

Carlos Eduardo Govea González
A00819647

Índice

Índice	1
DESCRIPCIÓN DEL PROYECTO:	2
Bitácora:	2
Lista de commits:	5
Reflexión:	6
DESCRIPCION DE ALICE	7
Lista de Errores:	8
COMPILACIÓN DE ALICE	11
Análisis Léxico	11
Palabras Reservadas:	11
Tokens sencillos	11
Patrones de construcción:	11
Análisis de Sintaxis	12
Generación de Código Intermedio y Análisis Semántico	16
Direcciones Virtuales	16
Diagramas de Sintaxis	16
Administración de Memoria en Compilación	26
Tabla de consideraciones semánticas	26
EJECUCIÓN DE ALICE	27
Administración de Memoria en Ejecución	27
PRUEBAS DEL FUNCIONAMIENTO	28
Fibonacci (recursivo e iterativo)	28
Factorial (recursivo e iterativo)	29
Funciones de data science	30
Find	32

DESCRIPCIÓN DEL PROYECTO:

El propósito del lenguaje Alice es crear una sintaxis fácil de entender para un lenguaje de programación orientado a ciencia de datos, con un enfoque estadístico que genere rápidamente resultados de las fórmulas más utilizadas de estadísticos descriptivos. El alcance del proyecto en materia de programación incluye 3 tipos de datos (int, float, string), lógica booleana, 3 diferentes tipos de ciclos (while, do while, for), funciones estadísticas, declaración y manejo de arreglos y matrices, manejo de módulos definidos por usuarios, recursividad, y estatutos de I/O.

Para poder desempeñar su función como lenguaje de programación estadístico, será necesario que Alice cuente con funciones de estadísticos descriptivos, tales como la media, mediana, moda, varianza, desviación estándar y rango de los valores de un arreglo. Por consiguiente, también se espera que el lenguaje pueda soportar tipos de datos enteros, flotantes y los arreglos de esos tipos de datos. Adicionalmente, se espera que maneje strings en caso de que el usuario requiera guardar mensajes que desee imprimir al usuario o que el usuario haya insertado a través de una operación de read.

Serán necesarias también operaciones aritméticas, lógica booleana y ciclos para poder hacer procedimientos que dependan de una condición para ejecutarse, o para procesos que necesiten repetirse múltiples veces. La modularización de procedimientos puede ayudar a reducir la complejidad de código de procesos estadísticos, por lo que tener funciones void y que retornen tipos también se encuentra dentro del alcance del proyecto.

Para interactuar con el usuario también se designaron operaciones de print e input para poder recibir información proveniente de fuera del ambiente de ejecución e integrarla a las operaciones que esté realizando un programa.

Para probar estas funcionalidades se diseñaron casos de prueba tanto de eficiencia como de la utilización de funciones estadísticas. Tanto operaciones utilizando los valores de retorno de las funciones estadísticas dentro de expresiones, como simplemente imprimir el valor de retorno de estas, así como el uso de funciones creadas por usuario que contienen recursividad y cálculos complejos, como factoriales y la secuencia fibonacci, para probar la eficiencia.

Bitácora:

Semana	Bitácora
Semana 1	En el primer avance se generaron los tokens en el archivo de <code>alice_lex.py</code> y las primeras reglas en <code>alice_yacc.py</code> de acuerdo a los diagramas realizados.
Semana 1	Se modificaron las reglas debido a que se encontraron problemas parseando archivos.
Semana 2	Se hizo una reestructuración del archivo <code>alice_yacc</code> para ajustar el formato de las reglas a LALR en vez del formato LL que se usó antes.
Semana 2	Se creó la sección de main para separar el main del ambiente global de donde

