001
110	- Sum	2001	3000	4015
111	- Assignment	3003	-	4015
112	- Ver	3005	3003	-
113	- Times	3005	3003	2001
114	- Ver	3005	3003	-
115	- Sum	3004	2001	2002
116	- Sum	2002	3005	4016
117	- Assignment	3001	-	4016
118	- Ver	3005	3003	-
119	- Times	3005	3003	2002
120	- Ver	3001	3003	-
121	- Sum	3004	2002	2001
122	- Sum	2001	3001	4017
123	- Assignment	3005	-	4017
124	- Ver	3001	3003	-
125	- Times	3001	3003	2001
126	- Ver	3000	3003	-
127	- Sum	3004	2001	2002
128	- Sum	2002	3000	4018
129	- Assignment	3003	-	4018
130	- Ver	3001	3003	-
131	- Times	3001	3003	2002
132	- Ver	3005	3003	-
133	- Sum	3004	2002	2001
134	- Sum	2001	3005	4019
135	- Assignment	3001	-	4019
136	- Ver	3001	3003	-
137	- Times	3001	3003	2001
138	- Ver	3001	3003	-
139	- Sum	3004	2001	2002
140	- Sum	2002	3001	4020
141	- Assignment	3005	-	4020
142	- Era	7	1	-
143	- GoSub	1	-	-
144	- Assignment	3000	-	1000
145	- Lte	1000	3001	2750
146	- GotoF	2750	-	168
147	- Ver	1000	3003	-
148	- Times	1000	3003	2002
149	- Ver	3000	3003	-
150	- Sum	3002	2002	2001
151	- Sum	2001	3000	4021
152	- Print	4021	-	-
153	- Ver	1000	3003	-
154	- Times	1000	3003	2001
155	- Ver	3005	3003	-
156	- Sum	3002	2001	2002
157	- Sum	2002	3005	4022
158	- Print	4022	-	-
159	- Ver	1000	3003	-
160	- Times	1000	3003	2002
161	- Ver	3001	3003	-
162	- Sum	3002	2002	2001
163	- Sum	2001	3001	4023
164	- Print	4023	-	-
165	- PrintNl	-	-	-
166	- Inc	-	-	1000
167	- Goto	-	-	145
168	- End	-	-	-

9 6 3

9 6 3

```

a = [4, 1, 5, 12, 42, 13, 69, 25, 3, 0, 2];
b = declare_arr<int>(11);
limit = 11;

func printArr(): void {
    for (i = 0 to limit - 1) print(a[i]);
}

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

    while (l1 <= mid AND l2 <= high) {
        if(a[l1] <= a[l2]) {
            global b[i] = a[l1];
            l1 = l1 + 1;
        }
        else {
            global b[i] = a[l2];
            l2 = l2 + 1;
        }
        i = i + 1;
    }

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

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

    for (i = low to high) global a[i] = b[i];
}

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

func main(): void {
    print("Values (11):");
    for (i = 0 to limit - 1) {
        global a[i] = input();
    }
    sort(0, limit - 1);
    print("Result: ");
    printArr();
}

```

Quads created sucessfully

0	- Goto	-	-	99
1	- Assignment	3000	-	1000
2	- Minus	22	3001	2000
3	- Lte	1000	2000	2750
4	- GotoF	2750	-	11
5	- Ver	1000	3002	-
6	- Sum	3000	1000	4000
7	- Print	4000	-	-
8	- PrintNl	-	-	-
9	- Inc	-	-	1000
10	- Goto	-	-	2
11	- EndProc	-	-	-
12	- Assignment	1000	-	1003
13	- Sum	1001	3001	2000
14	- Assignment	2000	-	1004
15	- Assignment	1000	-	1005
16	- Lte	1003	1001	2750
17	- Lte	1004	1002	2751
18	- And	2750	2751	2752
19	- GotoF	2752	-	44
20	- Ver	1003	3002	-
21	- Sum	3000	1003	4001
22	- Ver	1004	3002	-
23	- Sum	3000	1004	4002
24	- Lte	4001	4002	2752
25	- GotoF	2752	-	34
26	- Ver	1005	3002	-
27	- Sum	3002	1005	4003
28	- Ver	1003	3002	-
29	- Sum	3000	1003	4004
30	- Assignment	4004	-	4003
31	- Sum	1003	3001	2001
32	- Assignment	2001	-	1003
33	- Goto	-	-	41
34	- Ver	1005	3002	-
35	- Sum	3002	1005	4005
36	- Ver	1004	3002	-
37	- Sum	3000	1004	4006
38	- Assignment	4006	-	4005
39	- Sum	1004	3001	2001
40	- Assignment	2001	-	1004
41	- Sum	1005	3001	2001
42	- Assignment	2001	-	1005
43	- Goto	-	-	16
44	- Lte	1003	1001	2752
45	- GotoF	2752	-	56
46	- Ver	1005	3002	-
47	- Sum	3002	1005	4007
48	- Ver	1003	3002	-
49	- Sum	3000	1003	4008
50	- Assignment	4008	-	4007
51	- Sum	1005	3001	2001
52	- Assignment	2001	-	1005
53	- Sum	1003	3001	2001
54	- Assignment	2001	-	1003
55	- Goto	-	-	44
56	- Lte	1004	1002	2752
57	- GotoF	2752	-	68
58	- Ver	1005	3002	-

59	- Sum	3002	1005	4009
60	- Ver	1004	3002	-
61	- Sum	3000	1004	4010
62	- Assignment	4010	-	4009
63	- Sum	1005	3001	2001
64	- Assignment	2001	-	1005
65	- Sum	1004	3001	2001
66	- Assignment	2001	-	1004
67	- Goto	-	-	56
68	- Assignment	1000	-	1005
69	- Lte	1005	1002	2752
70	- GotoF	2752	-	78
71	- Ver	1005	3002	-
72	- Sum	3000	1005	4011
73	- Ver	1005	3002	-
74	- Sum	3002	1005	4012
75	- Assignment	4012	-	4011
76	- Inc	-	-	1005
77	- Goto	-	-	69
78	- EndProc	-	-	-
79	- Lt	1000	1001	2750
80	- GotoF	2750	-	98
81	- Sum	1000	1001	2000
82	- Div	2000	3003	2001
83	- Assignment	2001	-	1002
84	- Era	6	79	-
85	- Param	1000	-	0
86	- Param	1002	-	1
87	- GoSub	79	-	-
88	- Era	6	79	-
89	- Sum	1002	3001	2001
90	- Param	2001	-	0
91	- Param	1001	-	1
92	- GoSub	79	-	-
93	- Era	11	12	-
94	- Param	1000	-	0
95	- Param	1002	-	1
96	- Param	1001	-	2
97	- GoSub	12	-	-
98	- EndProc	-	-	-
99	- Ver	3000	3002	-
100	- Sum	3000	3000	4013
101	- Assignment	3004	-	4013
102	- Ver	3001	3002	-
103	- Sum	3000	3001	4014
104	- Assignment	3001	-	4014
105	- Ver	3003	3002	-
106	- Sum	3000	3003	4015
107	- Assignment	3005	-	4015
108	- Ver	3006	3002	-
109	- Sum	3000	3006	4016
110	- Assignment	3007	-	4016
111	- Ver	3004	3002	-
112	- Sum	3000	3004	4017
113	- Assignment	3008	-	4017
114	- Ver	3005	3002	-
115	- Sum	3000	3005	4018
116	- Assignment	3009	-	4018
117	- Ver	3010	3002	-
118	- Sum	3000	3010	4019
119	- Assignment	3011	-	4019
120	- Ver	3012	3002	-
121	- Sum	3000	3012	4020
122	- Assignment	3013	-	4020
123	- Ver	3014	3002	-

	124 - Sum 3000 3014 4021
	125 - Assignment 3006 - 4021
	126 - Ver 3015 3002 -
	127 - Sum 3000 3015 4022
	128 - Assignment 3000 - 4022
	129 - Ver 3016 3002 -
	130 - Sum 3000 3016 4023
	131 - Assignment 3003 - 4023
	132 - Assignment 3002 - 22
	133 - Print 3500 - -
	134 - PrintNl - - -
	135 - Assignment 3000 - 1000
	136 - Minus 22 3001 2000
	137 - Lte 1000 2000 2750
	138 - GotoF 2750 - 145
	139 - Ver 1000 3002 -
	140 - Sum 3000 1000 4024
	141 - Read - - 2500
	142 - Assignment 2500 - 4024
	143 - Inc - - 1000
	144 - Goto - - 136
	145 - Era 6 79 -
	146 - Minus 22 3001 2001
	147 - Param 3000 - 0
	148 - Param 2001 - 1
	149 - GoSub 79 - -
	150 - Print 3501 - -
	151 - PrintNl - - -
	152 - Era 3 1 -
	153 - GoSub 1 - -
	154 - End - - -
	Values (11):
	5
	2
	6
	3
	7
	1
	9
	4
	6
	3
	6
	Result:
	1
	2
	3
	3
	4
	5
	6
	6
	6
	7
	9

<pre> a = [4, 1, 5, 12, 42, 13, 69, 25, 3, 0, 2]; limit = 11;  func printArr(): void {     for (i = 0 to limit - 1) print(a[i]); </pre>	<pre> Quads created sucessfully 0 - Goto - - 71 1 - Assignment 3000 - 1000 2 - Minus 11 3001 2000 3 - Lte 1000 2000 2750 </pre>
---	---

```

}

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

```

```

func partition(low: int, high: int): int {
    pivot = a[high];
    i = low - 1;

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

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

```

```

func sort(low: int, high: int): void {
    if (low < high) {
        pi = partition(low, high);

        sort(low, pi - 1);
        sort(pi + 1, high);
    }
}

```

```

func main(): void {
    print("Values (11):");
    for (i = 0 to limit - 1) {
        global a[i] = input();
    }
    sort(0, limit - 1);
    print("Result: ");
    printArr();
}

```

```

4  - GotoF      2750 -    11
5  - Ver        1000 3002 -
6  - Sum        3000 1000 4000
7  - Print      4000 -    -
8  - PrintNl    -    -    -
9  - Inc        -    -    1000
10 - Goto       -    -    2
11 - EndProc    -    -    -
12 - Ver        1000 3002 -
13 - Sum        3000 1000 4001
14 - Assignment 4001 -    1002
15 - Ver        1000 3002 -
16 - Sum        3000 1000 4002
17 - Ver        1001 3002 -
18 - Sum        3000 1001 4003
19 - Assignment 4003 -    4002
20 - Ver        1001 3002 -
21 - Sum        3000 1001 4004
22 - Assignment 1002 -    4004
23 - EndProc    -    -    -
24 - Ver        1001 3002 -
25 - Sum        3000 1001 4005
26 - Assignment 4005 -    1002
27 - Minus      1000 3001 2000
28 - Assignment 2000 -    1003
29 - Assignment 1000 -    1004
30 - Lte        1004 1001 2750
31 - GotoF      2750 -    44
32 - Ver        1004 3002 -
33 - Sum        3000 1004 4006
34 - Lt         4006 1002 2750
35 - GotoF      2750 -    42
36 - Sum        1003 3001 2001
37 - Assignment 2001 -    1003
38 - Era        3    12    -
39 - Param      1003 -    0
40 - Param      1004 -    1
41 - GoSub      12    -    -
42 - Inc        -    -    1004
43 - Goto       -    -    30
44 - Era        3    12    -
45 - Sum        1003 3001 2001
46 - Param      2001 -    0
47 - Param      1001 -    1
48 - GoSub      12    -    -
49 - Sum        1003 3001 2001
50 - Return     2001 -    -
51 - EndProc    -    -    -
52 - Lt         1000 1001 2750
53 - GotoF      2750 -    70
54 - Era        8    24    -
55 - Param      1000 -    0
56 - Param      1001 -    1
57 - GoSub      24    -    -
58 - Assignment 12    -    2000
59 - Assignment 2000 -    1002
60 - Era        5    52    -
61 - Minus      1002 3001 2001
62 - Param      1000 -    0
63 - Param      2001 -    1
64 - GoSub      52    -    -
65 - Era        6    52    -
66 - Sum        1002 3001 2001
67 - Param      2001 -    0
68 - Param      1001 -    1

```

69	- GoSub	52	-	-
70	- EndProc	-	-	-
71	- Ver	3000	3002	-
72	- Sum	3000	3000	4007
73	- Assignment	3003	-	4007
74	- Ver	3001	3002	-
75	- Sum	3000	3001	4008
76	- Assignment	3001	-	4008
77	- Ver	3004	3002	-
78	- Sum	3000	3004	4009
79	- Assignment	3005	-	4009
80	- Ver	3006	3002	-
81	- Sum	3000	3006	4010
82	- Assignment	3007	-	4010
83	- Ver	3003	3002	-
84	- Sum	3000	3003	4011
85	- Assignment	3008	-	4011
86	- Ver	3005	3002	-
87	- Sum	3000	3005	4012
88	- Assignment	3009	-	4012
89	- Ver	3010	3002	-
90	- Sum	3000	3010	4013
91	- Assignment	3011	-	4013
92	- Ver	3012	3002	-
93	- Sum	3000	3012	4014
94	- Assignment	3013	-	4014
95	- Ver	3014	3002	-
96	- Sum	3000	3014	4015
97	- Assignment	3006	-	4015
98	- Ver	3015	3002	-
99	- Sum	3000	3015	4016
100	- Assignment	3000	-	4016
101	- Ver	3016	3002	-
102	- Sum	3000	3016	4017
103	- Assignment	3004	-	4017
104	- Assignment	3002	-	11
105	- Print	3500	-	-
106	- PrintNl	-	-	-
107	- Assignment	3000	-	1000
108	- Minus	11	3001	2000
109	- Lte	1000	2000	2750
110	- GotoF	2750	-	117
111	- Ver	1000	3002	-
112	- Sum	3000	1000	4018
113	- Read	-	-	2500
114	- Assignment	2500	-	4018
115	- Inc	-	-	1000
116	- Goto	-	-	108
117	- Era	6	52	-
118	- Minus	11	3001	2001
119	- Param	3000	-	0
120	- Param	2001	-	1
121	- GoSub	52	-	-
122	- Print	3501	-	-
123	- PrintNl	-	-	-
124	- Era	3	1	-
125	- GoSub	1	-	-
126	- End	-	-	-

Values (11):

11  
64  
23  
54  
3

	76
	3
	45
	6
	2
	35
	Result:
	2
	3
	3
	6
	11
	23
	35
	45
	54
	64
	76

<pre> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  func binarySearch(left: int, right: int, target: int): int {     if (right &lt; left) return -1;     mid = left + (right - left) / 2;     if (a[mid] == target) return mid;     if (a[mid] &gt; target) return binarySearch(left, mid - 1, target);     return binarySearch(mid + 1, right, target); }  func linearSearch(target: int): int {     i = 0;     while (a[i] != target AND i &lt; 10) i = i + 1;     if (i == 10) return -1;     return i; }  func main(): void {     option = -1;     while (option != 0 AND option != 1) {         print("What option you want to use");         print("  0 = Binary Search");         print("  1 = Linear Search");         option = input();     }     print("Value to search:");     value = input();     if (option == 0) print("Index:", binarySearch(0, 9, value));     else if (option == 1) print("Index:", linearSearch(value));     else print("This should not happen"); } </pre>	<pre> Quads created sucessfully 0  - Goto      -      -      49 1  - Lt        1001  1000  2750 2  - GotoF     2750  -      4 3  - Return    3000  -      - 4  - Minus     1001  1000  2000 5  - Div       2000  3001  2001 6  - Sum       1000  2001  2002 7  - Assignment 2002  -      1003 8  - Ver       1003  3003  - 9  - Sum       3002  1003  4000 10 - Eq        4000  1002  2750 11 - GotoF     2750  -      13 12 - Return    1003  -      - 13 - Ver       1003  3003  - 14 - Sum       3002  1003  4001 15 - Gt        4001  1002  2750 16 - GotoF     2750  -      25 17 - Era       8     1     - 18 - Minus     1003  3004  2002 19 - Param     1000  -      0 20 - Param     2002  -      1 21 - Param     1002  -      2 22 - GoSub     1     -      - 23 - Assignment 10    -      2002 24 - Return    2002  -      - 25 - Era       8     1     - 26 - Sum       1003  3004  2002 27 - Param     2002  -      0 28 - Param     1001  -      1 29 - Param     1002  -      2 30 - GoSub     1     -      - 31 - Assignment 10    -      2002 32 - Return    2002  -      - 33 - EndProc   -     -      - 34 - Assignment 3002  -      1001 35 - Ver       1001  3003  - 36 - Sum       3002  1001  4002 37 - Ne        4002  1000  2750 38 - Lt        1001  3003  2751 39 - And       2750  2751  2752 40 - GotoF     2752  -      44 41 - Sum       1001  3004  2000 </pre>
--	--

42	- Assignment	2000	-	1001
43	- Goto	-	-	35
44	- Eq	1001	3003	2752
45	- GotoF	2752	-	47
46	- Return	3000	-	-
47	- Return	1001	-	-
48	- EndProc	-	-	-
49	- Ver	3002	3003	-
50	- Sum	3002	3002	4003
51	- Assignment	3004	-	4003
52	- Ver	3004	3003	-
53	- Sum	3002	3004	4004
54	- Assignment	3001	-	4004
55	- Ver	3001	3003	-
56	- Sum	3002	3001	4005
57	- Assignment	3005	-	4005
58	- Ver	3005	3003	-
59	- Sum	3002	3005	4006
60	- Assignment	3006	-	4006
61	- Ver	3006	3003	-
62	- Sum	3002	3006	4007
63	- Assignment	3007	-	4007
64	- Ver	3007	3003	-
65	- Sum	3002	3007	4008
66	- Assignment	3008	-	4008
67	- Ver	3008	3003	-
68	- Sum	3002	3008	4009
69	- Assignment	3009	-	4009
70	- Ver	3009	3003	-
71	- Sum	3002	3009	4010
72	- Assignment	3010	-	4010
73	- Ver	3010	3003	-
74	- Sum	3002	3010	4011
75	- Assignment	3011	-	4011
76	- Ver	3011	3003	-
77	- Sum	3002	3011	4012
78	- Assignment	3003	-	4012
79	- Assignment	3000	-	1000
80	- Ne	1000	3002	2750
81	- Ne	1000	3004	2751
82	- And	2750	2751	2752
83	- GotoF	2752	-	93
84	- Print	3500	-	-
85	- PrintNl	-	-	-
86	- Print	3501	-	-
87	- PrintNl	-	-	-
88	- Print	3502	-	-
89	- PrintNl	-	-	-
90	- Read	-	-	2500
91	- Assignment	2500	-	1000
92	- Goto	-	-	80
93	- Print	3503	-	-
94	- PrintNl	-	-	-
95	- Read	-	-	2500
96	- Assignment	2500	-	1500
97	- Eq	1000	3002	2752
98	- GotoF	2752	-	109
99	- Print	3504	-	-
100	- Era	8	1	-
101	- Param	3002	-	0
102	- Param	3011	-	1
103	- Param	1500	-	2
104	- GoSub	1	-	-
105	- Assignment	10	-	2000
106	- Print	2000	-	-



	<pre> 107 - PrintNl      -   -   - 108 - Goto         -   -  121 109 - Eq           1000 3004 2752 110 - GotoF        2752 -   119 111 - Print        3504 -   - 112 - Era          6   34   - 113 - Param        1500 -   0 114 - GoSub        34   -   - 115 - Assignment   11   -  2001 116 - Print        2001 -   - 117 - PrintNl      -   -   - 118 - Goto         -   -  121 119 - Print        3505 -   - 120 - PrintNl      -   -   - 121 - End          -   -   -  What option you want to use 0 = Binary Search 1 = Linear Search 0 Value to search: 5 Index: 4 </pre>
--	---

## Código

Lifecycle			
Driver main.rs	Tabla de var y funcs dir_func module	Código Intermedio quadruples module	Máquina Virtual vm module
Manda a llamar a todas las clases involucradas en el proceso de compilación y ejecución.	Llena las tablas de funciones y de variables. Y genera los directorios para cada función del programa.	Genera la lista de cuádruplos que funcionan como representación intermedia para el proceso de compilación. Toma en cuenta las acciones semánticas necesarias. Y genera la memoria constante y de punteros	Convierte la representación intermedia de un 3AC a representación interna la cuál es implementación de rust.