	empieza el ambiente local. Parece correr sin ningún problema.
Semana 2	Se creó el archivo structs.py que contiene clases para representar la tabla de variables, el directorio de funciones, la pila de saltos, la de operadores y la de operandos.
Semana 3	Se encontró un error al momento de intentar parsear estatutos no lineales que causaba un tiempo de ejecución enorme. Se separó la lógica de parseo a un archivo de python por separado llamado compile.py
Semana 3	Se agregaron más reglas de parseo para módulos, así como estatutos de I/O.
Semana 4	Se arregló el bug que causaba enormes tiempos de ejecución, modificando la regla que empieza los bloques de estatutos para limitarlos entre un par de tokens y así evitar ambigüedad.
Semana 4	Se creó el cubo semántico, utilizando un diccionario de arreglos que utiliza el operador como llave y los índices del 0, 1, 2 y 3 para representar los tipos int, float, string y bool respectivamente.
Semana 4	Se empezaron a agregar un par de puntos neurálgicos para las expresiones aritméticas y de lógica booleana con el objetivo de meter valores en las pilas pertinentes.
Semana 4	Se terminaron de agregar los puntos neurálgicos de todos los estatutos lineales (expresiones, input, print, asignaciones y declaraciones). Se agregó la clase de cuádruplos en structs.py para empezar a generarlos en el compilador.
Semana 5	Se creó un sistema de logging que genera un archivo de texto con información útil para debuggear llamado log.txt, así mismo se generó la lógica para exportar los cuádruplos en formato JSON, con la esperanza de poder crear la máquina virtual en otro lenguaje de programación.
Semana 5	Se terminaron de agregar los puntos neurálgicos y generación de cuádruplos para los estatutos condicionales, y los ciclos for, while y do while. Así mismo se agregó la creación del cuádruplo Goto Main al principio de la compilación del programa.
Semana 5	Se reestructuraron los rangos de memoria en structs.py para remover los rangos de temporales string y, en vez de eso, dedicarle más espacio a temporales booleanos debido a que el lenguaje Alice no tiene la necesidad de variables temporales string.
Semana 6	Se generó la lógica semántica detrás de los estatutos de retorno, así como verificación de que se estén ejecutando en un ambiente que no sea void o global.
Semana 6	Se generaron los puntos neurálgicos y código intermedio para la declaración y llamada de módulos, así como las asignaciones de los valores de retorno de módulos tipados a variables que sean del mismo tipo, y la inclusión de estas mismas en expresiones aritméticas.

Semana 6	Se limitó el alcance de la función clear incluida en la clase que maneja la memoria dentro de structs.py para limpiar los contadores de memoria local, lo cual sirvió para resetear los contadores en ambientes locales para cada módulo declarado y el main.
Semana 6	Se modificó la lógica detrás de la exportación de cuádruplos en formato JSON para incluir el directorio de funciones y la tabla de constantes. Se modificaron los cuádruplos generados por los estatutos de print e input para optimizar su funcionamiento, reduciendo la cantidad de cuádruplos que se generaban para ambos.
Semana 6	Se creó la máquina virtual utilizando el lenguaje de programación Julia bajo el nombre de vm.jl. Por el momento solo procesa estatutos de I/O así como los Goto incondicionales.
Semana 6	Se creó un script en bash llamado alice el cual funciona como reemplazo y mejora del archivo compile.py, el cual corre el compilador alice_yacc.py y, posteriormente, la máquina virtual hecha en Julia. Se agregó el prototipo del repl en alice.
Semana 7	Se corrigió en el cubo semántico los tipos de retorno para las expresiones de división y la dirección en donde se guardan los valores para los operadores ++ y --.
Semana 7	Se creó el archivo virtual_memory.jl el cual contiene las clases y estructurar necesarias para la simulación de memoria dentro de la máquina virtual, ahora llamada virtual_machine.jl. Se separó la memoria en 2 tipos: Persistent y Temporary, los cuales son de tipo abstracto Memory.
Semana 7	Se generaron funciones auxiliares para la extracción y almacenamiento de valores en memoria, así como los rangos de memoria y operadores que utiliza Alice para poder facilitar la ejecución de los cuádruplos en la máquina virtual.
Semana 7	Se detectó un bug en funciones recursivas en donde el compilador tenía problemas pidiendo recursos de una función que no se había terminado de definir. Así mismo, se detectaron problemas atando los valores de retorno de una llamada de función cuando se tienen muchas en una expresión.
Semana 7	Se realizaron ajustes a la manera en la que se generan los cuádruplos de llamadas a función dentro de un módulo para tomar en cuenta casos de recursividad. Así mismo se incluyó un cuádruplo para atar el valor de la variable global que representa el valor de retorno de una función a una variable temporal para no perder el valor en una función.
Semana 8	Se empezó a generar la lógica detrás de los arreglos, incluyendo la inclusión de variables tipo apuntador, la generación de cuádruplos de verificación de valores y la declaración correcta de arreglos.
Semana 8	El compilador exitosamente pasó las pruebas de fibonacci y factorial, tanto

	recursivos como utilizando ciclos. Se descubrió un bug al momento de indexar a un arreglo utilizando una variable que está sufriendo cambios a través de un ciclo. Se terminó la lógica de la creación e indexación de matrices.
Semana 8	Se generaron los cuádruplos pertinentes para la ejecución de funciones de data science, incluyendo todas las prometidas al momento de generar la propuesta del proyecto. Se generó un archivo de pruebas que corrió exitosamente.
Semana 8	Se utilizó algo de tiempo de sobra para agregar un par de funciones adicionales a las del alcance original del proyecto, como la suma, el mínimo y el máximo de un arreglo, así como mejoras estáticas y finalización del repl de Alice, mejorando la calidad del ASCII art que genera y eliminando la necesidad de tenerlo guardado en un archivo por separado.
Semana 8	Se agregaron 5 funciones de graficación como extra fuera del alcance del proyecto, así como una función de lectura y parseo de contenidos de archivo para cargar información a arreglos. Se corrigió el bug de indexación que ocurría cuando se indexaba a un arreglo usando una variable atada a un ciclo.