### src/address/mod.rs

```

use std::{cmp::Ordering, collections::HashMap, fmt};

use crate::{
    dir_func::{variable::Dimensions, variable_value::VariableValue},
    enums::Types,
    vm::VMResult,
};

const THRESHOLD: usize = 250;
const COUNTER_SIZE: usize = 4;
pub const TOTAL_SIZE: usize = THRESHOLD * COUNTER_SIZE;

```

```

pub trait Address {
    fn is_temp_address(&self) -> bool;
    fn is_pointer_address(&self) -> bool;
}

impl Address for usize {
    fn is_temp_address(&self) -> bool {
        TOTAL_SIZE * 2 < *self && *self < TOTAL_SIZE
    }

    fn is_pointer_address(&self) -> bool {
        *self >= TOTAL_SIZE * 4
    }
}

impl Address for Option<usize> {
    fn is_temp_address(&self) -> bool {

```

```

        match self {
            Some(address) =>
address.is_temp_address(),
            None => false,
        }
    }

    fn is_pointer_address(&self) -> bool {
        match self {
            Some(address) =>
address.is_pointer_address(),
            None => false,
        }
    }
}

type AddressCounter = HashMap<Types, usize>;

fn get_type_base(data_type: Types) -> usize {
    match data_type {
        Types::Int => 0,
        Types::Float => THRESHOLD,
        Types::String => THRESHOLD * 2,
        Types::Bool => THRESHOLD * 3,
        _ => unreachable!(),
    }
}

fn get_amount(dimensions: Dimensions) -> usize {
    let dim_1 = dimensions.0.unwrap_or(0);
    let dim_2 = dimensions.1.unwrap_or(1);
    match dim_1 * dim_2 {
        0 => 1,
        v => v,
    }
}

pub trait GenericAddressManager {
    fn get_address_counter(&self) -> &AddressCounter;
    fn get_address(&mut self, data_type: Types,
dimensions: Dimensions) -> Option<usize>;
    fn size(&self) -> usize;
    fn get_base(&self) -> usize;
}

#[derive(PartialEq, Clone)]
#[allow(clippy::module_name_repetitions)]
pub struct AddressManager {
    base: usize,
    counter: AddressCounter,
}

impl AddressManager {
    pub fn new(base: usize) -> Self {
        let counter = HashMap::from([
            (Types::Int, 0),
            (Types::Float, 0),
            (Types::String, 0),
            (Types::Bool, 0),
        ]);
        debug_assert_eq!(counter.len(),
COUNTER_SIZE);
        AddressManager { base, counter }
    }
}

impl GenericAddressManager for AddressManager {
    #[inline]
    fn get_address_counter(&self) -> &AddressCounter
    {
        &self.counter
    }

    fn get_address(&mut self, data_type: Types,
dimensions: Dimensions) -> Option<usize> {
        if data_type == Types::Dataframe {
            return Some(10_000);
        }
        let type_counter = self

```

```

        .counter
        .get_mut(&data_type)
        .unwrap_or_else(|| panic!("{:?}",
data_type));
        let prev = *type_counter;
        let amount = get_amount(dimensions);
        let new_counter = prev + amount;
        if new_counter > THRESHOLD {
            return None;
        }
        *type_counter = new_counter;
        let type_base = get_type_base(data_type);
        Some(self.base + prev + type_base)
    }

    #[inline]
    fn size(&self) -> usize {
        self.counter
            .iter()
            .map(|v| v.1)
            .copied()
            .reduce(|a, v| a + v)
            .unwrap_or(0)
    }

    #[inline]
    fn get_base(&self) -> usize {
        self.base
    }
}

impl fmt::Debug for AddressManager {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) ->
fmt::Result {
        let int_counter =
self.counter.get(&Types::Int).unwrap();
        let float_counter =
self.counter.get(&Types::Float).unwrap();
        let string_counter =
self.counter.get(&Types::String).unwrap();
        let bool_counter =
self.counter.get(&Types::Bool).unwrap();
        write!(
            f,
            "AddressManager({:?}, {:?}, {:?}, {:?})",
            int_counter, float_counter,
            string_counter, bool_counter
        )
    }
}

#[derive(PartialEq, Clone)]
pub struct TempAddressManager {
    address_manager: AddressManager,
    released: HashMap<Types, Vec<usize>>,
}

impl TempAddressManager {
    pub fn new() -> Self {
        let released = HashMap::from([
            (Types::Int, Vec::new()),
            (Types::Float, Vec::new()),
            (Types::String, Vec::new()),
            (Types::Bool, Vec::new()),
        ]);
        debug_assert_eq!(released.len(),
COUNTER_SIZE);
        TempAddressManager {
            address_manager:
AddressManager::new(TOTAL_SIZE * 2),
            released,
        }
    }

    fn address_type(&self, address: usize) -> Types {
        let contextless_address = address -
self.address_manager.base;
        let type_determinant = contextless_address /
THRESHOLD;
        match type_determinant {

```

```

0 => Types::Int,
1 => Types::Float,
2 => Types::String,
3 => Types::Bool,
_ => unreachable!(
    "{:?}", {:?}, {:?}"
    address, contextless_address,
type_determinant
),
}
}

#[inline]
fn type_released_addresses(&mut self, data_type:
&Types) -> &mut Vec<usize> {
    self.released.get_mut(data_type).unwrap()
}

pub fn release_address(&mut self, address: usize)
{
    let data_type = self.address_type(address);
self.type_released_addresses(&data_type).push(address
);
}

impl Default for TempAddressManager {
    fn default() -> Self {
        Self::new()
    }
}

impl GenericAddressManager for TempAddressManager {
    #[inline]
    fn get_address_counter(&self) -> &AddressCounter
    {
        self.address_manager.get_address_counter()
    }
    #[inline]
    fn get_address(&mut self, data_type: Types,
dimensions: Dimensions) -> Option<usize> {
        self.type_released_addresses(&data_type)
            .pop()
            .or_else(||
self.address_manager.get_address(data_type,
dimensions))
    }
    #[inline]
    fn size(&self) -> usize {
        self.address_manager.size()
    }
    #[inline]
    fn get_base(&self) -> usize {
        self.address_manager.base
    }
}

impl fmt::Debug for TempAddressManager {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) ->
fmt::Result {
        write!(f, "TempAddressManager({:#?})",
self.released)
    }
}

#[derive(PartialEq, Clone)]
pub struct ConstantMemory {
    base: usize,
    memory: HashMap<Types, Vec<VariableValue>>,
}

fn get_address_info(address: usize, base: usize) ->
(usize, usize, Types) {
    let contextless_address = address - base;
    let type_determinant = contextless_address /
THRESHOLD;
    let address_type = match type_determinant {

```

```

0 => Types::Int,
1 => Types::Float,
2 => Types::String,
3 => Types::Bool,
_ => unreachable!(),
};
(contextless_address, type_determinant,
address_type)
}

impl ConstantMemory {
    pub fn new() -> Self {
        let memory = HashMap::from([
            (Types::Int, vec![]),
            (Types::Float, vec![]),
            (Types::String, vec![]),
            (Types::Bool, vec![]),
        ]);
        ConstantMemory {
            base: TOTAL_SIZE * 3,
            memory,
        }
    }

    fn get_address(&mut self, data_type: Types,
value: VariableValue) -> Option<usize> {
        let type_memory = self
            .memory
            .get_mut(&data_type)
            .unwrap_or_else(|| panic!("Get address
received {:?}", data_type));
        let type_base = get_type_base(data_type);
        match type_memory.iter_mut().position(|x| *x
== value) {
            None => {
                if
type_memory.len().to_owned().cmp(&THRESHOLD) ==
Ordering::Equal {
                    return None;
                }
                let position = type_memory.len();
                type_memory.push(value);
                Some(self.base + position +
type_base)
            }
            Some(position) => Some(self.base +
position + type_base),
        }
    }

    pub fn add(&mut self, value: VariableValue) ->
Option<(usize, Types)> {
        let data_type = Types::from(&value);
        let address = self.get_address(data_type,
value)?;
        Some((address, data_type))
    }

    pub fn get(&self, address: usize) ->
&VariableValue {
        let (contextless_address, type_determinant,
address_type) =
            get_address_info(address, self.base);
        self.memory
            .get(&address_type)
            .unwrap()
            .get(contextless_address -
type_determinant * THRESHOLD)
            .unwrap()
    }
}

impl fmt::Debug for ConstantMemory {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) ->
fmt::Result {
        write!(f, "ConstantMemory({:#?})",
self.memory)
    }
}

```

```

}

#[derive(Clone, Debug)]
pub struct Memory {
    base: usize,
    int_pointer: usize,
    float_pointer: usize,
    string_pointer: usize,
    bool_pointer: usize,
    space: Vec<Option<VariableValue>>,
}

impl Memory {
    pub fn new<T: GenericAddressManager>(manager: &T)
    -> Self {
        let counter = manager.get_address_counter();
        let base = manager.get_base();
        let int_pointer: usize = 0;
        let float_pointer = int_pointer +
        counter.get(&Types::Int).unwrap();
        let string_pointer = float_pointer +
        counter.get(&Types::Float).unwrap();
        let bool_pointer = string_pointer +
        counter.get(&Types::String).unwrap();
        let total_size = bool_pointer +
        counter.get(&Types::Bool).unwrap();
        let space = vec![None; total_size];
        Memory {
            base,
            int_pointer,
            float_pointer,
            string_pointer,
            bool_pointer,
            space,
        }
    }

    fn get_index(&self, address: usize) -> (usize,
    Types) {
        let (contextless_address, _, address_type) =
        get_address_info(address, self.base);
        let type_index = contextless_address %
        THRESHOLD;
        let pointer = match address_type {
            Types::Int => self.int_pointer,
            Types::Float => self.float_pointer,
            Types::String => self.string_pointer,
            Types::Bool => self.bool_pointer,
            data_type => unreachable!("{:?}",
            data_type),
        };
        (type_index + pointer, address_type)
    }

    pub fn get(&self, address: usize) ->
    &Option<VariableValue> {
        let index = self.get_index(address).0;
        self.space.get(index).unwrap()
    }

    pub fn write(&mut self, address: usize, uncast:
    &VariableValue) -> VMResult<()> {
        let (index, address_type) =
        self.get_index(address);
        let value = uncast.cast_to(address_type)?;
        *self.space.get_mut(index).unwrap() =
        Some(value);
        Ok(())
    }
}

#[derive(Clone, PartialEq, Debug)]
pub struct PointerMemory {
    counter: usize,
    pointers: HashMap<usize, usize>,
}

impl PointerMemory {

```

```

    pub fn new() -> Self {
        Self {
            counter: TOTAL_SIZE * 4,
            pointers: HashMap::new(),
        }
    }

    pub fn get_pointer(&mut self) -> usize {
        let prev_counter = self.counter;
        self.counter += 1;
        prev_counter
    }

    pub fn write(&mut self, address: usize, var:
    VariableValue) {
        self.pointers.insert(address, var.into());
    }

    pub fn get(&self, address: usize) -> usize {
        self.pointers.get(&address).unwrap().to_owned()
    }
}

#[cfg(test)]
mod tests;

```

## src/address/tests.rs

```

use super::*;

#[test]
fn valid_get_address() {
    let mut address_manager = AddressManager::new(0);
    let address =
    address_manager.get_address(Types::Int, (None,
    None));
    assert_eq!(address, Some(0));
}

#[test]
fn invalid_get_address() {
    let mut address_manager = AddressManager::new(0);
    for i in 0..250 {
        let address =
        address_manager.get_address(Types::Int, (None,
        None));
        assert_eq!(address, Some(i));
    }
    let address =
    address_manager.get_address(Types::Int, (None,
    None));
    assert_eq!(address, None);
}

```

## src/args/mod.rs

```

use clap::{Arg, ArgMatches, Command};

pub fn parse_arguments() -> ArgMatches {
    Command::new("raoul")
        .version("1.0")
        .author("ricglz")
        .about("My cool programming language")
        .arg(
            Arg::new("file")
                .value_name("FILE")
                .help("Sets a file to parse")
                .required(true),
        )
        .arg(
            Arg::new("debug")
                .short('d')
                .long("debug")
                .value_name("DEBUG")
                .help("Displays debugging prints
                throughout the process")
                .default_value("false")

```

```

        .takes_value(false)
        .required(false),
    )
    .arg(
        Arg::new("quads")
            .short('q')
            .long("quads")
            .value_name("QUADS")
            .help("Displays quads generated by
the compiler")
            .default_value("false")
            .takes_value(false)
            .required(false),
    )
    .get_matches()
}

```

## src/ast/ast\_kind.rs

```

use super::{BoxedNode, Nodes};
use crate::{
    dir_func::variable::Dimensions,
    enums::{Operator, Types},
};
use std::fmt;

#[derive(PartialEq, Clone)]
pub enum AstNodeKind<'a> {
    Id(String),
    Integer(i64),
    Float(f64),
    String(String),
    Bool(bool),
    Array(Nodes<'a>),
    ArrayDeclaration {
        data_type: Types,
        dim1: usize,
        dim2: Option<usize>,
    },
    ArrayVal {
        name: String,
        idx_1: BoxedNode<'a>,
        idx_2: Option<BoxedNode<'a>>,
    },
    Assignment {
        assignee: BoxedNode<'a>,
        global: bool,
        value: BoxedNode<'a>,
    },
    UnaryOperation {
        operator: Operator,
        operand: BoxedNode<'a>,
    },
    BinaryOperation {
        operator: Operator,
        lhs: BoxedNode<'a>,
        rhs: BoxedNode<'a>,
    },
    Main {
        assignments: Nodes<'a>,
        body: Nodes<'a>,
        functions: Nodes<'a>,
    },
    Argument {
        arg_type: Types,
        name: String,
    },
    Function {
        arguments: Nodes<'a>,
        body: Nodes<'a>,
        name: String,
        return_type: Types,
    },
    Write(Nodes<'a>),
    Read,
    Decision {
        expr: BoxedNode<'a>,
        statements: Nodes<'a>,
    },
}

```

```

        else_block: Option<BoxedNode<'a>>,
    },
    ElseBlock(Nodes<'a>),
    While {
        expr: BoxedNode<'a>,
        statements: Nodes<'a>,
    },
    For {
        assignment: BoxedNode<'a>,
        expr: BoxedNode<'a>,
        statements: Nodes<'a>,
    },
    FuncCall {
        name: String,
        exprs: Nodes<'a>,
    },
    Return(BoxedNode<'a>),
    ReadCSV(BoxedNode<'a>),
    PureDataframeOp {
        name: String,
        operator: Operator,
    },
    UnaryDataframeOp {
        column: BoxedNode<'a>,
        name: String,
        operator: Operator,
    },
    Correlation {
        name: String,
        column_1: BoxedNode<'a>,
        column_2: BoxedNode<'a>,
    },
    Plot {
        name: String,
        column_1: BoxedNode<'a>,
        column_2: BoxedNode<'a>,
    },
    Histogram {
        column: BoxedNode<'a>,
        name: String,
        bins: BoxedNode<'a>,
    },
}

impl From<&AstNodeKind<'_>> for String {
    fn from(val: &AstNodeKind) -> Self {
        match val {
            AstNodeKind::Integer(n) => n.to_string(),
            AstNodeKind::Id(s) |
            AstNodeKind::String(s) => s.clone(),
            AstNodeKind::Assignment { assignee, .. }
            => assignee.into(),
            AstNodeKind::ArrayVal { name, .. } =>
                name.clone(),
            node => unreachable!("Node {:?}, cannot
be a string", node),
        }
    }
}

impl<'a> From<AstNodeKind<'a>> for usize {
    fn from(val: AstNodeKind) -> Self {
        match val {
            AstNodeKind::Integer(n) =>
                n.try_into().unwrap_or(0),
            node => unreachable!("{node:?}, cannot be
a usize"),
        }
    }
}

impl<'a> From<usize> for AstNodeKind<'a> {
    fn from(i: usize) -> Self {
        AstNodeKind::Integer(i.try_into().unwrap())
    }
}

impl fmt::Debug for AstNodeKind<'_> {