Lista de commits:

Commit ID	Fecha	Comentario
ca44e97	09/Mar/2022	Initial commit
0569d87	11/Mar/2022	Add lex, basic grammar and tests
c204e8c	23/Mar/2022	Fix rules on yacc file
2352600	05/Abr/2022	Set theme jekyll-theme-hacker
eb0117e	05/Abr/2022	Add GitHub page to repo
98c539c	05/Abr/2022	Fix GitHub Page file
90b425e	05/Abr/2022	Renamed GitHub Page file
528d0b8	05/Abr/2022	Delete index.md
29c529b	05/Abr/2022	Delete _config.yml
fca2838	13/Abr/2022	Add new tokens, expand formal grammar
6e417c9	24/Abr/2022	Add structs, expand grammar and tokens
43dcdf0	03/May/2022	Add semantic to linear stmts, create sem cube

71fd647	04/May/2022	Add intermediate code for non-linear stmts
102e70c	08/May/2022	Add more intermediate code, modify outputs
7a734b6	11/May/2022	Add intermediate code for modules
d1fde50	13/May/2022	Create VM, add bash interface
90ba63e	13/May/2022	Fix README.md
fe6c0b6	19/May/2022	Add functionality to VM
266e203	22/May/2022	Add module support and recursion
de6587f	26/May/2022	Fix recursive calls, add array support
d1c4891	28/May/2022	Fully implement arrays and stat functions
2584a7a	28/May/2022	Fix README
6813ab8	29/May/2022	Update repl, fix sem cube, add tests
bbdcd6b	29/May/2022	Add matrix to language
088d455	29/May/2022	Fix README and executable
d0eeaae	01/Jun/2022	Add plots and readfile function
78ac07b	02/Jun/2022	Fixed array indexation bug
4821818	06/Jun/2022	Add documentation

Reflexión:

Crear un lenguaje de programación no es una tarea fácil, pero tampoco lo llamaría una tarea imposible. Definitivamente, lo más importante durante el proyecto fue la constancia y la dedicación diaria a mejorar o crear nuevos procesos para desarrollar una compilación y ejecución eficiente. Junto con el reto, sin embargo, viene una enorme cantidad de entendimiento de los procesos que utilizan los lenguajes de programación existentes para proveer la funcionalidad que nos ofrecen a los programadores, lo cual ayuda bastante a idear maneras de generar código más eficiente, ya tomando en cuenta este conocimiento que recibimos. Tomando en cuenta mis conocimientos al final de este semestre, y de mi carrera, puedo ver incluso maneras en las que pude haber mejorado mi diseño, mejorar o como expandirlo para agregar más utilidades. Definitivamente, me gustó llevar la materia, y siento que el conocimiento que obtuve al realizar este proyecto durante todo un semestre me ayudó a forjar una mentalidad de trabajo más constante, más responsable, y a obtener una sensibilidad o intuición acerca de los procesos internos de cualquier lenguaje o pieza de software con la que interactúo.



DESCRIPCION DE ALICE

Alice es un lenguaje de programación enfocado en un paradigma de programación estructurado y procedural, orientado a uso estadístico y para la ciencia de datos. Es un lenguaje fuertemente tipado que cuenta con los tipos básicos de *int*, *float* y *string*, y maneja operaciones booleanas, pero no maneja el *bool* como un tipo de dato que se pueda almacenar o imprimir en pantalla. Estos tipos no solamente se utilizan para las variables atómicas, arreglos y matrices, sino también para las funciones creadas por usuarios, llamadas *módulos*, los cuales también incluyen el tipo *void* para la declaración de funciones que no tienen valor de retorno alguno.