```

```

fn fmt(&self, f: &mut fmt::Formatter<'_>) ->
fmt::Result {
    match &self {
        Self::Id(s) => write!(f, "Id({})", s),
        Self::Integer(n) => write!(f,
"Integer({})", n,
        Self::Float(n) => write!(f, "Float({})",
n),
        Self::String(s) => write!(f,
"String({})", s),
        Self::Bool(s) => write!(f, "Bool({})",
s),
        Self::Array(s) => write!(f,
"Array({s:?})",
        Self::ArrayDeclaration {
            data_type,
            dim1,
            dim2,
        } => {
            write!(f,
"ArrayDeclaration({data_type:?}, {dim1}, {dim2:?})",
            Self::ArrayVal { name, idx_1, idx_2 } =>
{
            write!(f, "ArrayVal({name},
{idx_1:?}, {idx_2:?})",
            Self::Assignment {
                assignee,
                global,
                value,
            } => write!(f, "Assignment({}, {:?},
{:?})", global, assignee, value),
            Self::UnaryOperation {
                operator: operation,
                operand,
            } => {
                write!(f, "Unary({:?}, {:?})",
operation, operand)
            }
            Self::Main {
                assignments,
                body,
                functions,
            } => write!(f, "Main(({assignments:#?},
{:#?}, {:#?}))", functions, body),
            Self::Argument { arg_type, name } =>
write!(f, "Argument({:?}, {})", arg_type, name),
            Self::Function {
                arguments,
                body,
                name,
                return_type,
            } => {
                write!(
                    f,
                    "Function({}, {:?}, {:?},
                    name, return_type, arguments,
                    body
                )
            }
            Self::Write(exprs) => write!(f,
"Write({:?})", exprs),
            Self::Read => write!(f, "Read"),
            Self::BinaryOperation { operator, lhs,
rhs } => {
                write!(f, "BinaryOperation({:?},
{:?}, {:?})", operator, lhs, rhs)
            }
            Self::Decision {
                expr,
                statements,
                else_block,
            } => {
                write!(f, "Decision({expr:?},
{statements:?}, {else_block:?})",
            }
        }
    }
}

```

```

        Self::ElseBlock(statements) => write!(f,
"ElseBlock({:?})", statements),
        Self::While { expr, statements } =>
write!(f, "While({:?}, {:?})", expr, statements),
        Self::For {
            expr,
            statements,
            assignment,
        } => {
            write!(f, "For({expr:?},
{statements:?}, {assignment:?})",
            Self::FuncCall { name, exprs } =>
write!(f, "FunctionCall({name}, {exprs:?})",
            Self::Return(expr) => write!(f,
"Return({expr:?})",
            Self::ReadCSV(file) => write!(f,
"ReadCSV({file:?})",
            Self::PureDataframeOp { name, operator }
=> {
                write!(f,
"PureDataframeOp({operator:?}, {name})",
            }
            Self::UnaryDataframeOp {
                operator,
                name,
                column,
            } => {
                write!(f,
"UnaryDataframeOp({operator:?}, {name}, {column:?})",
            }
            Self::Correlation {
                name,
                column_1,
                column_2,
            } => {
                write!(f, "Correlation({name},
{column_1:?}, {column_2:?})",
            }
            Self::Plot {
                name,
                column_1,
                column_2,
            } => write!(f, "Plot({name},
{column_1:?}, {column_2:?})",
            Self::Histogram { column, name, bins } =>
{
                write!(f, "Histogram({column:?},
{name}, {bins:?})",
            }
        }
    }
}

impl<'a> AstNodeKind<'a> {
    pub fn is_array(&self) -> bool {
        matches!(self, Self::Array(_) |
Self::ArrayDeclaration { .. })
    }

    pub fn is_declaration(&self) -> bool {
        matches!(self, Self::Assignment { .. } |
Self::Argument { .. })
    }

    pub fn get_dimensions(&self) ->
Result<Dimensions, Dimensions> {
        if !self.is_array() {
            return Ok((None, None));
        }
        match self {
            Self::ArrayDeclaration { dim1, dim2, .. }
=> Ok((Some(*dim1), *dim2)),
            Self::Array(exprs) => {
                let dim1 = Some(exprs.len());
                let dim2 =
exprs.get(0).unwrap().get_dimensions().?0;
                let errors: Vec<_> = exprs

```

```

        .iter()
        .map(|expr| {
            let expr_dim_1 =
expr.get_dimensions()?.0;
            if expr_dim_1 == dim2 {
                Ok(())
            } else {
                Err((expr_dim_1, dim2))
            }
        })
        .filter_map(Result::err)
        .collect();
    if errors.is_empty() {
        Ok((dim1, dim2))
    } else {
        Err(*errors.get(0).unwrap())
    }
}
_ => unreachable!("{self:?}"),
}
}
}

```

## src/ast/mod.rs

```

#[allow(clippy::module_name_repetitions)]
pub mod ast_kind;

use crate::dir_func::variable::Dimensions;

use self::ast_kind::AstNodeKind;
use pest::Span;
use std::fmt;

#[derive(PartialEq, Clone)]
#[allow(clippy::module_name_repetitions)]
pub struct AstNode<'a> {
    pub kind: AstNodeKind<'a>,
    pub span: Span<'a>,
}

impl<'a> From<&AstNode<'a>> for String {
    fn from(val: &AstNode) -> Self {
        Self::from(&val.kind)
    }
}

impl<'a> From<AstNode<'a>> for String {
    fn from(val: AstNode) -> Self {
        Self::from(&val)
    }
}

impl From<Box<AstNode<'_>>> for String {
    fn from(val: Box<AstNode>) -> Self {
        String::from(*val)
    }
}

impl From<&Box<AstNode<'_>>> for String {
    fn from(val: &Box<AstNode>) -> Self {
        String::from(*val.clone())
    }
}

impl<'a> From<AstNode<'a>> for usize {
    fn from(val: AstNode) -> Self {
        val.kind.into()
    }
}

impl<'a> AstNode<'a> {
    pub fn expand_node(v: &AstNode<'a>) -> Nodes<'a>
    {
        match &v.kind {
            AstNodeKind::Decision { statements, .. }
            | AstNodeKind::ElseBlock(statements)

```

```

            | AstNodeKind::While { statements, .. }
=> {
    statements.iter().flat_map(AstNode::expand_node).collect()
}
AstNodeKind::For {
    statements,
    assignment,
    ..
} => vec![*assignment.clone()]
.iter()
.chain(statements)
.flat_map(AstNode::expand_node)
.collect(),
_ => vec![v.clone()],
}
}

pub fn expand_array(&self) -> &Nodes<'a> {
    match &self.kind {
        AstNodeKind::Array(exprs) => exprs,
        _ => unreachable!(),
    }
}

pub fn new(kind: AstNodeKind<'a>, span:
&Span<'a>) -> AstNode<'a> {
    AstNode {
        kind,
        span: span.clone(),
    }
}

pub fn is_array(&self) -> bool {
    self.kind.is_array()
}

pub fn is_declaration(&self) -> bool {
    self.kind.is_declaration()
}

pub fn get_dimensions(&self) ->
Result<Dimensions, Dimensions> {
    self.kind.get_dimensions()
}
}

impl fmt::Debug for AstNode<'_> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) ->
fmt::Result {
        write!(f, "{:?}", self.kind)
    }
}

pub type BoxedNode<'a> = Box<AstNode<'a>>;
pub type Nodes<'a> = Vec<AstNode<'a>>;

```

## src/dir\_func/function.rs

```

use std::collections::HashMap;

use crate::{
    address::{AddressManager, GenericAddressManager,
TempAddressManager, TOTAL_SIZE},
    ast::ast_kind::AstNodeKind,
    ast::AstNode,
    enums::Types,
    error::{error_kind::RaoulErrorKind, RaoulError,
Results},
    quadruple::quadruple_manager::Operand,
};

use super::variable::{Dimensions, Variable};

pub type VariablesTable = HashMap<String, Variable>;
type InsertResult = std::result::Result<(),
RaoulErrorKind>;

```

```

pub trait Scope {
    fn get_variable(&self, name: &str) ->
Option<&Variable>;
    fn _insert_variable(&mut self, name: String,
variable: Variable);
    fn insert_variable(&mut self, variable: Variable)
-> InsertResult {
        let name = variable.name.clone();
        if let Some(stored_var) =
self.get_variable(&name) {
            if
!variable.data_type.can_cast(stored_var.data_type) {
                return
Err(RaoulErrorKind::RedefinedType {
                    name,
                    from: stored_var.data_type,
                    to: variable.data_type,
                });
            }
        } else {
            self._insert_variable(name, variable);
        }
        Ok(())
    }
    fn _get_variable_address(&mut self, data_type:
Types, dimensions: Dimensions) -> Option<usize>;
    fn get_variable_address(
        &mut self,
        name: &str,
        data_type: Types,
        dimensions: Dimensions,
    ) -> Option<usize> {
        match self.get_variable(name) {
            Some(variable) => Some(variable.address),
            None =>
self._get_variable_address(data_type, dimensions),
        }
    }
}

#[derive(PartialEq, Clone, Debug)]
pub struct Function {
    pub address: usize,
    pub args: Vec<Operand>,
    pub first_quad: usize,
    pub local_addresses: AddressManager,
    pub name: String,
    pub return_type: Types,
    pub temp_addresses: TempAddressManager,
    pub variables: VariablesTable,
}

impl Function {
    fn new(name: String, return_type: Types) -> Self
    {
        Self {
            address: usize::MAX,
            args: Vec::new(),
            local_addresses:
AddressManager::new(TOTAL_SIZE),
            name,
            return_type,
            temp_addresses:
TempAddressManager::new(),
            variables: HashMap::new(),
            first_quad: 0,
        }
    }

    fn insert_variable_from_node<'a>(
        &mut self,
        node: &AstNode<'a>,
        global_fn: &mut GlobalScope,
        argument: bool,
    ) -> Results<'a, ()> {
        match Variable::from_node(node, self,
global_fn) {
            Ok((variable, global)) => {
                let address = variable.address;
                let data_type = variable.data_type;
                let result = if global {
                    global_fn.insert_variable(variable)
                } else {
                    self.insert_variable(variable)
                };
                if let Err(kind) = result {
                    return
Err(RaoulError::new_vec(node, kind));
                }
                if argument {
                    self.args.push((address,
data_type));
                }
                Ok(())
            }
            Err(errors) => Err(errors),
        }
    }

    fn insert_from_nodes<'a>(
        &mut self,
        nodes: &[AstNode<'a>],
        global_fn: &mut GlobalScope,
        is_arg: bool,
    ) -> Results<'a, ()> {
        RaoulError::create_results(
            nodes
                .iter()
                .flat_map(AstNode::expand_node)
                .filter(AstNode::is_declaration)
                .map(|node|
self.insert_variable_from_node(&node, global_fn,
is_arg)),
        )
    }

    pub fn try_create<'a>(v: &AstNode<'a>, global_fn:
&mut GlobalScope) -> Results<'a, Function> {
        match v.kind.clone() {
            AstNodeKind::Function {
                name,
                return_type,
                ref body,
                ref arguments,
            } => {
                let mut function =
Function::new(name, return_type);
                function.insert_from_nodes(arguments,
global_fn, true)?;
                function.insert_from_nodes(body,
global_fn, false)?;
                Ok(function)
            }
            AstNodeKind::Main { ref body, .. } => {
                let mut function =
Function::new("main".to_string(), Types::Void);
                function.insert_from_nodes(body,
global_fn, false)?;
                Ok(function)
            }
            _ => unreachable!(),
        }
    }

    pub fn size(&self) -> usize {
        self.local_addresses.size() +
self.temp_addresses.size()
    }

    pub fn update_quad(&mut self, first_quad: usize)
    {
        self.first_quad = first_quad;
    }
}

```



```

impl Scope for Function {
    fn get_variable(&self, name: &str) ->
Option<&Variable> {
        self.variables.get(name)
    }
    fn _insert_variable(&mut self, name: String,
variable: Variable) {
        self.variables.insert(name, variable);
    }
    fn _get_variable_address(&mut self, data_type:
Types, dimensions: Dimensions) -> Option<usize> {
        self.local_addresses.get_address(data_type,
dimensions)
    }
}

#[derive(PartialEq, Clone, Debug)]
pub struct GlobalScope {
    has_dataframe: bool,
    pub addresses: AddressManager,
    pub variables: VariablesTable,
}

impl GlobalScope {
    pub fn new() -> Self {
        Self {
            addresses: AddressManager::new(0),
            variables: HashMap::new(),
            has_dataframe: false,
        }
    }

    pub fn add_dataframe(&mut self) -> bool {
        if self.has_dataframe {
            false
        } else {
            self.has_dataframe = true;
            true
        }
    }
}

impl Default for GlobalScope {
    fn default() -> Self {
        Self::new()
    }
}

impl Scope for GlobalScope {
    fn get_variable(&self, name: &str) ->
Option<&Variable> {
        self.variables.get(name)
    }
    fn _insert_variable(&mut self, name: String,
variable: Variable) {
        self.variables.insert(name, variable);
    }
    fn _get_variable_address(&mut self, data_type:
Types, dimensions: Dimensions) -> Option<usize> {
        self.addresses.get_address(data_type,
dimensions)
    }
}

```

## src/dir\_func/mod.rs

```

use std::collections::HashMap;

use crate::{
    address::GenericAddressManager,
    ast::ast_kind::AstNodeKind,
    ast::AstNode,
    enums::Types,
    error::{error_kind::RaoulErrorKind, RaoulError,
Result, Results},
};

```

```

use self::{
    function::{Function, GlobalScope, Scope},
    variable::Variable,
};

pub mod function;
pub mod variable;
pub mod variable_value;

pub type FunctionTable = HashMap<String, Function>;

#[derive(PartialEq, Debug, Clone)]
pub struct DirFunc {
    pub functions: FunctionTable,
    pub global_fn: GlobalScope,
}

impl DirFunc {
    pub fn new() -> Self {
        Self {
            global_fn: GlobalScope::new(),
            functions: HashMap::new(),
        }
    }

    pub fn clear_variables(&mut self) {
        self.global_fn.variables.clear();
        self.functions
            .values_mut()
            .for_each(|f| f.variables.clear());
    }

    fn insert_function<'a>(&mut self, function:
Function, node: &AstNode<'a>) -> Result<'a, ()> {
        let name = function.name.clone();
        match self.functions.get(&name) {
            Some(_) => Err(RaoulError::new(
                node,
                RaoulErrorKind::RedeclaredFunction(name),
            )),
            None => {
                self.functions.insert(name,
function);
                Ok(())
            }
        }
    }

    fn insert_function_from_node<'a>(&mut self, node:
&AstNode<'a>) -> Results<'a, ()> {
        let mut function = Function::try_create(node,
&mut self.global_fn)?;
        if function.return_type != Types::Void {
            let address = self
                .global_fn
                .addresses
                .get_address(function.return_type,
(None, None));
            match address {
                Some(address) => {
                    let result = self
                        .global_fn
                        .insert_variable(Variable::from_function(&function,
address));

                    if let Err(kind) = result {
                        return
Err(vec![RaoulError::new(node, kind)]);
                    }
                    function.address = address;
                }
                None => {
                    let kind =
RaoulErrorKind::MemoryExceeded;
                    return
Err(vec![RaoulError::new(node, kind)]);
                }
            }
        }
    }
}

```

```

    }
    match self.insert_function(function, node) {
        Ok(_) => Ok(()),
        Err(error) => Err(vec![error]),
    }
}

pub fn build_dir_func<'a>(&mut self, node:
&AstNode<'a>) -> Results<'a, ()> {
    match &node.kind {
        AstNodeKind::Main {
            functions,
            assignments,
            ..
        } => {
RaoulError::create_results(assignments.iter().map(|no
de| -> Results<()> {
    let variable =
Variable::from_global(node, &mut self.global_fn)?;
    match
self.global_fn.insert_variable(variable) {
        Ok(_) => Ok(()),
        Err(kind) =>
Err(RaoulError::new_vec(node, kind)),
    }
    )))?;
    RaoulError::create_results(
        functions
        .iter()
        .chain(Some(node))
        .map(|node|
self.insert_function_from_node(node)),
    )
    }
    _ => unreachable!(),
}
}
}

```

## src/dir\_func/variable.rs

```

use crate::{
    address::GenericAddressManager,
    ast::ast_kind::AstNodeKind,
    ast::AstNode,
    enums::Types,
    error::error_kind::RaoulErrorKind,
    error::{RaoulError, Results},
};

use super::function::{Function, GlobalScope, Scope};

pub type Dimensions = (Option<usize>, Option<usize>);

#[derive(Clone, PartialEq, Debug)]
pub struct Variable {
    pub address: usize,
    pub data_type: Types,
    pub dimensions: Dimensions,
    pub name: String,
}

fn get_value_dimensions<'a>(value: &AstNode<'a>,
node: &AstNode<'a>) -> Results<'a, Dimensions> {
    match value.get_dimensions() {
        Ok(dimensions) => Ok(dimensions),
        Err((expected, given)) => {
            let kind =
RaoulErrorKind::InconsistentSize { expected, given };
            Err(RaoulError::new_vec(node, kind))
        }
    }
}

fn assert_dataframe<'a>(
    data_type: Types,

```

```

    global_fn: &mut GlobalScope,
    node: &AstNode<'a>,
) -> Results<'a, ()> {
    if data_type != Types::Dataframe {
        return Ok(());
    }
    if global_fn.add_dataframe() {
        Ok(())
    } else {
        Err(RaoulError::new_vec(node,
RaoulErrorKind::OnlyOneDataframe))
    }
}

impl Variable {
    pub fn from_global<'a>(v: &AstNode<'a>,
global_fn: &mut GlobalScope) -> Results<'a, Variable>
    {
        match &v.kind {
            AstNodeKind::Assignment {
                assignee, value, ..
            } => {
                let data_type =
                    Types::from_node(&*value,
&global_fn.variables, &global_fn.variables)?;
                assert_dataframe(data_type,
global_fn, v)?;
                let dimensions =
get_value_dimensions(value, v)?;
                let name: String = assignee.into();
                match
global_fn.get_variable_address(&name, data_type,
dimensions) {
                    Some(address) => Ok(Variable {
                        address,
                        data_type,
                        dimensions,
                        name,
                    }),
                    None =>
Err(RaoulError::new_vec(v,
RaoulErrorKind::MemoryExceeded)),
                }
            }
            _ => unreachable!("(kind:?)"),
        }
    }

    pub fn from_node<'a>(
        v: &AstNode<'a>,
        current_fn: &mut Function,
        global_fn: &mut GlobalScope,
    ) -> Results<'a, (Variable, bool)> {
        match v.kind.clone() {
            AstNodeKind::Assignment {
                assignee,
                value,
                global,
            } => {
                let data_type =
                    Types::from_node(&*value,
&current_fn.variables, &global_fn.variables)?;
                assert_dataframe(data_type,
global_fn, v)?;
                let dimensions =
get_value_dimensions(&value, v)?;
                let name: String = assignee.into();
                let address = if global {
                    global_fn.get_variable_address(&name, data_type,
dimensions)
                } else {
                    current_fn.get_variable_address(&name, data_type,
dimensions)
                };
                match address {
                    Some(address) => Ok((