Alice cuenta con 17 operadores que se pueden usar tanto en expresiones aritméticas, como en expresiones de lógica booleana. A continuación se listarán los operadores, junto con su asociatividad y precedencia:

Precedencia	Operador	Descripción	Asociatividad
1	x++, x--	Incremento y decremento	Izq a Der
2	- x	Menos unario	Der a Izq
3	x^y, x*y, x/y	Potenciación, Multiplicación y división	Izq a Der
4	x+y, x-y	Suma y resta binaria	Izq a Der
5	x<y, x<=y, x>y, x>=y	Operadores relacionales	Izq a Der
5	x==y, x!=y	Operadores de igualdad	Izq a Der
6	x and y	AND Lógico	Izq a Der
7	x or y	OR Lógico	Izq a Der
8	x <- y	Asignación	Der a Izq

* El operador de asignación solo puede aparecer una vez por expresión.

En cuanto a estructuras de control de flujos, Alice soporta estatutos condicionales bajo la nomenclatura *if-then-else*, así como 3 diferentes tipos de ciclos (sin contar la recursividad): *while*, *do while*, y *for loop*. Dentro de estas estructuras de control se puede poner cualquier estatuto de ejecución simple, como *inputs*, *prints*, *asignaciones*, *llamadas a módulos* y *expresiones*, así como otras estructuras de control y estatutos complejos.

Finalmente, Alice provee herramientas a los programadores para analizar datos a través de funciones de estadísticos descriptivos, de carga de datos y de graficación. Las funciones de estadísticos descriptivos son: *tamaño*, *media*, *mediana*, *moda*, *varianza*, *desviación estándar*, *rango*, *suma*, *mínimo* y *máximo*. Además, cuenta con funciones para generar gráficas de barras, de violín, histogramas, diagrama de cajas y diagramas de dispersión.

Para mayor información acerca de las funciones del lenguaje y su comportamiento, consulte el Manual de Referencia en: <https://charlesgovea.github.io/alice/>

Lista de Errores:

No.	Origen	Mensaje de Error
1	Compilación: Cuando se utiliza una variable que no ha sido creada	"Error! Variable '[var_name]' wasn't declared!"
2	Compilación: Cuando se declaran demasiadas globales de X tipo	"Error! Too many global [X] variables!"
3	Compilación: Cuando se declaran demasiadas locales de X tipo	"Error! Too many local [X] variables!"
4	Compilación: Cuando se crean demasiadas temporales de X tipo	"Error! Too many temporal [X] variables!"
5	Compilación: Cuando se crean demasiados apuntadores	"Error! Too many pointers!"
6	Compilación: Cuando se declaran demasiadas constantes de X tipo	"Error! Too many [X] literals!"
7	Compilación: Cuando se intenta reusar un nombre para una variable	"Error! Variable '[ID]' already declared!"
8	Compilador: Cuando se intenta indexar en una variable atómica	"Error! Variable '{id}' is atomic!"
9	Compilación: Cuando se intenta declarar un arreglo de tamaño < 2	"Error! Array's size isn't valid!"
10	Compilación: Cuando se declara matriz con $n < 2$ para alguna dimensión	"Error! Matrix's [n] dimension's value isn't valid!"
11	Compilación: Cuando se intenta indexar en un arreglo con un valor no entero	"Index error on variable '[var_name]'! Indexes must be integer values."
12	Compilación: Cuando se intenta crear una matriz sin proveer 2 números	"Error! [var_name] is a matrix, expected another index."
13	Ejecución: Cuando una variable sin valor participa en una expresión o impresión a terminal	"Semantic error! Variable was never assigned a value!"
14	Compilación: Cuando se intenta imprimir un valor no soportado	"Incorrect value provided to print function!"
15	Ejecución: Cuando el valor input no es del mismo tipo que la variable destino	"Error! Type mismatch on input, expected [type]."

16	Compilación: Cuando se usa un mensaje en input que no es tipo string	"Error while performing input! '[ID]' is not a string!"
17	Compilación: Cuando la dirección que recibe la función mirror no es un string	"Semantic error in first argument of mirror function! Expected string value."
18	Compilación: Cuando el destino de mirror no es un arreglo unidimensional	"Error! Variable '[var_name]' is not an unidimensional array!"
19	Compilación: Cuando el destino de mirror es un arreglo unidimensional de strings	"Semantic error in second argument of mirror function! Expected an unidimensional array of types int or float."
20	Ejecución: Cuando el formato del archivo mirror es inadecuado o contiene valores de diferentes tipos ente sí o de la variable destino	"Error file [file_name] contains incorrect values or does not match the type of the storage value!"
21	Ejecución: Cuando el archivo que se quiere copiar no fue encontrado	"Error file [file_name] doesn't exist!"
22	Compilación: Cuando se intenta usar una función estadística en arreglo de strings	"Error! Variable '[var_name]' does not contain numeric values!"
23	Compilación: Cuando se intenta exportar la gráfica en un formato incorrecto	"Error! [ext] export format not supported! Supported formats are [extensions]."
24	Compilación: Cuando el nombre del archivo de exportación no tiene el formato correcto	"Error! Incorrect filename provided."
25	Compilación: Cuando se quiere reusar un nombre existente para una función	"Error! Module '[fun_name]' already exists!"
26	Compilación: Cuando se quiere usar el nombre de una variable para una función	"Error! Cannot create module '[fun_name]', name already reserved!"
27	Compilación: Cuando se quiere usar un return en ambiente global, main o void	"Error! Cannot return in '[env]' context!"
28	Compilación: Cuando se quiere retornar un valor que no es del mismo tipo que la función en donde se encuentra	"Error! Return type mismatch! Expected [env], got [type]."
29	Compilación: Cuando se quiere llamar a una función que no existe	"Error! Module '[fun_name]' does not exist!"
30	Compilación: Cuando no se le provee a una llamada de función argumentos, pero si esperaba	"Error! No arguments received! Expected [n]."