```

```

        Variable {
            address,
            data_type,
            dimensions,
            name,
        },
        global,
    )),
    None =>
Err(RaoulError::new_vec(v,
RaoulErrorKind::MemoryExceded)),
}
}
AstNodeKind::Argument {
    arg_type: data_type,
    name,
} => {
    let address = current_fn
        .local_addresses
        .get_address(data_type, (None,
None));
    match address {
        Some(address) => Ok((
            Variable {
                address,
                data_type,
                name,
                dimensions: (None, None),
            },
            false,
        )),
        None => {
            let kind =
RaoulErrorKind::MemoryExceded;
            Err(RaoulError::new_vec(v,
kind))
        }
    }
    _ => unreachable!(),
}
}
}

pub fn from_function(function: &Function,
address: usize) -> Self {
    Variable {
        address,
        data_type: function.return_type,
        name: function.name.clone(),
        dimensions: (None, None),
    }
}
}
}

```

## src/dir\_func/variable\_value.rs

```

use std::fmt;
use std::io::stdin;
use std::ops::{Add, BitAnd, BitOr, Div, Mul, Not,
Sub};

use crate::vm::VMResult;
use crate::{ast::ast_kind::AstNodeKind,
enums::Types};

#[derive(Clone, PartialEq)]
pub enum VariableValue {
    Integer(i64),
    Float(f64),
    String(String),
    Bool(bool),
}

impl VariableValue {
    pub fn from_stdin() -> Self {
        let mut line = String::new();
        stdin().read_line(&mut line).unwrap();
        Self::String(line.replace("\n", ""))
    }
}

```

```

    }

    pub fn is_number(&self) -> bool {
        matches!(self, Self::Integer(_) |
Self::Float(_) | Self::String(_))
    }

    pub fn is_boolish(&self) -> bool {
        matches!(self, Self::Integer(_) |
Self::Bool(_))
    }

    #[inline]
    fn cast_to_bool(&self) -> VariableValue {
        Self::Bool(bool::from(self))
    }

    #[inline]
    fn cast_to_float(&self) ->
VMResult<VariableValue> {
        Ok(Self::Float(f64::try_from(self)?))
    }

    #[inline]
    fn cast_to_int(&self) -> VMResult<VariableValue>
{
        Ok(Self::Integer(i64::try_from(self)?))
    }

    pub fn cast_to(&self, to: Types) ->
VMResult<VariableValue> {
        match to {
            Types::Bool => Ok(self.cast_to_bool()),
            Types::Float => self.cast_to_float(),
            Types::Int => self.cast_to_int(),
            _ => Ok(self.clone()),
        }
    }

    pub fn increase(&self) -> VMResult<Self> {
        match self {
            Self::Integer(v) => Ok(Self::Integer(v +
1)),
            v => {
                if v.is_number() {
                    self.cast_to_float()? +
Self::Float(1.0)
                } else {
                    unreachable!()
                }
            }
        }
    }
}
}

```

```

impl From<&VariableValue> for Types {
    fn from(v: &VariableValue) -> Self {
        match v {
            VariableValue::Integer(_) => Types::Int,
            VariableValue::Float(_) => Types::Float,
            VariableValue::String(_) =>
Types::String,
            VariableValue::Bool(_) => Types::Bool,
        }
    }
}

impl From<&AstNodeKind<'>> for VariableValue {
    fn from(v: &AstNodeKind) -> Self {
        match v {
            AstNodeKind::Integer(value) =>
VariableValue::Integer(*value),
            AstNodeKind::Float(value) =>
VariableValue::Float(*value),
            AstNodeKind::String(value) =>
VariableValue::String(value.clone()),
            AstNodeKind::Bool(value) =>
VariableValue::Bool(*value),
        }
    }
}

```

```

        _ => unreachable!(),
    }
}

impl From<AstNodeKind<'_>> for VariableValue {
    fn from(v: AstNodeKind) -> Self {
        Self::from(&v)
    }
}

impl TryFrom<&VariableValue> for f64 {
    type Error = &'static str;

    fn try_from(v: &VariableValue) -> VMResult<Self> {
        {
            if let VariableValue::Float(a) = v {
                return Ok(*a);
            }
            let string = match v {
                VariableValue::Integer(a) =>
                    a.to_string(),
                VariableValue::String(a) => a.clone(),
                _ => unreachable!(),
            };
            match string.parse:::<Self>() {
                Ok(a) => Ok(a),
                Err(_) => {
                    println!("Given: {string}");
                    Err("Could not parse to float")
                }
            }
        }
    }
}

impl TryFrom<VariableValue> for f64 {
    type Error = &'static str;

    fn try_from(v: VariableValue) -> VMResult<Self> {
        Self::try_from(&v)
    }
}

impl From<f64> for VariableValue {
    fn from(v: f64) -> Self {
        Self::Float(v)
    }
}

impl From<VariableValue> for bool {
    fn from(v: VariableValue) -> Self {
        match v {
            VariableValue::Integer(a) => a != 0,
            VariableValue::Bool(a) => a,
            _ => unreachable!(),
        }
    }
}

impl From<&VariableValue> for bool {
    fn from(v: &VariableValue) -> Self {
        Self::from(v.clone())
    }
}

impl From<usize> for VariableValue {
    fn from(v: usize) -> Self {
        Self::Integer(v.try_into().unwrap())
    }
}

impl From<VariableValue> for usize {
    fn from(v: VariableValue) -> Self {
        match v {
            VariableValue::Integer(v) =>
                v.try_into().unwrap(),
            _ => unreachable!(),
        }
    }
}

```

```

    }
}

impl From<VariableValue> for String {
    fn from(v: VariableValue) -> Self {
        match v {
            VariableValue::String(v) => v,
            _ => unreachable!(),
        }
    }
}

impl TryFrom<VariableValue> for i64 {
    type Error = &'static str;

    fn try_from(v: VariableValue) -> VMResult<Self> {
        {
            if let VariableValue::Integer(a) = &v {
                return Ok(*a);
            }
            if let VariableValue::Bool(a) = &v {
                return match a {
                    true => Ok(1),
                    false => Ok(0),
                };
            }
            let string = match v {
                VariableValue::Float(a) =>
                    a.floor().to_string(),
                VariableValue::String(a) => a,
                _ => unreachable!(),
            };
            match string.parse:::<Self>() {
                Ok(a) => Ok(a),
                Err(_) => {
                    println!("Given: {string}");
                    Err("Could not parse to int")
                }
            }
        }
    }
}

impl TryFrom<&VariableValue> for i64 {
    type Error = &'static str;

    fn try_from(v: &VariableValue) -> VMResult<Self> {
        {
            Self::try_from(v.clone())
        }
    }
}

impl fmt::Debug for VariableValue {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) ->
        fmt::Result {
        let value = match self {
            VariableValue::Bool(value) =>
                value.to_string(),
            VariableValue::Integer(value) =>
                value.to_string(),
            VariableValue::Float(value) =>
                value.to_string(),
            VariableValue::String(value) =>
                value.clone(),
        };
        write!(f, "{}", value)
    }
}

impl Add for VariableValue {
    type Output = VMResult<Self>;

    fn add(self, other: Self) -> Self::Output {
        if let (Self::Integer(a), Self::Integer(b)) =
            (self.clone(), other.clone()) {
            Ok(Self::Integer(a + b))
        } else {
            Ok(Self::Float(f64::try_from(self)? +
                f64::try_from(other)?))
        }
    }
}

```

```

    }
}

impl Sub for VariableValue {
    type Output = VMResult<Self>;

    fn sub(self, other: Self) -> Self::Output {
        if let (Self::Integer(a), Self::Integer(b)) =
            (self.clone(), other.clone()) {
            Ok(Self::Integer(a - b))
        } else {
            Ok(Self::Float(f64::try_from(self)? -
                f64::try_from(other)?))
        }
    }
}

impl Mul for VariableValue {
    type Output = VMResult<Self>;

    fn mul(self, other: Self) -> Self::Output {
        if let (Self::Integer(a), Self::Integer(b)) =
            (self.clone(), other.clone()) {
            Ok(Self::Integer(a * b))
        } else {
            Ok(Self::Float(f64::try_from(self)? *
                f64::try_from(other)?))
        }
    }
}

impl Div for VariableValue {
    type Output = VMResult<Self>;

    fn div(self, other: Self) -> Self::Output {
        if let (Self::Integer(a), Self::Integer(b)) =
            (self.clone(), other.clone()) {
            match b {
                0 => Err("Attempt to divide by
zero"),
                b => Ok(Self::Integer(a / b)),
            }
        } else {
            match (f64::try_from(self)?,
                f64::try_from(other)?) {
                (_, b) if b == 0.0 => Err("Attempt to
divide by zero"),
                (a, b) => Ok(Self::Float(a / b)),
            }
        }
    }
}

impl PartialOrd for VariableValue {
    fn partial_cmp(&self, other: &Self) ->
        Option<std::cmp::Ordering> {
        match (self.is_number(), other.is_number()) {
            (true, true) => match
                (f64::try_from(self), f64::try_from(other)) {
                    (Ok(a), Ok(b)) => a.partial_cmp(&b),
                    _ => None,
                },
            _ => match (self.is_boolish(),
                other.is_boolish()) {
                    (true, true) =>
                        bool::from(self).partial_cmp(&bool::from(other)),
                    _ => None,
                },
        }
    }
}

impl BitOr for VariableValue {
    type Output = Self;

    fn bitor(self, other: Self) -> Self::Output {
        Self::Bool(bool::from(self) |
            bool::from(other))
    }
}

```

```

    }
}

impl BitAnd for VariableValue {
    type Output = Self;

    fn bitand(self, other: Self) -> Self::Output {
        Self::Bool(bool::from(self) &
            bool::from(other))
    }
}

impl Not for VariableValue {
    type Output = Self;

    fn not(self) -> Self::Output {
        Self::Bool(!bool::from(self))
    }
}

```

## src/enums/mod.rs

```

use core::fmt;

use crate::ast::ast_kind::AstNodeKind;
use crate::ast::AstNode;
use crate::dir_func::function::VariablesTable;
use crate::dir_func::variable::Variable;
use crate::error::error_kind::RaoulErrorKind;
use crate::error::{RaoulError, Results};

#[derive(Clone, Copy, PartialEq, Debug, Hash, Eq)]
pub enum Types {
    Int,
    Void,
    Float,
    String,
    Bool,
    Dataframe,
}

impl Types {
    #[inline]
    pub fn is_boolish(self) -> bool {
        matches!(self, Types::Int | Types::Bool)
    }

    #[inline]
    fn is_number(self) -> bool {
        matches!(self, Types::Int | Types::Float |
            Types::String)
    }

    pub fn can_cast(self, to: Types) -> bool {
        if self.is_number() && to.is_number() {
            return true;
        }
        if self.is_boolish() && to.is_boolish() {
            return true;
        }
        self == to
    }

    pub fn assert_cast<'a>(self, to: Types, node:
        &AstNode<'a>) -> Results<'a, ()> {
        if self.can_cast(to) {
            return Ok(());
        }
        let error = RaoulError::new_vec(node,
            RaoulErrorKind::InvalidCast { from: self, to });
        Err(error)
    }

    pub fn binary_operator_type(
        self,
        operator: Operator,
        rhs_type: Types,
    ) -> Result<Types, (Types, Types)> {

```

```

        match operator {
            Operator::Not | Operator::Or |
Operator::And => {
            let type_res = Types::Bool;
            match (self.is_boolish(),
rhs_type.is_boolish()) {
                (true, true) => Ok(type_res),
                (true, false) => Err((rhs_type,
type_res)),
                _ => Err((self, type_res)),
            }
            Operator::Gte | Operator::Lte |
Operator::Gt | Operator::Lt => {
            let type_res = Types::Bool;
            match (self.is_number(),
rhs_type.is_number()) {
                (true, true) => Ok(type_res),
                (true, false) => Err((rhs_type,
type_res)),
                _ => Err((self, type_res)),
            }
            Operator::Eq | Operator::Ne => {
            if self.can_cast(rhs_type) {
                return Ok(Types::Bool);
            }
            Err((self, rhs_type))
        }
        Operator::Sum | Operator::Minus |
Operator::Times | Operator::Div => {
            if self == rhs_type && self ==
Types::Int {
                return Ok(Types::Int);
            }
            let type_res = Types::Float;
            match (self.is_number(),
rhs_type.is_number()) {
                (true, true) => Ok(type_res),
                (true, false) => Err((rhs_type,
type_res)),
                _ => Err((self, type_res)),
            }
            _ => unreachable!("{:?}" , operator),
        }
    }

    pub fn assert_bin_op<'a>(
        self,
        operator: Operator,
        rhs_type: Types,
        node: &AstNode<'a>,
    ) -> Results<'a, Types> {
        match self.binary_operator_type(operator,
rhs_type) {
            Ok(data_type) => Ok(data_type),
            Err((from, to)) =>
Err(RaoulError::new_vec(
                node,
                RaoulErrorKind::InvalidCast { from,
to },
            )),
        }
    }

    #[inline]
    fn get_variable<'a>(
        name: &str,
        variables: &'a VariablesTable,
        global: &'a VariablesTable,
    ) -> Option<&'a Variable> {
        variables.get(name).or_else(||
global.get(name))
    }

    pub fn from_node<'a>(
        v: &AstNode<'a>,

```

```

        variables: &VariablesTable,
        global: &VariablesTable,
    ) -> Results<'a, Types> {
        match &v.kind {
            AstNodeKind::Integer(_) |
AstNodeKind::PureDataframeOp { .. } =>
Ok(Types::Int),
            AstNodeKind::Float(_)
| AstNodeKind::UnaryDataframeOp { .. }
| AstNodeKind::Correlation { .. } =>
Ok(Types::Float),
            AstNodeKind::String(_) |
AstNodeKind::Read => Ok(Types::String),
            AstNodeKind::Bool(_) => Ok(Types::Bool),
            AstNodeKind::Id(name) |
AstNodeKind::ArrayVal { name, .. } => {
                match Types::get_variable(name,
variables, global) {
                    Some(variable) =>
Ok(variable.data_type),
                    None => Err(RaoulError::new_vec(
                        v,
RaoulErrorKind::UndeclaredVar(name.to_string()),
                    )),
                }
            }
            AstNodeKind::FuncCall { name, .. } => {
                match Types::get_variable(name,
variables, global) {
                    Some(variable) =>
Ok(variable.data_type),
                    None => Err(RaoulError::new_vec(
                        v,
RaoulErrorKind::UndeclaredFunction(name.to_string()),
                    )),
                }
            }
            AstNodeKind::ArrayDeclaration {
data_type, .. } => Ok(*data_type),
            AstNodeKind::Array(exprs) => {
                let types =
RaoulError::create_partition(
                    exprs
                        .iter()
                        .map(|node|
Types::from_node(node, variables, global)),
                );
                let first_type =
*(types.get(0).unwrap());
                RaoulError::create_results(
                    types
                        .into_iter()
                        .zip(exprs)
                        .map(|(data_type, node)|
data_type.assert_cast(first_type, node)),
                );
                Ok(first_type)
            }
            AstNodeKind::BinaryOperation { operator,
lhs, rhs } => {
                let lhs_type =
Types::from_node(&lhs, variables, global)?;
                let rhs_type =
Types::from_node(&rhs, variables, global)?;
                lhs_type.assert_bin_op(*operator,
rhs_type, v)
            }
            AstNodeKind::UnaryOperation { operator,
operand } => match operator {
                Operator::Not => {
                    let operand_type =
Types::from_node(&operand, variables, global)?;
                    let res_type = Types::Bool;
                    operand_type.assert_cast(res_type, v)?;
                    Ok(res_type)
                }

```

```

        }
        _ => unreachable!("{:?}", operator),
    },
    AstNodeKind::ReadCSV(_) =>
Ok(Self::Dataframe),
    kind => unreachable!("{kind:?}"),
}
}

#[derive(Clone, Copy, PartialEq, Debug, Hash, Eq)]
pub enum Operator {
    // Boolean
    Not,
    Or,
    And,
    // Relational
    Gte,
    Lte,
    Gt,
    Lt,
    // Equality
    Eq,
    Ne,
    // Arithmetic
    Sum,
    Minus,
    Times,
    Div,
    Inc,
    // ByteCode
    Assignment,
    Print,
    PrintNl,
    Read,
    Goto,
    GotoF,
    End,
    // Functions
    Return,
    EndProc,
    Era,
    GoSub,
    Param,
    // Arrays
    Ver,
    // Dataframe
    Rows,
    Columns,
    Average,
    Std,
    Median,
    Variance,
    Min,
    Max,
    Range,
    Corr,
    ReadCSV,
    Plot,
    Histogram,
}

impl Operator {
    pub fn is_goto(self) -> bool {
        matches!(self, Operator::Goto |
Operator::GotoF)
    }
}

impl fmt::Display for Operator {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) ->
fmt::Result {
        write!(f, "{:10}", format!("{:?}", self))
    }
}