31	Compilación: Cuando se le provee a una llamada argumentos cuando no esperaba ninguno	"Error! Call to '[fun_name]' received too many arguments! Expected no arguments, received [n]."
32	Compilación: Cuando se le provee a una llamada una cantidad diferente de argumentos de los que esperaba	"Error! Call to '[fun_name]' received incorrect amount of arguments! Expected [n], received [m]."
33	Compilación: Cuando el tipo del argumento local es diferente del formal	"Error! Type mismatch in function call to '[fun_name]' with parameter [n]!"
34	Compilación: Cuando la variable que se usará para iterar en un for no es entera	"Semantic error! Type mismatch: Expected int, got [type]."
35	Compilación: Cuando la expresión de un for no es del mismo tipo que la variable de la iteración	"Semantic error! Type mismatch: Got [type1] and [type2]"
36	Compilación: Cuando se usa una expresión no booleana en una condición o ciclo	"Error! Cannot evaluate the truth value of [type] expressions!"
37	Compilación: Cuando se intenta comparar valores booleanos o strings con cualquier otro tipo	"Semantic error! Cannot compare [type1] and [type2]!"
38	Compilación: Cuando se intenta usar expresiones no booleanas con operandos and u or	"Semantic error! Expected two bool operands, got [type1] and [type2]!"
39	Compilación: Cuando participan bools o strings en expresiones aritméticas	"Semantic error! Cannot perform arithmetic operations between [type1] and [type2]!"
40	Compilación: Cuando se usa el menos unario en bools o strings	"Semantic error! [type] data cannot be turned negative!"
41	Compilación: Cuando se usan los operandos ++ o -- en bools o strings	"Semantic error! Cannot perform arithmetic increment or decrease on [type]!"
42	Ejecución: Cuando el resultado de una expresión aritmética retorna un número complejo o puramente imaginario	"Error! Expression returned imaginary result!"
43	Ejecución: Cuando el resultado de una expresión es una división entre 0	"Error! Division by zero is undefined!"
44	Compilación: Cuando se inserta un token no reconocido por el lenguaje	"Line [lineno], illegal token: [token]."
45	Compilación: Cuando la compilación de código es interrumpida	"Unexpected end of input."
46	Compilación: Cuando el archivo a compilar no es encontrado por el compilador	"Error! File [file_name] not found!"

COMPILACIÓN DE ALICE

Alice fue desarrollado dentro de un ambiente Manjaro Linux, sin embargo, también ha sido probado exitosamente en otros distros de Linux como Ubuntu y Arch, así como Windows 10. El compilador de Alice fue desarrollado en Python, por lo que será necesario tener instaladas las librerías de PLY para Python y tener la versión más reciente de dicho lenguaje para poder compilar el código.

Análisis Léxico

Palabras Reservadas:

'begin' : 'BEGIN'	'endprog' : 'ENDPROG',	'std' : 'STD',
'main' : 'MAIN'	'end' : 'END',	'range' : 'RANGE',
'let' : 'LET'	'and' : 'AND',	'sum' : 'SUM',
'input' : 'INPUT'	'or' : 'OR',	'min' : 'MIN',
'print' : 'PRINT'	'int' : 'INT',	'max' : 'MAX',
'if' : 'IF'	'float' : 'FLOAT',	'histogram' : 'HIST',
'then' : 'THEN'	'string' : 'STRING',	'violin' : 'VIOLIN',
'else' : 'ELSE'	'void' : 'VOID',	'box' : 'BOXPLOT',
'do' : 'DO'	'size' : 'SIZE',	'bar' : 'BAR',
'while' : 'WHILE'	'mean' : 'MEAN',	'scatter' : 'SCATTER',
'for' : 'FOR',	'median' : 'MEDIAN',	'mirror' : 'MIRROR'
'module' : 'MODULE',	'mode' : 'MODE',	
'return' : 'RETURN',	'variance' : 'VARIANCE',	