```

## src/error/error\_kind.rs

```

use core::fmt;

use crate::enums::Types;

#[derive(PartialEq, Eq, Clone)]
#[allow(clippy::module_name_repetitions)]
pub enum RaoulErrorKind {
    MemoryExceded,
    UndeclaredVar(String),
    UndeclaredFunction(String),
    UndeclaredFunction2(String),
    RedeclaredFunction(String),
    RedefinedType {
        name: String,
        from: Types,
        to: Types,
    },
    InvalidCast {
        from: Types,
        to: Types,
    },
    UnmatchArgsAmount {
        expected: usize,
        given: usize,
    },
    MissingReturn(String),
    NotList(String),
    NotMatrix(String),
    UsePrimitive,
    InconsistentSize {
        expected: Option<usize>,
        given: Option<usize>,
    },
    OnlyOneDataframe,
}

impl fmt::Debug for RaoulErrorKind {
    fn fmt(&self, f: &mut fmt::Formatter) ->
fmt::Result {
        match self {
            Self::UsePrimitive => write!(f, "We can't
handle using the complete array"),
            Self::UndeclaredVar(name) => write!(f,
"Variable \"{name}\" was not declared"),
            Self::UndeclaredFunction(name) => {
                write!(
                    f,
                    "Function \"{name}\" was not
declared or does not return a non-void value",
                )
            }
            Self::UndeclaredFunction2(name) => {
                write!(f, "Function \"{name}\" was
not declared")
            }
            Self::RedeclaredFunction(name) => {
                write!(f, "Function \"{name}\" was
already declared before")
            }
            Self::RedefinedType { name, from, to } =>
{
                write!(
                    f,
                    "\"{name}\" was originally
defined as {from:?} and you're attempting to
redefined it as a {to:?}",
                )
            }
            Self::InvalidCast { from, to } =>
write!(f, "Cannot cast from {from:?} to {to:?}"),
            Self::MemoryExceded => write!(f, "Memory
was exceded"),
            Self::UnmatchArgsAmount { expected, given
} => {
                write!(
                    f,

```

```

        "Wrong args amount: Expected
{expected}, but were given {given}"
    )
}
Self::MissingReturn(name) => {
    write!(f, "In function {name} not all
branches return a value")
}
Self::NotList(name) => write!(f,
    "{name}` is not a list"),
Self::NotMatrix(name) => write!(f,
    "{name}` is not a matrix"),
Self::InconsistentSize { expected, given
} => {
    write!(
        f,
        "Expecting matrix with second
dimension being {} but received {}",
        expected.unwrap_or(0),
        given.unwrap_or(0)
    )
}
Self::OnlyOneDataframe => write!(f, "Only
one dataframe is allowed per program"),
}
}
}

```

## src/error/mod.rs

```

#[allow(clippy::module_name_repetitions)]
pub mod error_kind;

use core::fmt;
use std::fmt::Debug;

use pest::error::{Error, ErrorVariant};
use pest::Span;

use crate::ast::AstNode;
use crate::parser::Rule;

use self::error_kind::RaoulErrorKind;

#[derive(PartialEq, Eq, Clone)]
#[allow(clippy::module_name_repetitions)]
pub struct RaoulError<'a> {
    kind: RaoulErrorKind,
    span: Span<'a>,
}

impl fmt::Debug for RaoulError<'_> {
    fn fmt(&self, f: &mut fmt::Formatter) ->
    fmt::Result {
        let message = format!("{:?}", self.kind);
        let error: Error<Rule> =
            Error::new_from_span(ErrorVariant::CustomError {
                message }, self.span.clone());
        write!(f, "{}", error)
    }
}

impl RaoulError<'_> {
    pub fn new<'a>(node: &AstNode<'a>, kind:
    RaoulErrorKind) -> RaoulError<'a> {
        RaoulError {
            kind,
            span: node.span.clone(),
        }
    }

    pub fn new_vec<'a>(node: &AstNode<'a>, kind:
    RaoulErrorKind) -> Vec<RaoulError<'a>> {
        vec![RaoulError::new(node, kind)]
    }
}

```

```

fn from_results_iter<'a, T, I: IntoIterator<Item
= Results<'a, T>>>(
    iter: I,
) -> Vec<RaoulError<'a>> {
    iter.into_iter()
        .filter_map(Results::err)
        .flatten()
        .collect()
}

pub fn create_results<'a, T, I: IntoIterator<Item
= Results<'a, T>>>(
    iter: I,
) -> Results<'a, ()> {
    let errors =
    RaoulError::from_results_iter(iter);
    if errors.is_empty() {
        Ok(())
    } else {
        Err(errors)
    }
}

pub fn create_partition<'a, T: Debug, I:
IntoIterator<Item = Results<'a, T>>>(
    iter: I,
) -> Results<'a, Vec<T>> {
    let (oks, errors): (Vec<_>, Vec<_>) =
    iter.into_iter().partition(Results::is_ok);
    if errors.is_empty() {
        Ok(oks.into_iter().map(Results::unwrap).collect())
    } else {
        Err(errors.into_iter().flat_map(Results::unwrap_err).
        collect())
    }
}

pub type Result<'a, T> = std::result::Result<T,
RaoulError<'a>>;
pub type Results<'a, T> = std::result::Result<T,
Vec<RaoulError<'a>>>;

```

## src/parser/mod.rs

```

use pest_consume::match_nodes;
use pest_consume::Parser;

use crate::ast::ast_kind::AstNodeKind;
use crate::ast::AstNode;
use crate::enums::{Operator, Types};

#[derive(Parser)]
#[grammar = "parser/grammar.pest"] // relative to src
struct LanguageParser;

use pest_consume::Error;
type Result<T> = std::result::Result<T, Error<Rule>>;
type Node<'i> = pest_consume::Node<'i, Rule, bool>;

// This is the other half of the parser, using
pest_consume.
#[pest_consume::parser]
impl LanguageParser {
    // Extra
    fn EOI(input: Node) -> Result<()> {
        Ok(())
    }

    fn global(input: Node) -> Result<()> {
        Ok(())
    }

    // Types
    fn void(input: Node) -> Result<Types> {
        Ok(Types::Void)
    }
}

```



```

}

fn int(input: Node) -> Result<Types> {
    Ok(Types::Int)
}

fn float(input: Node) -> Result<Types> {
    Ok(Types::Float)
}

fn string(input: Node) -> Result<Types> {
    Ok(Types::String)
}

fn bool(input: Node) -> Result<Types> {
    Ok(Types::Bool)
}

fn atomic_types(input: Node) -> Result<Types> {
    Ok(match_nodes!(input.into_children();
        [int(value)] => value,
        [float(value)] => value,
        [string(value)] => value,
        [bool(value)] => value,
    ))
}

fn types(input: Node) -> Result<Types> {
    Ok(match_nodes!(input.into_children();
        [void(value)] => value,
        [atomic_types(value)] => value,
    ))
}

// Operations
fn not(input: Node) -> Result<Operator> {
    Ok(Operator::Not)
}

fn gte(input: Node) -> Result<Operator> {
    Ok(Operator::Gte)
}

fn lte(input: Node) -> Result<Operator> {
    Ok(Operator::Lte)
}

fn gt(input: Node) -> Result<Operator> {
    Ok(Operator::Gt)
}

fn lt(input: Node) -> Result<Operator> {
    Ok(Operator::Lt)
}

fn rel_op(input: Node) -> Result<Operator> {
    Ok(match_nodes!(input.into_children();
        [gte(value)] => value,
        [lte(value)] => value,
        [gt(value)] => value,
        [lt(value)] => value,
    ))
}

fn eq(input: Node) -> Result<Operator> {
    Ok(Operator::Eq)
}

fn ne(input: Node) -> Result<Operator> {
    Ok(Operator::Ne)
}

fn comp_op(input: Node) -> Result<Operator> {
    Ok(match_nodes!(input.into_children();
        [eq(value)] => value,
        [ne(value)] => value,
    ))
}

fn sum(input: Node) -> Result<Operator> {
    Ok(Operator::Sum)
}

fn minus(input: Node) -> Result<Operator> {
    Ok(Operator::Minus)
}

fn art_op(input: Node) -> Result<Operator> {
    Ok(match_nodes!(input.into_children();
        [sum(value)] => value,
        [minus(value)] => value,
    ))
}

fn times(input: Node) -> Result<Operator> {
    Ok(Operator::Times)
}

fn div(input: Node) -> Result<Operator> {
    Ok(Operator::Div)
}

fn fact_op(input: Node) -> Result<Operator> {
    Ok(match_nodes!(input.into_children();
        [times(value)] => value,
        [div(value)] => value,
    ))
}

// Values
fn int_cte(input: Node) -> Result<AstNode> {
    let value = input
        .as_str()
        .parse:::<i64>()
        .map_err(|e| input.error(e))
        .unwrap();
    let kind = AstNodeKind::Integer(value);
    Ok(AstNode {
        kind,
        span: input.as_span(),
    })
}

fn float_cte(input: Node) -> Result<AstNode> {
    let value = input
        .as_str()
        .parse:::<f64>()
        .map_err(|e| input.error(e))
        .unwrap();
    Ok(AstNode {
        kind: AstNodeKind::Float(value),
        span: input.as_span(),
    })
}

fn string_value(input: Node) -> Result<AstNode> {
    Ok(AstNode {
        kind:
            AstNodeKind::String(input.as_str().to_owned()),
        span: input.as_span(),
    })
}

fn bool_cte(input: Node) -> Result<AstNode> {
    let value = input
        .as_str()
        .parse:::<bool>()
        .map_err(|e| input.error(e))
        .unwrap();
    Ok(AstNode {
        kind: AstNodeKind::Bool(value),
        span: input.as_span(),
    })
}

fn func_call(input: Node) -> Result<AstNode> {

```

```

        let span = input.as_span();
        Ok(match_nodes!(input.into_children());
        [id(id)] => {
            let kind = AstNodeKind::FuncCall {
name: String::from(id), exprs: Vec::new() };
            AstNode { kind, span }
        },
        [id(id), exprs(exprs)] => {
            let kind = AstNodeKind::FuncCall {
name: String::from(id), exprs };
            AstNode { kind, span }
        },
    ))
}

fn non_cte(input: Node) -> Result<AstNode> {
    Ok(match_nodes!(input.into_children());
    [expr(expr)] => expr,
    [id(id)] => id,
    [func_call(call)] => call,
    [arr_val(id)] => id,
    [dataframe_value_ops(id)] => id,
    ))
}

fn possible_str(input: Node) -> Result<AstNode> {
    Ok(match_nodes!(input.into_children());
    [non_cte(expr)] => expr,
    [string_value(string)] => string,
    ))
}

// ID
fn id(input: Node) -> Result<AstNode> {
    Ok(AstNode {
        kind:
AstNodeKind::Id(input.as_str().to_owned()),
        span: input.as_span(),
    })
}

// Expressions
fn expr(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children());
    [and_term(value)] => value,
    [and_term(lhs), and_term(rhs)] => {
        let kind =
AstNodeKind::BinaryOperation {
            operator: Operator::Or,
            lhs: Box::new(lhs),
            rhs: Box::new(rhs),
        };
        AstNode { kind, span }
    },
    ))
}

fn and_term(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children());
    [comp_term(value)] => value,
    [comp_term(lhs), comp_term(rhs)] => {
        let kind =
AstNodeKind::BinaryOperation {
            operator: Operator::And,
            lhs: Box::new(lhs),
            rhs: Box::new(rhs),
        };
        AstNode { kind, span }
    },
    ))
}

fn comp_term(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children());
    [rel_term(value)] => value,

```

```

        [rel_term(lhs), comp_op(operator),
rel_term(rhs)] => {
            let kind =
AstNodeKind::BinaryOperation {
                operator,
                lhs: Box::new(lhs),
                rhs: Box::new(rhs),
            };
            AstNode { kind, span }
        }
    ))
}

fn rel_term(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children());
    [art_term(value)] => value,
    [art_term(lhs), rel_op(operator),
art_term(rhs)] => {
        let kind =
AstNodeKind::BinaryOperation {
            operator,
            lhs: Box::new(lhs),
            rhs: Box::new(rhs),
        };
        AstNode { kind, span }
    })
}

fn art_term(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children());
    [fact_term(value)] => value,
    [fact_term(lhs), art_op(operator),
fact_term(rhs)] => {
        let kind =
AstNodeKind::BinaryOperation {
            operator,
            lhs: Box::new(lhs),
            rhs: Box::new(rhs),
        };
        AstNode { kind, span }
    })
}

fn fact_term(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children());
    [operand(value)] => value,
    [operand(lhs), fact_op(operator),
operand(rhs)] => {
        let kind =
AstNodeKind::BinaryOperation {
            operator,
            lhs: Box::new(lhs),
            rhs: Box::new(rhs),
        };
        AstNode { kind, span }
    })
}

fn operand(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children());
    [operand_value(value)] => value,
    [not(operator), operand_value(operand)]
=> {
        let kind =
AstNodeKind::UnaryOperation { operator, operand:
Box::new(operand) };
        AstNode { kind, span }
    })
}

```

```

fn operand_value(input: Node) -> Result<AstNode>
{
    Ok(match_nodes!(input.into_children();
        [expr(expr)] => expr,
        [non_cte(expr)] => expr,
        [int_cte(number)] => number,
        [float_cte(number)] => number,
        [string_value(string)] => string,
        [bool_cte(value)] => value,
    ))
}

fn exprs(input: Node) -> Result<Vec<AstNode>> {
    Ok(match_nodes!(input.into_children();
        [expr(exprs)..] => exprs.collect(),
    ))
}

// Arrays
fn declare_arr_type(input: Node) -> Result<Types>
{
    Ok(match_nodes!(input.into_children();
        [atomic_types(data_type)] => data_type,
    ))
}

fn declare_arr(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [declare_arr_type(data_type),
int_cte(dim1)] => {
        let kind =
AstNodeKind::ArrayDeclaration { data_type, dim1:
dim1.into(), dim2: None };
        AstNode { kind, span }
    },
        [declare_arr_type(data_type),
int_cte(dim1), int_cte(dim2)] => {
        let kind =
AstNodeKind::ArrayDeclaration { data_type, dim1:
dim1.into(), dim2: Some(dim2.into()) };
        AstNode { kind, span }
    },
    ))
}

fn list_cte(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [exprs(exprs)] => {
        AstNode { kind:
AstNodeKind::Array(exprs), span }
    },
    ))
}

fn mat_cte(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [list_cte(exprs)..] => {
        AstNode { kind:
AstNodeKind::Array(exprs.collect(), span }
    },
    ))
}

fn arr_cte(input: Node) -> Result<AstNode> {
    Ok(match_nodes!(input.into_children();
        [list_cte(node)] => node,
        [mat_cte(node)] => node,
    ))
}

fn arr_val(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [id(name), expr(idx_1)] => {
        let name = String::from(name);

```

```

        let idx_1 = Box::new(idx_1);
        let kind = AstNodeKind::ArrayVal {
name, idx_1, idx_2: None };
        AstNode { kind, span }
    },
        [id(name), expr(idx_1), expr(idx_2)] => {
        let name = String::from(name);
        let idx_1 = Box::new(idx_1);
        let kind = AstNodeKind::ArrayVal {
name, idx_1, idx_2: Some(Box::new(idx_2)) };
        AstNode { kind, span }
    },
    ))
}

// Dataframe
fn read_csv(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [possible_str(file)] => {
        let node = Box::new(file);
AstNode::new(AstNodeKind::ReadCSV(node), &span)
    },
    ))
}

fn get_rows(input: Node) -> Result<Operator> {
    Ok(Operator::Rows)
}

fn get_columns(input: Node) -> Result<Operator> {
    Ok(Operator::Columns)
}

fn pure_dataframe_key(input: Node) ->
Result<Operator> {
    Ok(match_nodes!(input.into_children();
        [get_rows(op)] => op,
        [get_columns(op)] => op,
    ))
}

fn pure_dataframe_op(input: Node) ->
Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [pure_dataframe_key(operator), id(id)] =>
{
        let name = String::from(id);
        let kind =
AstNodeKind::PureDataframeOp {
            name, operator
        };
        AstNode { kind, span }
    },
    ))
}

fn average(input: Node) -> Result<Operator> {
    Ok(Operator::Average)
}

fn std(input: Node) -> Result<Operator> {
    Ok(Operator::Std)
}

fn median(input: Node) -> Result<Operator> {
    Ok(Operator::Median)
}

fn variance(input: Node) -> Result<Operator> {
    Ok(Operator::Variance)
}

fn min(input: Node) -> Result<Operator> {
    Ok(Operator::Min)
}

```

```

fn max(input: Node) -> Result<Operator> {
    Ok(Operator::Max)
}

fn range(input: Node) -> Result<Operator> {
    Ok(Operator::Range)
}

fn unary_dataframe_key(input: Node) ->
Result<Operator> {
    Ok(match_nodes!(input.into_children();
        [average(op)] => op,
        [std(op)] => op,
        [median(op)] => op,
        [variance(op)] => op,
        [min(op)] => op,
        [max(op)] => op,
        [range(op)] => op,
    ))
}

fn unary_dataframe_op(input: Node) ->
Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [unary_dataframe_key(operator), id(id),
possible_str(col)] => {
            let name = String::from(id);
            let column = Box::new(col);
            let kind =
AstNodeKind::UnaryDataframeOp {
                name, column, operator
            };
            AstNode { kind, span }
        },
    ))
}

fn correlation(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [id(id), possible_str(col_1),
possible_str(col_2)] => {
            let name = String::from(id);
            let column_1 = Box::new(col_1);
            let column_2 = Box::new(col_2);
            let kind = AstNodeKind::Correlation {
                name, column_1, column_2
            };
            AstNode { kind, span }
        },
    ))
}

fn dataframe_value_ops(input: Node) ->
Result<AstNode> {
    Ok(match_nodes!(input.into_children();
        [pure_dataframe_op(node)] => node,
        [unary_dataframe_op(node)] => node,
        [correlation(node)] => node,
    ))
}

fn plot(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [id(id), possible_str(col_1),
possible_str(col_2)] => {
            let name = String::from(id);
            let column_1 = Box::new(col_1);
            let column_2 = Box::new(col_2);
            let kind = AstNodeKind::Plot {
                name, column_1, column_2
            };
            AstNode { kind, span }
        },
    ))
}

```

```

}

fn histogram(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [id(id), possible_str(col), expr(bins)]
=> {
            let name = String::from(id);
            let column = Box::new(col);
            let bins = Box::new(bins);
            let kind = AstNodeKind::Histogram {
name, column, bins };
            AstNode { kind, span }
        },
    ))
}

// Condition
fn else_block(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [block_or_statement(statements)] => {
            let kind =
AstNodeKind::ElseBlock(statements);
            AstNode {kind, span}
        },
        [decision(decision)] => decision,
    ))
}

fn decision(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [expr(expr),
block_or_statement(statements)] => {
            let kind = AstNodeKind::Decision {
                expr: Box::new(expr),
                statements,
                else_block: None
            };
            AstNode {kind, span}
        },
        [expr(expr),
block_or_statement(statements),
else_block(else_block)] => {
            let kind = AstNodeKind::Decision {
                expr: Box::new(expr),
                statements,
                else_block:
Some(Box::new(else_block))
            };
            AstNode {kind, span}
        },
    ))
}

// Loops
fn while_loop(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [expr(expr),
block_or_statement(statements)] => {
            let kind = AstNodeKind::While {
                expr: Box::new(expr),
                statements,
            };
            AstNode {kind, span}
        },
    ))
}

fn for_loop(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [assignment(assignment), expr(stop_expr),
block_or_statement(statements)] => {

```

```

        let id_node =
AstNode::new(AstNodeKind::Id(String::from(&assignment
)), &assignment.span);
        let expr_kind =
AstNodeKind::BinaryOperation {
            operator: Operator::Lte,
            lhs: Box::new(id_node),
            rhs: Box::new(stop_expr.clone()),
        };
        let expr =
Box::new(AstNode::new(expr_kind, &stop_expr.span));
        let kind = AstNodeKind::For {
            assignment: Box::new(assignment), expr, statements };
        AstNode { kind, span }
    },
    ))
}

// Inline statements
fn assignee(input: Node) -> Result<Box<AstNode>>
{
    Ok(match_nodes!(input.into_children();
        [id(id)] => Box::new(id),
        [arr_val(id)] => Box::new(id),
    ))
}

fn read(input: Node) -> Result<AstNode> {
    Ok(AstNode::new(AstNodeKind::Read,
&input.as_span()))
}

fn assignment_exp(input: Node) -> Result<AstNode>
{
    Ok(match_nodes!(input.into_children();
        [expr(value)] => value,
        [read(value)] => value,
        [declare_arr(value)] => value,
        [arr_cte(arr)] => arr,
        [read_csv(v)] => v,
    ))
}

fn assignment(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [global(_), assignee(id),
assignment_exp(value)] => {
            let kind = AstNodeKind::Assignment {
global: true, assignee: id, value: Box::new(value) };
            AstNode { kind, span }
        },
        [assignee(id), assignment_exp(value)] =>
{
            let kind = AstNodeKind::Assignment {
global: false, assignee: id, value: Box::new(value)
};
            AstNode { kind, span }
        },
    ))
}

fn global_assignment(input: Node) ->
Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [assignee(id), assignment_exp(value)] =>
{
            let kind = AstNodeKind::Assignment {
global: true, assignee: id, value: Box::new(value) };
            AstNode { kind, span }
        },
    ))
}

fn write(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();

```

```

        [exprs(exprs)] => {
            AstNode { kind:
AstNodeKind::Write(exprs), span }
        },
    ))
}

fn return_statement(input: Node) ->
Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [expr(expr)] => {
            AstNode { kind:
AstNodeKind::Return(Box::new(expr)), span }
        },
    ))
}

fn inline_statement(input: Node) ->
Result<AstNode> {
    Ok(match_nodes!(input.into_children();
        [assignment(node)] => node,
        [write(node)] => node,
        [func_call(node)] => node,
        [return_statement(node)] => node,
        [plot(node)] => node,
        [histogram(node)] => node,
    ))
}

fn statement(input: Node) -> Result<AstNode> {
    Ok(match_nodes!(input.into_children();
        [inline_statement(node)] => node,
        [decision(node)] => node,
        [while_loop(node)] => node,
        [for_loop(node)] => node,
    ))
}

fn block(input: Node) -> Result<Vec<AstNode>> {
    Ok(match_nodes!(input.into_children();
        [statement(statements)..] =>
statements.collect(),
    ))
}

fn block_or_statement(input: Node) ->
Result<Vec<AstNode>> {
    Ok(match_nodes!(input.into_children();
        [inline_statement(statements)] =>
vec![statements],
        [block(block)] => block,
    ))
}

// Function
fn func_arg(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
        [id(id), atomic_types(arg_type)] => {
            let kind = AstNodeKind::Argument {
arg_type, name: String::from(id) };
            AstNode { kind, span }
        },
    ))
}

fn func_args(input: Node) -> Result<Vec<AstNode>>
{
    Ok(match_nodes!(input.into_children();
        [func_arg(args)..] => args.collect(),
    ))
}

fn function(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();

```

```

        [id(id), func_args(arguments),
types(return_type), block(body)] => {
    let kind = AstNodeKind::Function
{arguments, name: String::from(id), body,
return_type};
        AstNode { kind, span }
    },
    [id(id), types(return_type), block(body)]
=> {
    let kind = AstNodeKind::Function
{arguments: Vec::new(), name: String::from(id), body,
return_type};
        AstNode { kind, span }
    },
    ))
}

fn global_assignments(input: Node) ->
Result<Vec<AstNode>> {
    Ok(match_nodes!(input.into_children();
    [global_assignment(args)..] =>
args.collect(),
    ))
}

fn program(input: Node) -> Result<AstNode> {
    let span = input.as_span();
    Ok(match_nodes!(input.into_children();
    [global_assignments(nodes),
function(functions).., _, block(body), _] => {
        let kind = AstNodeKind::Main {
            assignments: nodes,
            body,
            functions: functions.collect(),
        };
        AstNode { kind, span }
    },
    ))
}

pub fn parse(source: &str, debug: bool) ->
Result<AstNode> {
    let inputs =
LanguageParser::parse_with_userdata(Rule::program,
source, debug)?;
    // There should be a single root node in the
    parsed tree
    let input = inputs.single()?;
    LanguageParser::program(input)
}

#[cfg(test)]
mod tests;

```

## src/parser/tests.rs

```

use super::*;
use std::fs::read_dir;

#[test]
fn valid_files() {
    let paths =
read_dir("src/examples/valid").unwrap();
    for path in paths {
        let file_path = path.expect("File must
exist").path();
        let file = file_path.to_str().unwrap();
        if file == "examples/valid/complete.ra" {
            continue;
        }
        let program =
std::fs::read_to_string(file).expect(file);
        println!("Testing {:?}", file);
        let res = parse(&program, true);
        assert!(res.is_ok());
    }
}

```

```

#[test]
fn invalid_file() {
    let filename =
"src/examples/invalid/syntax/syntax-error.ra";
    let program =
std::fs::read_to_string(filename).expect(filename);
    let res = parse(&program, true);
    assert!(res.is_err());
}

```

## src/quadruple/mod.rs

```

#[allow(clippy::module_inception)]
pub mod quadruple;
#[allow(clippy::module_name_repetitions)]
pub mod quadruple_manager;

```

## src/quadruple/quadruple.rs

```

use std::fmt;

use crate::enums::Operator;

#[derive(Clone, Copy, PartialEq, Hash, Eq)]
pub struct Quadruple {
    pub operator: Operator,
    pub op_1: Option<usize>,
    pub op_2: Option<usize>,
    pub res: Option<usize>,
}

impl Quadruple {
    fn format_address(option: Option<usize>) ->
String {
        match option {
            None => "-".to_owned(),
            Some(address) => address.to_string(),
        }
    }

    pub fn new(
        operator: Operator,
        op_1: Option<usize>,
        op_2: Option<usize>,
        res: Option<usize>,
    ) -> Self {
        Quadruple {
            operator,
            op_1,
            op_2,
            res,
        }
    }

    pub fn new_empty(operator: Operator) -> Self {
        Self::new(operator, None, None, None)
    }

    pub fn new_arg(operator: Operator, op_1: usize)
-> Self {
        Self::new(operator, Some(op_1), None, None)
    }

    pub fn new_res(operator: Operator, res: usize) ->
Self {
        Self::new(operator, None, None, Some(res))
    }

    pub fn new_un(operator: Operator, op_1: usize,
res: usize) -> Self {
        Self::new(operator, Some(op_1), None,
Some(res))
    }

    pub fn new_args(operator: Operator, op_1: usize,
op_2: usize) -> Self {

```

```

        Self::new(operator, Some(op_1), Some(op_2),
None)
    }

    pub fn new_com(operator: Operator, op_1: usize,
op_2: usize, res: usize) -> Self {
        Self::new(operator, Some(op_1), Some(op_2),
Some(res))
    }
}

impl fmt::Debug for Quadruple {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) ->
fmt::Result {
        write!(
            f,
            "{} {:5} {:5} {}",
            self.operator,
            Quadruple::format_address(self.op_1),
            Quadruple::format_address(self.op_2),
            Quadruple::format_address(self.res),
        )
    }
}

```

## src/quadruple/quadruple\_manager.rs

```

use std::fmt;

use crate::{
    address::{Address, ConstantMemory,
GenericAddressManager, PointerMemory},
    ast::{ast_kind::AstNodeKind, AstNode, BoxedNode},
    dir_func::{
        function::{Function, VariablesTable},
        variable::Variable,
        variable_value::VariableValue,
        DirFunc,
    },
    enums::{Operator, Types},
    error::{error_kind::RaoulErrorKind, RaoulError,
Results},
    quadruple::quadruple::Quadruple,
};

#[derive(PartialEq, Debug)]
pub struct QuadrupleManager {
    function_name: String,
    jump_list: Vec<usize>,
    missing_return: bool,
    pub dir_func: DirFunc,
    pub memory: ConstantMemory,
    pub pointer_memory: PointerMemory,
    pub quad_list: Vec<Quadruple>,
}

pub type Operand = (usize, Types);

fn safe_address<'a, T>(option: Option<T>, node:
&AstNode<'a>) -> Results<'a, T> {
    match option {
        Some(value) => Ok(value),
        None => Err(vec![RaoulError::new(node,
RaoulErrorKind::MemoryExceeded)]),
    }
}

impl QuadrupleManager {
    pub fn new(dir_func: DirFunc) -> QuadrupleManager
    {
        QuadrupleManager {
            dir_func,
            function_name: "".to_owned(),
            jump_list: Vec::new(),
            memory: ConstantMemory::new(),
            missing_return: false,
            pointer_memory: PointerMemory::new(),
            quad_list: Vec::new(),
        }
    }
}

```

```

    }

    #[inline]
    pub fn clear_variables(&mut self) {
        self.dir_func.clear_variables();
    }

    #[inline]
    fn get_function(&self, name: &str) -> &Function {
        self.dir_func
            .functions
            .get(name)
            .expect(&self.function_name)
    }

    #[inline]
    fn function(&self) -> &Function {
        self.get_function(&self.function_name)
    }

    #[inline]
    fn function_variables(&self) -> &VariablesTable {
        &self.function().variables
    }

    #[inline]
    fn global_variables(&self) -> &VariablesTable {
        &self.dir_func.global_fn.variables
    }

    fn get_variable_address(&self, global: bool,
name: &str) -> usize {
        let variables = if global {
            self.global_variables()
        } else {
            self.function_variables()
        };
        variables.get(name).expect(name).address
    }

    #[inline]
    fn function_mut(&mut self) -> &mut Function {
        self.dir_func
            .functions
            .get_mut(&self.function_name)
            .unwrap()
    }

    #[inline]
    fn add_temp(&mut self, data_type: Types) ->
Option<usize> {
        self.function_mut()
            .temp_addresses
            .get_address(data_type, (None, None))
    }

    #[inline]
    fn safe_add_temp<'a>(&mut self, data_type: Types,
node: &AstNode<'a>) -> Results<'a, usize> {
        safe_address(self.add_temp(data_type), node)
    }

    fn safe_remove_temp_address(&mut self, operand:
Option<usize>) {
        if !operand.is_temp_address() {
            return;
        }
        self.function_mut()
            .temp_addresses
            .release_address(operand.unwrap());
    }

    fn add_quad(&mut self, quad: Quadruple) {
        self.quad_list.push(quad);
        self.safe_remove_temp_address(quad.op_1);
        self.safe_remove_temp_address(quad.op_2);
    }
}

```

```

    fn get_variable<'a>(&mut self, name: &str, node:
&AstNode<'a>) -> Results<'a, &Variable> {
        match self
            .function_variables()
            .get(name)
            .or_else(||
self.global_variables().get(name))
        {
            Some(var) => Ok(var),
            None => Err(RaoulError::new_vec(
                node,

RaoulErrorKind::UndeclaredVar(name.to_string()),
            )),
        }
    }

    fn get_variable_name_address<'a>(&mut self,
name: &str,
node: &AstNode<'a>,
) -> Results<'a, (usize, Types)> {
    let variable = self.get_variable(name,
node)?;
    Ok((variable.address, variable.data_type))
}

    fn parse_args_exprs<'a>(&mut self,
node: &AstNode<'a>,
exprs: &[AstNode<'a>],
args: &[Operand],
) -> Results<'a, Vec<Operand>> {
    if args.len() != exprs.len() {
        let kind =
RaoulErrorKind::UnmatchArgsAmount {
            expected: args.len(),
            given: exprs.len(),
        };
        return Err(vec![RaoulError::new(node,
kind)]);
    }
    let addresses =
RaoulError::create_partition(exprs.iter().zip(args).map(
| (node, (_, arg_type)) | ->
Results<(usize, Types)> {
    let (v, v_type) =
self.parse_expr(node)?;
    v_type.assert_cast(*arg_type, node)?;
    Ok((v, v_type))
},
));
    Ok(addresses)
}

    fn add_era_quad(&mut self, name: &str) {
        let function = self.get_function(name);
        let function_size = function.size();
        let first_quad = function.first_quad;
        self.add_quad(Quadruple::new_args(
            Operator::Era,
            function_size,
            first_quad,
        ));
    }

    fn add_go_sub_quad(&mut self, name: &str) {
        let first_quad =
self.get_function(name).first_quad;

self.add_quad(Quadruple::new_arg(Operator::GoSub,
first_quad));
    }

    fn parse_func_call<'a>(&mut self,

```

```

name: &str,
node: &AstNode<'a>,
exprs: &[AstNode<'a>],
) -> Results<'a, ()> {
    self.add_era_quad(name);
    let args =
&self.get_function(name).args.clone();
    let addresses = self.parse_args_exprs(node,
exprs, args)?;
    addresses
        .into_iter()
        .enumerate()
        .for_each(|(i, (address, _))| {
self.add_quad(Quadruple::new_un(Operator::Param,
address, i));
        });
    self.add_go_sub_quad(name);
    Ok(())
}

#[inline]
fn safe_add_cte<'a>(&mut self,
value: VariableValue,
node: &AstNode<'a>,
) -> Results<'a, (usize, Types)> {
    safe_address(self.memory.add(value), node)
}

    fn add_binary_op_quad<'a>(&mut self,
operator: Operator,
op_1: Operand,
op_2: Operand,
node: &AstNode<'a>,
) -> Results<'a, Operand> {
    let data_type =
op_1.1.assert_bin_op(operator, op_2.1, node)?;
    let res = self.safe_add_temp(data_type,
node)?;
    self.add_quad(Quadruple::new_com(operator,
op_1.0, op_2.0, res));
    Ok((res, data_type))
}

    fn get_array_val_operand<'a>(&mut self,
name: &str,
node: &AstNode<'a>,
idx_1_op: &Operand,
idx_2_op: Option<Operand>,
) -> Results<'a, Operand> {
    let v = (self.get_variable(name,
node)?).clone();
    let (dim_1, dim_2) = v.dimensions;
    if dim_1.is_none() {
        return Err(RaoulError::new_vec(
            node,

RaoulErrorKind::NotList(name.to_owned()),
        ));
    }
    match (dim_2.is_none(), idx_2_op.is_none()) {
        (true, false) => Err(RaoulError::new_vec(
            node,

RaoulErrorKind::NotMatrix(name.to_owned()),
        )),
        (false, true) =>
Err(RaoulError::new_vec(node,
RaoulErrorKind::UsePrimitive)),
        _ => Ok(()),
    };
    let v_address_op =
self.safe_add_cte(v.address.into(), node)?;
    let dim_1_op =
self.safe_add_cte(dim_1.unwrap().into(), node)?;

```



```

self.add_quad(Quadruple::new_args(Operator::Ver,
idx_1_op.0, dim_1_op.0));
    let address: usize = match idx_2_op {
        None => {
            let pointer =
self.pointer_memory.get_pointer();
            self.add_quad(Quadruple::new_com(
                Operator::Sum,
                v_address_op.0,
                idx_1_op.0,
                pointer,
            ));
            pointer
        }
        Some(idx_2_op) => {
            let dim_2_op =
self.safe_add_cte(dim_2.unwrap().into(), node)?;
            let mult_op =

self.add_binary_op_quad(Operator::Times, *idx_1_op,
dim_2_op, node)?;

self.add_quad(Quadruple::new_args(Operator::Ver,
idx_2_op.0, dim_2_op.0));
            let (sum_res, _) =

self.add_binary_op_quad(Operator::Sum, v_address_op,
mult_op, node)?;
            let pointer =
self.pointer_memory.get_pointer();
            self.add_quad(Quadruple::new_com(
                Operator::Sum,
                sum_res,
                idx_2_op.0,
                pointer,
            ));
            pointer
        }
    };
    Ok((address, v.data_type))
}

fn arr_val_op_node<'a>(&mut self,
name: &str,
node: &AstNode<'a>,
idx_1: &AstNode<'a>,
idx_2: Option<BoxedNode<'a>>,
) -> Results<'a, Operand> {
    let idx_1_op = &self.assert_expr_type(idx_1,
Types::Int)?;
    let idx_2_op = match idx_2 {
        Some(idx_2) =>
Some(self.assert_expr_type(&*idx_2, Types::Int)?),
        None => None,
    };
    self.get_array_val_operand(name, node,
idx_1_op, idx_2_op)
}

fn assert_dataframe<'a>(&mut self, name: &str,
node: &AstNode<'a>) -> Results<'a, ()> {
    let data_type = self.get_variable(name,
node)?.data_type;
    data_type.assert_cast(Types::Dataframe, node)
}

fn dataframe_op<'a>(&mut self,
name: &str,
node: &AstNode<'a>,
operator: Operator,
op_1: usize,
op_2: Option<usize>,
) -> Results<'a, Operand> {
    self.assert_dataframe(name, node)?;
    let data_type = Types::Float;