Tokens sencillos

EXPONENT	= r'\^'	LE	= r'\<='	SEMICOLON	= r'\,'
ADD	= r'\+\'	LT	= r'\<'	COMA	= r'\,'
DECREASE	= r'\--'	GE	= r'\>='	LPAREN	= r'\('
PLUS	= r'\+\'	GT	= r'\>'	RPAREN	= r'\)'
MINUS	= r'\--'	EQ	= r'\=\'	L_SBRKT	= r'\['
MULTIPLY	= r'*\'	NE	= r'\!=\'	R_SBRKT	= r'\]'
DIVIDE	= r'\/'	TYPE_ASSIGN	= r'\:\'		
ASSIGN	= r'\<-'	COLON	= r'\:\'		

Patrones de construcción:

IDs:	'[a-zA-Z_]w*'	Constantes string:	'"([^\\"n]+ \\.)*"'
Constantes float:	'\d+\.(?d*)?(e(\+ -)?d+)?'	Ingorar espacio en blanco	'\t\n\r\f'
Constantes enteras:	'\d+'		

Análisis de Sintaxis

A continuación se presentará la gramática formal del lenguaje Alice:

program : BEGIN ID : ENDPROG

global : module global
 | declaration global
 | main

main : MAIN stmtblock

stmtblock : : initstmt END

initstmt : stmt stmtchain
 | ϵ

stmtchain : stmtchain stmt
 | ϵ

stmt : assignment
 | conditional
 | print
 | input
 | iteration
 | plot
 | declaration
 | mirror
 | expression
 | return

conditional : IF expr THEN stmtblock
 | IF expr THEN : initstmt ELSE stmtblock

iteration : while
 | do_while
 | for

while : WHILE expr stmtblock

do_while : DO stmtblock WHILE expression

for : FOR ID ASSIGN expr : expr stmtblock

module : MODULE ID :: (params) stmtblock

mdl_type : VOID
| type

type : INT
| FLOAT
| STRING

params : ID :: type rparams
| ϵ

rparams : rparams , ID :: type
| ϵ

return : RETURN expression

call : ID (funparam)

assignment : variable <- expression

declaration : LET ID others :: type idxsize ;

others : others , ID
| ϵ

idxsize : [CTE_I matrix]
| ϵ

matrix : , CTE_I
| ϵ

expression : expr ;

expr : andexpr
| expr OR andexpr

andexpr : eqlexpr
| andexpr AND eqlexpr

eqlexpr : relexpr
| eqlexpr eqop relexpr

eqop : ==
| !=

relexpr : sumexpr
| relexpr relop sumexpr

relop : <=
| <
| >=
| >

sumexpr : term
| sumexpr sumop term

sumop : +
| -

term : unary
| term mulop unary

mulop : ^
| *
| /

unary : postfix
| - postfix

postfix : factor
| factor postop

postop : ++
| --

factor : (expr)
| value
| variable
| systemdef
| call

value : CTE_I
| CTE_F
| CTE_STRING

variable : ID
 | ID [expr]
 | ID [expr , expr]

print : PRINT (funparam) ;

funparam : expr auxparams
 | ϵ

auxparams : auxparams , expr
 | ϵ

input : INPUT (expr , ID) ;

mirror : MIRROR (expr , ID) ;

plot : x_plot
 | xy_plot

x_plot : HIST (ID , CTE_STRING) ;
 | VIOLIN (ID , CTE_STRING) ;
 | BOXPLOT (ID , CTE_STRING) ;

xy_plot : BAR (ID , ID , CTE_STRING) ;
 | SCATTER (ID , ID , CTE_STRING) ;

systemdef : SIZE (ID)
 | MEAN (ID)
 | MEDIAN (ID)
 | MODE (ID)
 | VARIANCE (ID)
 | STD (ID)
 | RANGE (ID)
 | SUM (ID)
 | MIN (ID)
 | MAX (ID)

Generación de Código Intermedio y Análisis Semántico

Direcciones Virtuales

Rango	Tipo
1000:2999	Enteros globales
3000:4999	Flotantes globales
5000:5999	Strings globales
6000:7999	Enteros locales
8000:9999	Flotantes locales
10000:10999	Strings locales
11000:15999	Enteros temporales
16000:20999	Flotantes temporales
21000:25999	Strings temporales
26000:27999	Constantes enteros
28000:29999	Constantes floatantes
30000:30999	Constantes strings
31000:31999	Pointers