```

```

        let res = self.safe_add_temp(data_type,
node)?;
        self.add_quad(Quadruple::new(operator,
Some(op_1), op_2, Some(res)));
        Ok((res, data_type))
    }

    fn parse_expr<'a>(&mut self, node: &AstNode<'a>)
-> Results<'a, Operand> {
        match &node.kind {
            AstNodeKind::Bool(_)
            | AstNodeKind::Float(_)
            | AstNodeKind::Integer(_)
            | AstNodeKind::String(_) =>
self.safe_add_cte(VariableValue::from(&node.kind),
node),
            AstNodeKind::UnaryOperation { operator,
operand } => {
                let (op, op_type) =
self.parse_expr(&*operand)?;
                let res_type = match operator {
                    Operator::Not => match op_type {
                        Types::Bool | Types::Int =>
Types::Bool,
                        op_type => {
                            let kind =
RaoulErrorKind::InvalidCast {
                                from: op_type,
                                to: Types::Bool,
                            };
                            return
Err(vec![RaoulError::new(node, kind)]);
                        },
                    }
                    _ => unreachable!(),
                };
                let res =
self.safe_add_temp(res_type, node)?;

self.add_quad(Quadruple::new_un(*operator, op, res));
                Ok((res, res_type))
            }
            AstNodeKind::Id(name) => {
                let variable =
self.get_variable(name, node)?;
                match variable.dimensions.0 {
                    None => Ok((variable.address,
variable.data_type)),
                    _ =>
Err(RaoulError::new_vec(node,
RaoulErrorKind::UsePrimitive)),
                }
            }
            AstNodeKind::Read => {
                let data_type = Types::String;
                let res =
self.safe_add_temp(data_type, node)?;

self.add_quad(Quadruple::new_res(Operator::Read,
res));
                Ok((res, data_type))
            }
            AstNodeKind::BinaryOperation { operator,
lhs, rhs } => {
                let op_1 = self.parse_expr(&*lhs)?;
                let op_2 = self.parse_expr(&*rhs)?;
                self.add_binary_op_quad(*operator,
op_1, op_2, node)
            }
            AstNodeKind::FuncCall { name, ref exprs }
=> {
                self.parse_func_call(name, node,
exprs)?;
                let (fn_address, return_type) =
self.get_variable_name_address(name, node)?;
                let temp_address =
self.safe_add_temp(return_type, node)?;
                self.add_quad(Quadruple::new_un(

```

```

        Operator::Assignment,
        fn_address,
        temp_address,
    ));
    Ok((temp_address, return_type))
}
AstNodeKind::ArrayVal {
    ref name,
    idx_1,
    idx_2,
} => self.arr_val_op_node(name, node,
    &idx_1, idx_2.clone()),
AstNodeKind::PureDataframeOp { operator,
ref name } => {
    self.assert_dataframe(name, node)?;
    let data_type = Types::Int;
    let res =
self.safe_add_temp(data_type, node)?;
self.add_quad(Quadruple::new_res(*operator, res));
    Ok((res, data_type))
}
AstNodeKind::UnaryDataframeOp {
    operator,
    ref name,
    column,
} => {
    let (column_address, _) =
self.assert_expr_type(&*column, Types::String)?;
    self.dataframe_op(name, node,
    *operator, column_address, None)
}
AstNodeKind::Correlation {
    ref name,
    column_1,
    column_2,
} => {
    let (col_1, _) =
self.assert_expr_type(&*column_1, Types::String)?;
    let (col_2, _) =
self.assert_expr_type(&*column_2, Types::String)?;
    let operator = Operator::Corr;
    self.dataframe_op(name, node,
    operator, col_1, Some(col_2))
}
    kind => unreachable!("{kind:?}"),
}
}

fn assert_expr_type<'a>(&mut self,
    expr: &AstNode<'a>,
    to: Types,
) -> Results<'a, (usize, Types)> {
    let (res_address, res_type) =
self.parse_expr(expr)?;
    res_type.assert_cast(to, expr)?;
    Ok((res_address, res_type))
}

#[inline]
fn parse_body<'a>(&mut self, body:
    &[AstNode<'a>]) -> Results<'a, ()> {
RaoulError::create_results(body.iter().map(|node|
self.parse_statement(node)))
}

fn parse_return_body<'a>(&mut self, body:
    &[AstNode<'a>]) -> Results<'a, bool> {
    let prev = self.missing_return;
    self.parse_body(body)?;
    let current = self.missing_return;
    if self.missing_return != prev {
        self.missing_return = prev;
    }
    Ok(current)
}

```

```

    fn add_goto(&mut self, goto_type: Operator,
condition: Option<usize>) {
        debug_assert!(goto_type.is_goto());
        self.jump_list.push(self.quad_list.len());
        self.add_quad(Quadruple::new(goto_type,
condition, None, None));
    }

    fn fill_goto_index(&mut self, index: usize) {
        let res = self.quad_list.len();
        let mut quad =
self.quad_list.get_mut(index).unwrap();
        debug_assert!(quad.operator.is_goto());
        quad.res = Some(res);
    }

    fn fill_goto(&mut self) {
        let index = self.jump_list.pop().unwrap();
        self.fill_goto_index(index);
    }

    fn add_assign_quad<'a>(&mut self, res: usize,
value: &AstNode<'a>) -> Results<'a, ()> {
        let (op_1, _) = self.parse_expr(value)?;

self.add_quad(Quadruple::new_un(Operator::Assignment,
op_1, res));
        Ok(())
    }

    fn parse_array<'a>(&mut self,
        assignee: &AstNode<'a>,
        exprs: &[AstNode<'a>],
        node: &AstNode<'a>,
    ) -> Results<'a, ()> {
        let name = String::from(assignee);
        let variable = self.get_variable(&name,
assignee)?.clone();
        let dim_2 = variable.dimensions.1;
        if dim_2.is_none() {
RaoulError::create_results(exprs.iter().enumerate().m
ap(|(i, expr)| -> Results<()> {
            let idx_1_op =
self.safe_add_cte(i.into(), expr)?;
            let (variable_address, _) =
self.get_array_val_operand(&name,
node, &idx_1_op, None)?;

self.add_assign_quad(variable_address, expr)
            )))
        } else {
RaoulError::create_results(exprs.iter().enumerate().m
ap(|(i, exprs)| -> Results<()> {
            let idx_1_op =
self.safe_add_cte(i.into(), exprs)?;

RaoulError::create_results(exprs.expand_array().iter(
).enumerate().map(
                |(j, expr)| -> Results<()> {
                    let idx_2_op =
self.safe_add_cte(j.into(), expr)?;
                    let (variable_address, _) =
self.get_array_val_operand(&name, node, &idx_1_op,
Some(idx_2_op))?;

self.add_assign_quad(variable_address, expr)
                    },
                )))
            )))
        }
    }

    fn parse_assignment<'a>(&mut self,

```

```

        &mut self,
        assignee: &AstNode<'a>,
        global: bool,
        value: &AstNode<'a>,
        node: &AstNode<'a>,
    ) -> Results<'a, ()> {
        match &value.kind {
            AstNodeKind::ArrayDeclaration { .. } =>
                Ok(()),
            AstNodeKind::Array(exprs) =>
                self.parse_array(assignee, exprs, node),
            AstNodeKind::ReadCSV(file_node) => {
                let (file_address, _) =
                    self.assert_expr_type(&file_node, Types::String)?;
                self.add_quad(Quadruple::new_arg(Operator::ReadCSV,
                    file_address));
                Ok(())
            }
        }
        - => {
            let variable_address = if let
                AstNodeKind::ArrayVal {
                    ref name,
                    idx_1,
                    idx_2,
                } = &assignee.kind
            {
                let op =
                    self.arr_val_op_node(name, node, &idx_1,
                    idx_2.clone())?;
                op.0
            } else {
                let name: String =
                    assignee.into();
                self.get_variable_address(global,
                    &name)
            };
            self.add_assign_quad(variable_address, value)
        }
    }

    fn parse_for<'a>(&mut self,
        assignment: &AstNode<'a>,
        expr: &AstNode<'a>,
        statements: &[AstNode<'a>],
        node: &AstNode<'a>,
    ) -> Results<'a, ()> {
        let name = String::from(assignment);
        self.parse_statement(assignment)?;
        self.jump_list.push(self.quad_list.len());
        let (res_address, _) =
            self.assert_expr_type(expr, Types::Bool)?;
        self.add_goto(Operator::GotoF,
            Some(res_address));
        self.parse_return_body(statements)?;
        let (var_address, var_type) =
            self.get_variable_name_address(&name, node)?;
        var_type.assert_cast(Types::Int, node)?;

        self.add_quad(Quadruple::new_res(Operator::Inc,
            var_address));
        let index = self.jump_list.pop().unwrap();
        let goto_res = self.jump_list.pop().unwrap();

        self.add_quad(Quadruple::new_res(Operator::Goto,
            goto_res));
        self.fill_goto_index(index);
        Ok(())
    }

    fn parse_statement<'a>(&mut self, node:
        &AstNode<'a>) -> Results<'a, ()> {
        match &node.kind {
            AstNodeKind::Assignment {
                assignee,

```

```

                global,
                value,
            } => self.parse_assignment(&*assignee,
                *global, &*value, node),
            AstNodeKind::Write(exprs) => {
                RaoulError::create_results(exprs.iter().map(|expr| ->
                    Results<()> {
                        let (address, _) =
                            self.parse_expr(expr)?;
                        self.add_quad(Quadruple::new_arg(Operator::Print,
                            address));
                        Ok(())
                    })?;
                self.add_quad(Quadruple::new_empty(Operator::PrintNl)
                );
                Ok(())
            }
            AstNodeKind::Decision {
                expr,
                statements,
                else_block,
            } => {
                let (res_address, _) =
                    self.assert_expr_type(&*expr, Types::Bool)?;
                self.add_goto(Operator::GotoF,
                    Some(res_address));
                let if_misses_return =
                    self.parse_return_body(statements)?;
                if let Some(node) = else_block {
                    let index =
                        self.jump_list.pop().unwrap();
                    self.add_goto(Operator::Goto,
                        None);
                    self.fill_goto_index(index);
                    self.parse_statement(&*node)?;
                    self.fill_goto();
                    if if_misses_return &&
                        !self.missing_return {
                        self.missing_return = true;
                    }
                } else {
                    self.fill_goto();
                }
                Ok(())
            }
            AstNodeKind::ElseBlock(statements) =>
                self.parse_body(statements),
            AstNodeKind::While { expr, statements }
        } => {
            self.jump_list.push(self.quad_list.len());
            let (res_address, _) =
                self.assert_expr_type(&*expr, Types::Bool)?;
            self.add_goto(Operator::GotoF,
                Some(res_address));
            self.parse_return_body(statements)?;
            let index =
                self.jump_list.pop().unwrap();
            let goto_res =
                self.jump_list.pop().unwrap();

            self.add_quad(Quadruple::new_res(Operator::Goto,
                goto_res));
            self.fill_goto_index(index);
            Ok(())
        }
        AstNodeKind::For {
            assignment,
            expr,
            statements,
        } => self.parse_for(&*assignment, &*expr,
            statements, node),
        AstNodeKind::Return(expr) => {
            let return_type =
                self.function().return_type;

```

```

        let (expr_address, _) =
self.assert_expr_type(&*expr, return_type)?;
        self.missing_return = false;

self.add_quad(Quadruple::new_arg(Operator::Return,
expr_address));
        Ok(())
    }
    AstNodeKind::FuncCall { ref name, exprs }
=> {
        if
self.dir_func.functions.get(name).is_some() {
            self.parse_func_call(name, node,
exprs)
        } else {
            let kind =
RaoulErrorKind::UndeclaredFunction2(name.to_string())
;
            Err(RaoulError::new_vec(node,
kind))
        }
        AstNodeKind::Plot {
            name,
            column_1,
            column_2,
        } => {
            self.assert_dataframe(name, node)?;
            let (col_1, _) =
self.assert_expr_type(&*column_1, Types::String)?;
            let (col_2, _) =
self.assert_expr_type(&*column_2, Types::String)?;

self.add_quad(Quadruple::new_args(Operator::Plot,
col_1, col_2));
            Ok(())
        }
        AstNodeKind::Histogram { bins, column,
name } => {
            self.assert_dataframe(name, node)?;
            let (col, _) =
self.assert_expr_type(&*column, Types::String)?;
            let (bins, _) =
self.assert_expr_type(&*bins, Types::Int)?;

self.add_quad(Quadruple::new_args(Operator::Histogram
, col, bins));
            Ok(())
        }
        kind => unreachable!("{kind:?}"),
    }
}

#[inline]
fn update_quad(&mut self, first_quad: usize) {
    self.function_mut().update_quad(first_quad);
}

pub fn parse<'a>(&mut self, node: &AstNode<'a>)
-> Results<'a, ()> {
    match &node.kind {
        AstNodeKind::Main {
            body,
            functions,
            assignments,
        } => {
            self.add_goto(Operator::Goto, None);

RaoulError::create_results(functions.iter().map(|node|
self.parse(node)))?;
            self.fill_goto();
            self.function_name =
"main".to_owned();
            RaoulError::create_results(
                assignments.iter().map(|node|
self.parse_statement(node)),
            )?;
            self.parse_body(body)?;

```

```

self.add_quad(Quadruple::new_empty(Operator::End));
        Ok(())
    }
    AstNodeKind::Function {
        name,
        body,
        return_type,
        ..
    } => {
        self.function_name = name.clone();
        let first_quad =
self.quad_list.len();
        self.update_quad(first_quad);
        if *return_type != Types::Void {
            self.missing_return = true;
        }
        self.parse_body(body)?;
        if self.missing_return {
            let kind =
RaoulErrorKind::MissingReturn(self.function_name.clon
e());
            return
Err(vec![RaoulError::new(node, kind)]);
        }

self.add_quad(Quadruple::new_empty(Operator::EndProc)
);
        Ok(())
    }
    _ => unreachable!(),
}

impl fmt::Display for QuadrupleManager {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) ->
fmt::Result {
        let value: String = self
            .quad_list
            .clone()
            .into_iter()
            .enumerate()
            .map(|(i, quad)| format!("{: <4} -
{:?}\n", i, quad))
            .collect();
        write!(f, "{value}")
    }
}

src/test_parser/mod.rs
use pest::error::Error;
use pest::Parser;

#[derive(Parser)]
#[grammar = "parser/grammar.pest"] // relative to src
struct MyParser;

fn parse(source: &str) -> Result<(), Error<Rule>> {
    if let Err(err) = MyParser::parse(Rule::program,
source) {
        Err(err)
    } else {
        Ok(())
    }
}

#[allow(dead_code)]
pub fn parse_file(filename: &str, debug: bool) ->
Result<(), Error<Rule>> {
    let program =
std::fs::read_to_string(filename).expect(filename);
    if debug {
        println!("Testing {:?}", filename);
    }
    parse(&program)
}
#[deny(dead_code)]

```

```

#[cfg(test)]
mod tests {
    use std::fs::read_dir;

    use super::*;

    #[test]
    fn valid_files() {
        let paths =
read_dir("src/examples/valid").unwrap();
        for path in paths {
            let file_path = path.expect("File must
exist").path();
            let file = file_path.to_str().unwrap();
            assert!(parse_file(file, true).is_ok());
        }
    }

    #[test]
    fn invalid_file() {
        let filename =
"src/examples/invalid/syntax/syntax-error.ra";
        assert!(parse_file(filename, true).is_err());
    }
}

```

## src/vm/gui.rs

```

use eframe::egui;
use egui::{
    plot::{Bar, BarChart, Line, LineStyle, Plot,
Value, Values},
    Color32, InnerResponse, Ui,
};
use polars::prelude::DataFrame;

enum AppType {
    Plot,
    Histogram,
}

pub struct App {
    app_type: AppType,
    bins: Option<usize>,
    data: DataFrame,
    line_style: LineStyle,
}

impl App {
    fn new(data: DataFrame, app_type: AppType, bins:
Option<usize>) -> Self {
        Self {
            app_type,
            data,
            line_style: LineStyle::dotted_loose(),
            bins,
        }
    }

    pub fn new_plot(data: DataFrame) -> Self {
        App::new(data, AppType::Plot, None)
    }

    pub fn new_histogram(data: DataFrame, bins:
usize) -> Self {
        App::new(data, AppType::Histogram,
Some(bins))
    }

    fn plot_line(&self) -> Line {
        let column_1 =
self.data["column_1"].f64().unwrap();
        let column_2 =
self.data["column_2"].f64().unwrap();
        let iter = column_1
            .into_iter()
            .zip(column_2.into_iter())
            .map(|(x, y)| {

```

```

                let x: f64 = x.unwrap();
                let y: f64 = y.unwrap();
                Value::new(x, y)
            });
        Line::new(Values::from_values_iter(iter))
            .color(Color32::BLUE)
            .style(self.line_style)
    }

    fn plot_histogram(&self) -> BarChart {
        let bins = self.bins.unwrap() + 1;
        let mut data: Vec<(f64, f64)> = vec![(0.0,
f64::MAX); bins];
        let column = &self.data["column"];
        let min = column.min::<(f64)>().unwrap();
        let max = column.max::<(f64)>().unwrap();
        let f64_bins =
bins.to_string().parse::<(f64)>().unwrap();
        let step = (max - min) / f64_bins;
        let chunked_arr = column.f64().unwrap();
        chunked_arr.into_iter().for_each(|v| {
            let value = v.unwrap();
            let index: usize = match (value - min) /
step {
                x if x >= f64_bins => bins - 1,
                x =>
x.floor().to_string().parse().unwrap(),
            };
            let (count, start) =
data.get_mut(index).unwrap();
            *count += 1.0;
            if *start > value {
                *start = value;
            }
        });
        let bars: Vec<Bar> = data
            .windows(2)
            .map(|v| {
                let (count, start) = v[0];
                let limit = v[1].1;
                Bar::new(start, count).width((limit -
start) * 0.95)
            })
            .collect();
        BarChart::new(bars)
    }

    fn ui(&self, ui: &mut Ui) -> InnerResponse<()> {
        Plot::new("raoul").show(ui, |plot_ui| match
self.app_type {
            AppType::Plot =>
plot_ui.line(self.plot_line()),
            AppType::Histogram =>
plot_ui.bar_chart(self.plot_histogram()),
        })
    }

    impl eframe::App for App {
        fn update(&mut self, ctx: &egui::Context, _: &mut
eframe::Frame) {
            egui::CentralPanel::default().show(ctx, |ui|
self.ui(ui));
        }
    }
}

```

## src/vm/mod.rs

```

mod gui;

use std::{cmp::Ordering, collections::HashMap};

use polars::{
    datatypes::{AnyValue, DataType},
    io::SerReader,
    prelude::{DataFrame, Series},
};

```

```

use polars_lazy::prelude::{col, pearson_corr,
IntoLazy};

use crate::{
    address::{Address, ConstantMemory, Memory,
PointerMemory, TOTAL_SIZE},
    dir_func::{function::Function,
variable_value::VariableValue},
    enums::Operator,
    quadruple::{quadruple::Quadruple,
quadruple_manager::QuadrupleManager},
};

use self::gui::App;

#[derive(Clone, Debug)]
pub struct VMContext {
    address: usize,
    args: Vec<usize>,
    local_memory: Memory,
    quad_pos: usize,
    size: usize,
    temp_memory: Memory,
}

impl VMContext {
    pub fn new(function: &Function) -> Self {
        let size = function.size();
        let address = function.address;
        let local_memory =
Memory::new(&function.local_addresses);
        let temp_memory =
Memory::new(&function.temp_addresses);
        let quad_pos = function.first_quad;
        let args = function.args.iter().map(|v|
v.0).collect();
        Self {
            address,
            args,
            local_memory,
            quad_pos,
            size,
            temp_memory,
        }
    }
}

pub type VMResult<T> = std::result::Result<T,
&'static str>;

#[derive(Debug)]
pub struct VM {
    call_stack: Vec<VMContext>,
    constant_memory: ConstantMemory,
    contexts_stack: Vec<VMContext>,
    debug: bool,
    functions: HashMap<usize, Function>,
    global_memory: Memory,
    pointer_memory: PointerMemory,
    pub messages: Vec<String>,
    quad_list: Vec<Quadruple>,
    stack_size: usize,
    data_frame: Option<DataFrame>,
}

const STACK_SIZE_CAP: usize = 1024;

fn cast_to_f64(v: &AnyValue) -> f64 {
    match v {
        AnyValue::Float64(v) => *v,
        AnyValue::Float32(v) =>
(*v).try_into().unwrap(),
        AnyValue::Int64(v) =>
v.to_string().parse::<f64>().unwrap(),
        _ => unreachable!(),
    }
}

```

```

fn safe_address(value: &Option<VariableValue>) ->
VMResult<VariableValue> {
    match value {
        Some(v) => Ok(v.clone()),
        None => Err("Found initialized value"),
    }
}

#[inline]
fn min(c: &Series) -> f64 {
    c.min().unwrap_or(0.0)
}

#[inline]
fn max(c: &Series) -> f64 {
    c.max().unwrap_or(0.0)
}

impl VM {
    pub fn new(quad_manager: &QuadrupleManager,
debug: bool) -> Self {
        let constant_memory =
quad_manager.memory.clone();
        let functions =
quad_manager.dir_func.functions.clone();
        let global_fn =
quad_manager.dir_func.global_fn.clone();
        let pointer_memory =
quad_manager.pointer_memory.clone();
        let global_memory =
Memory::new(&global_fn.addresses);
        let quad_list =
quad_manager.quad_list.clone();
        let main_function =
functions.get("main").unwrap();
        let stack_size = main_function.size();
        let initial_context =
VMContext::new(main_function);
        Self {
            call_stack: vec![],
            constant_memory,
            contexts_stack: vec![initial_context],
            data_frame: None,
            debug,
            functions: functions
                .into_iter()
                .map(|(_, function)|
(function.first_quad, function))
                .collect(),
            global_memory,
            messages: Vec::new(),
            pointer_memory,
            quad_list,
            stack_size,
        }
    }

    fn add_call_stack(&mut self, function: &Function)
-> VMResult<()> {
        self.stack_size += function.size();
        if self.stack_size > STACK_SIZE_CAP ||
self.contexts_stack.len() == STACK_SIZE_CAP {
            return Err("Stack overflow!");
        }
    }

    self.call_stack.push(VMContext::new(function));
    Ok(())
}

#[inline]
fn current_context(&self) -> &VMContext {
    self.contexts_stack.last().unwrap()
}

#[inline]
fn local_addresses(&self) -> &Memory {
    &self.current_context().local_memory
}

```

```

#[inline]
fn temp_addresses(&self) -> &Memory {
    &self.current_context().temp_memory
}

#[inline]
fn current_context_mut(&mut self) -> &mut
VMContext {
    self.contexts_stack.last_mut().unwrap()
}

#[inline]
fn local_addresses_mut(&mut self) -> &mut Memory
{
    &mut self.current_context_mut().local_memory
}

#[inline]
fn temp_addresses_mut(&mut self) -> &mut Memory {
    &mut self.current_context_mut().temp_memory
}

#[inline]
fn update_quad_pos(&mut self, quad_pos: usize) {
    self.current_context_mut().quad_pos =
quad_pos;
}

#[inline]
fn get_function(&self, first_quad: usize) ->
&Function {
    self.functions.get(&first_quad).unwrap()
}

fn get_current_quad(&self) -> Quadruple {
    let quad_pos =
self.current_context().quad_pos;
    *self.quad_list.get(quad_pos).unwrap()
}

fn get_value(&self, address: usize) ->
VMResult<VariableValue> {
    match address / TOTAL_SIZE {
        0 =>
safe_address(self.global_memory.get(address)),
        1 =>
safe_address(self.local_addresses().get(address)),
        2 =>
safe_address(self.temp_addresses().get(address)),
        3 =>
Ok(self.constant_memory.get(address).clone()),
        _ => {
            let address =
self.pointer_memory.get(address);
            self.get_value(address)
        }
    }
}

fn write_value(&mut self, value: VariableValue,
address: usize) -> VMResult<()> {
    let determinant = address / TOTAL_SIZE;
    if determinant >= 4 {
        self.pointer_memory.write(address,
value);
        return Ok(());
    }
    let memory = match determinant {
        0 => &mut self.global_memory,
        1 => self.local_addresses_mut(),
        2 => self.temp_addresses_mut(),
        _ => unreachable!(),
    };
    memory.write(address, &value)
}

fn process_assign(&mut self) -> VMResult<()> {

```

```

        let quad = self.get_current_quad();
        let value =
self.get_value(quad.op_1.unwrap())?;
        let mut assignee = quad.res.unwrap();
        if assignee.is_pointer_address() {
            assignee =
self.pointer_memory.get(assignee);
        }
        self.write_value(value, assignee)
    }

    fn print_message(&mut self, message: &str) {
        self.messages.push(message.to_string());
        let separator = if message.contains('\n') {
"" } else { " ";
        print!("{}", message){separator}");
    }

    fn process_print(&mut self) -> VMResult<()> {
        let quad = self.get_current_quad();
        let value =
self.get_value(quad.op_1.unwrap())?;
        self.print_message(&format!("{}", value));
        Ok(())
    }

    fn process_read(&mut self) -> VMResult<()> {
        let quad = self.get_current_quad();
        let value = VariableValue::from_stdin();
        self.write_value(value, quad.res.unwrap())
    }

    fn unary_operation<F>(&mut self, f: F) ->
VMResult<()>
    where
        F: FnOnce(VariableValue) -> VariableValue,
    {
        let quad = self.get_current_quad();
        let a = self.get_value(quad.op_1.unwrap())?;
        let value = f(a);
        self.write_value(value, quad.res.unwrap())
    }

    fn binary_operation<F>(&mut self, f: F) ->
VMResult<()>
    where
        F: FnOnce(VariableValue, VariableValue) ->
VMResult<VariableValue>,
    {
        let quad = self.get_current_quad();
        let a = self.get_value(quad.op_1.unwrap())?;
        let b = self.get_value(quad.op_2.unwrap())?;
        let value = f(a, b)?;
        self.write_value(value, quad.res.unwrap())
    }

    fn comparison(&mut self) -> VMResult<()> {
        let quad = self.get_current_quad();
        let a = self.get_value(quad.op_1.unwrap())?;
        let b = self.get_value(quad.op_2.unwrap())?;
        let ord = a.partial_cmp(&b);
        let res = match ord {
            None => false,
            Some(ord) => match quad.operator {
                Operator::Lt => ord ==
Ordering::Less,
                Operator::Lte => ord !=
Ordering::Greater,
                Operator::Gt => ord ==
Ordering::Greater,
                Operator::Gte => ord !=
Ordering::Less,
                Operator::Eq => ord ==
Ordering::Equal,
                Operator::Ne => ord !=
Ordering::Equal,
                _ => unreachable!(),
            },

```

```

    };
    let value = VariableValue::Bool(res);
    self.write_value(value, quad.res.unwrap())
}

fn conditional_goto(&mut self, approved: bool) ->
VMResult<usize> {
    let quad = self.get_current_quad();
    let cond =
self.get_value(quad.op_1.unwrap())?;
    let quad_pos =
self.current_context().quad_pos;
    if bool::from(cond) == approved {
        return Ok(quad.res.unwrap() - 1);
    }
    Ok(quad_pos)
}

fn process_inc(&mut self) -> VMResult<()> {
    let quad = self.get_current_quad();
    let a = self.get_value(quad.res.unwrap())?;
    let value = a.increase()?;
    self.write_value(value, quad.res.unwrap())
}

fn process_era(&mut self) -> VMResult<()> {
    let quad = self.get_current_quad();
    let first_quad = quad.op_2.unwrap();
    let function =
self.get_function(first_quad).clone();
    self.add_call_stack(&function)
}

fn process_go_sub(&mut self) {
    let quad_pos =
self.current_context().quad_pos;
    self.update_quad_pos(quad_pos + 1);
    let call = self.call_stack.pop().unwrap();
    self.contexts_stack.push(call);
}

fn process_end_proc(&mut self) {
    let context =
self.contexts_stack.pop().unwrap();
    self.stack_size -= context.size;
}

#[inline]
fn current_call(&self) -> &VMContext {
    self.call_stack.last().unwrap()
}

#[inline]
fn current_call_mut(&mut self) -> &mut VMContext
{
    self.call_stack.last_mut().unwrap()
}

fn write_value_param(&mut self, value:
&VariableValue, address: usize) -> VMResult<()> {
    let memory = match address / TOTAL_SIZE {
        1 => &mut
self.current_call_mut().local_memory,
        val => unreachable!("{val}"),
    };
    memory.write(address, value)
}

fn process_param(&mut self) -> VMResult<()> {
    let quad = self.get_current_quad();
    let value =
self.get_value(quad.op_1.unwrap())?;
    let index = quad.res.unwrap();
    let address =
*self.current_call().args.get(index).unwrap();
    self.write_value_param(&value, address)
}

```

```

#[inline]
fn get_context_global_address(&self) -> usize {
    self.current_context().address
}

fn process_return(&mut self) -> VMResult<()> {
    let quad = self.get_current_quad();
    let value =
self.get_value(quad.op_1.unwrap())?;
    let address =
self.get_context_global_address();
    self.write_value(value, address)?;
    self.process_end_proc();
    Ok(())
}

fn process_ver(&mut self) -> VMResult<()> {
    let quad = self.get_current_quad();
    let index =
self.get_value(quad.op_1.unwrap())?;
    let limit =
self.get_value(quad.op_2.unwrap())?;
    if limit <= index ||
VariableValue::Integer(0) > index {
        return Err("Index out of range for
array");
    }
    Ok(())
}

fn read_csv(&mut self) -> VMResult<()> {
    let quad = self.get_current_quad();
    let filename =
String::from(self.get_value(quad.op_1.unwrap())?);
    let res =
polars::io::csv::CsvReader::from_path(&filename);
    if res.is_err() {
        return Err("Could not read the file");
    }
    let res =
res.unwrap().has_header(true).finish();
    if res.is_err() {
        return Err("File is not a valid CSV");
    }
    self.data_frame = Some(res.unwrap());
    Ok(())
}

fn get_dataframe(&self) -> VMResult<&DataFrame> {
    if self.data_frame.is_none() {
        return Err("No data frame was created.
You need to create one using `read_csv`");
    }
    let data_frame =
self.data_frame.as_ref().unwrap();
    Ok(data_frame)
}

fn pure_df_operation(&mut self) -> VMResult<()> {
    let quad = self.get_current_quad();
    let data_frame = self.get_dataframe()?;
    let value = match quad.operator {
        Operator::Rows => data_frame.shape().0,
        Operator::Columns =>
data_frame.shape().1,
        _ => unreachable!(),
    }
    .into();
    self.write_value(value, quad.res.unwrap())
}

fn unary_df_operation<F>(&mut self, f: F) ->
VMResult<()>
where
    F: FnOnce(&Series) -> f64,
{
    let quad = self.get_current_quad();
}

```



```

        let column_name =
String::from(self.get_value(quad.op_1.unwrap())?);
let data_frame = self.get_dataframe()?;
let column = data_frame.column(&column_name);
if column.is_err() {
    return Err("Dataframe key not found in
file");
}
let value = f(column.unwrap()).into();
self.write_value(value, quad.res.unwrap())
}

fn correlation(&mut self) -> VMResult<()> {
let quad = self.get_current_quad();
let data_frame = self.get_dataframe()?;
let col_1_name =
String::from(self.get_value(quad.op_1.unwrap())?);
let col_2_name =
String::from(self.get_value(quad.op_2.unwrap())?);
let temp = data_frame
    .clone()
    .lazy()
    .select([pearson_corr(
col(&col_1_name).cast(DataType::Float64),
col(&col_2_name).cast(DataType::Float64),
    )
    .alias("correlation"))
    .collect()
    .unwrap());
let value =
cast_to_f64(&temp.column("correlation").unwrap().get(
0)).into();
self.write_value(value, quad.res.unwrap())
}

fn plot(&mut self) -> VMResult<()> {
let quad = self.get_current_quad();
let data_frame = self.get_dataframe()?;
let col_1_name =
String::from(self.get_value(quad.op_1.unwrap())?);
let col_2_name =
String::from(self.get_value(quad.op_2.unwrap())?);
let temp = data_frame
    .clone()
    .lazy()
    .select([
col(&col_1_name).cast(DataType::Float64).alias("column_1"),
col(&col_2_name).cast(DataType::Float64).alias("column_2"),
    ])
    .collect()
    .unwrap();
let app = App::new_plot(temp);
eframe::run_native(
    "Raoul",
    eframe::NativeOptions::default(),
    Box::new(|_cc| Box::new(app)),
);
}

fn histogram(&mut self) -> VMResult<()> {
let quad = self.get_current_quad();
let data_frame = self.get_dataframe()?;
let col_name =
String::from(self.get_value(quad.op_1.unwrap())?);
let bins_value =
self.get_value(quad.op_2.unwrap())?;
let bins = match bins_value {
    VariableValue::Integer(a) if a <= 0 =>
Err("The amount of bins should be positive"),
    _ => Ok(usize::from(bins_value)),
};
let temp = data_frame
        .clone()
        .lazy()
        .select([col(&col_name).cast(DataType::Float64).alias(
"column")])
        .collect()
        .unwrap();
let app = App::new_histogram(temp, bins);
eframe::run_native(
    "Raoul",
    eframe::NativeOptions::default(),
    Box::new(|_cc| Box::new(app)),
);
}

pub fn run(&mut self) -> VMResult<()> {
loop {
let mut quad_pos =
self.current_context().quad_pos;
if self.debug {
self.print_message(&format!("Quad -
{quad_pos}\n"));
}
let quad =
self.quad_list.get(quad_pos).unwrap();
match quad.operator {
    Operator::End => break,
    Operator::Goto => {
quad_pos = quad.res.unwrap() - 1;
Ok(())
}
    Operator::Assignment =>
self.process_assign(),
    Operator::Print =>
self.process_print(),
    Operator::PrintNl => {
self.print_message("\n");
Ok(())
}
    Operator::Read =>
self.process_read(),
    Operator::Or =>
self.binary_operation(|a, b| Ok(a | b)),
    Operator::And =>
self.binary_operation(|a, b| Ok(a & b)),
    Operator::Sum =>
self.binary_operation(|a, b| a + b),
    Operator::Minus =>
self.binary_operation(|a, b| a - b),
    Operator::Times =>
self.binary_operation(|a, b| a * b),
    Operator::Div =>
self.binary_operation(|a, b| a / b),
    Operator::Lt
| Operator::Lte
| Operator::Gt
| Operator::Gte
| Operator::Eq
| Operator::Ne => self.comparison(),
    Operator::Not =>
self.unary_operation(|a| !a),
    Operator::GotoF => {
quad_pos =
self.conditional_goto(false)?;
Ok(())
}
    Operator::Inc => self.process_inc(),
    Operator::Era => self.process_era(),
    Operator::GoSub => {
self.process_go_sub();
continue;
}
    Operator::EndProc => {
self.process_end_proc();
continue;
}
    Operator::Param =>
self.process_param(),
}
}

```

```

        Operator::Return => {
            self.process_return()?;
            continue;
        }
        Operator::Ver => self.process_ver(),
        Operator::ReadCSV => self.read_csv(),
        Operator::Rows | Operator::Columns =>
self.pure_df_operation(),
        Operator::Average =>
self.unary_df_operation(|c| c.mean().unwrap_or(0.0)),
        Operator::Std => {
            self.unary_df_operation(|c|
cast_to_f64(&c.std_as_series().get(0)))
        }
        Operator::Variance => {
            self.unary_df_operation(|c|
cast_to_f64(&c.var_as_series().get(0)))
        }
        Operator::Median =>
self.unary_df_operation(|c|
c.median().unwrap_or(0.0)),
        Operator::Min =>
self.unary_df_operation(min),
        Operator::Max =>
self.unary_df_operation(max),
        Operator::Range =>
self.unary_df_operation(|c| max(c) - min(c)),
        Operator::Corr => self.correlation(),
        Operator::Plot => self.plot(),
        Operator::Histogram =>
self.histogram(),
    }?;
    self.update_quad_pos(quad_pos + 1);
}
Ok(())
}
}

```