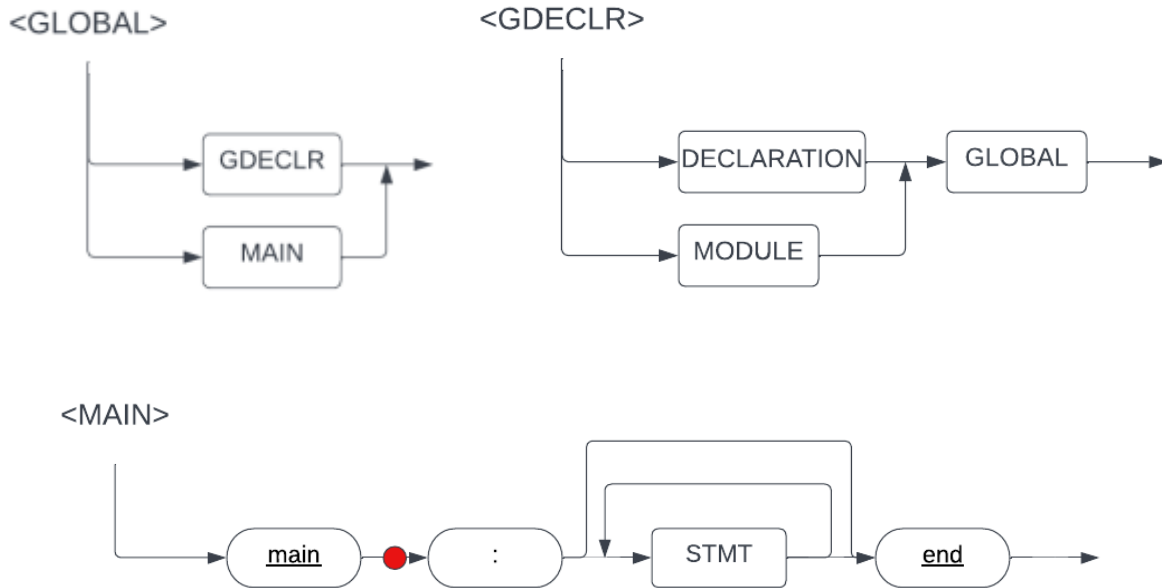
Diagramas de Sintáxis

<PROGRAM>



Puntos neurálgicos:

1. Agregar el primer cuádruplo a la lista de cuádruplos (Goto Main).
2. Agregar en el directorio de funciones un renglón con el ID del programa y la tabla de variables.
3. Agregar el último cuádruplo a la lista de cuádruplos (EndProgram), indicando el fin de la compilación.



Puntos neurálgicos:

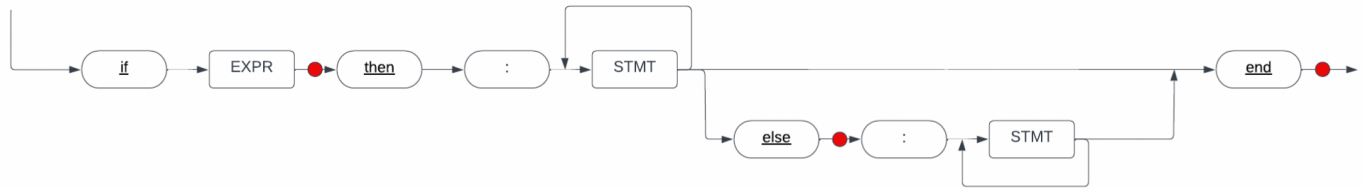
1. Modificar el cuádruplo de (Goto Main) para incluir el número de cuádruplo donde empieza la ejecución del main.



Puntos neurálgicos:

1. Realiza un pop a las pilas de operandos y tipos después de realizar una expresión aritmética (como 2+2) para no acumular información innecesaria en las pilas.

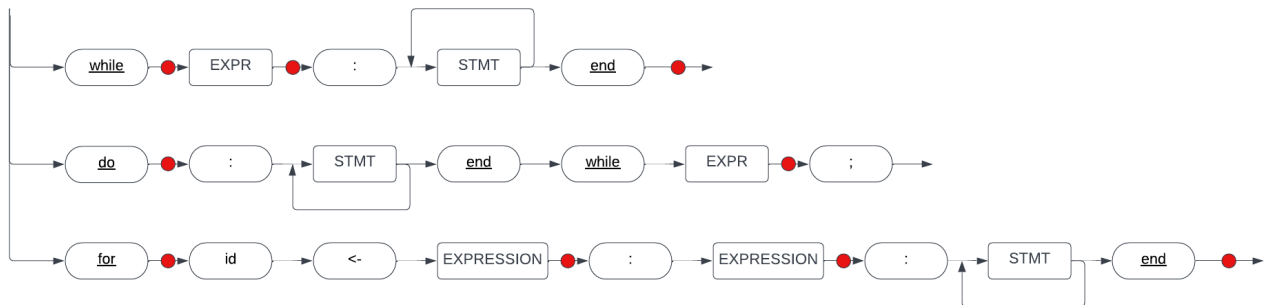
<CONDITIONAL>



Puntos neurálgicos:

1. Evalúa si la expresión que si genera un tipo booleano, en caso de que no, termina la compilación lanzando el error 36, en caso de que sí crea un cuádruplo GotoF que depende del resultado, y agrega la dirección del cuádruplo actual a la pila de saltos.
2. Saca el último valor de la tabla de saltos y le agrega al GotoF generado la dirección al siguiente cuádruplo.
3. En caso de haber un else, genera un cuádruplo Goto, le agrega al cuádruplo GotoF generado la dirección del cuádruplo Goto, y le agrega a la tabla de saltos el cuádruplo actual.

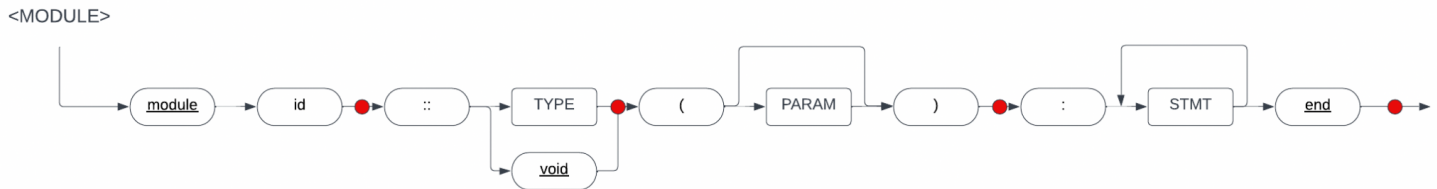
<ITERATION>



Puntos neurálgicos:

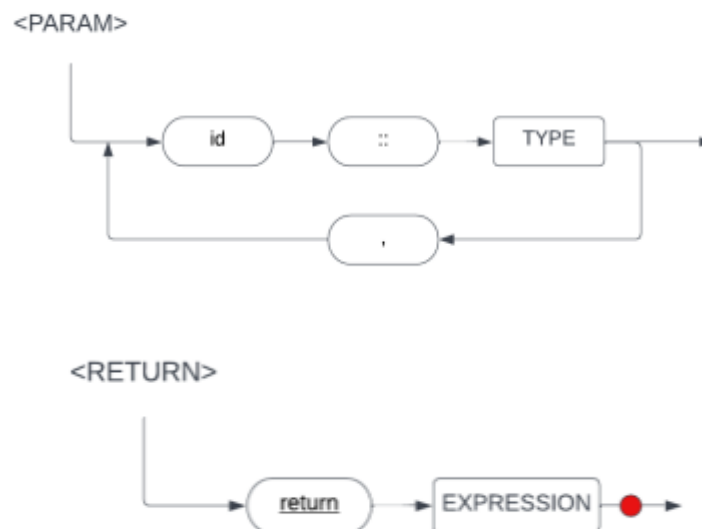
1. En while: Agrega a la tabla de saltos la dirección del cuádruplo actual.
2. En while: Evalúa si la expresión que si genera un tipo booleano, en caso de que no, termina la compilación lanzando el error 36, en caso de que sí crea un cuádruplo GotoF que depende del resultado, y agrega la dirección del cuádruplo actual a la pila de saltos.
3. En while: Realiza dos pops a la pila de saltos, con el primer valor agrega la dirección de siguiente cuádruplo al GotoF generado, y con el segundo genera un Goto al inicio del while.
4. En do while: Agrega a la tabla de saltos la dirección del cuádruplo actual.
5. En do while: Evalúa si la expresión que si genera un tipo booleano, en caso de que no, termina la compilación lanzando el error 36, en caso de que sí crea un cuádruplo GotoT que depende del resultado con la dirección del último valor de la pila de saltos, haciéndole un pop.
6. En for: Evalúa que la variable que se use para el ciclo haya sido declarada y sea de tipo int, de lo contrario arroja los errores 1 o 34 respectivamente. En caso de proceder, agrega la variable a la pila de operadores y su tipo a la pila de tipos.
7. En for: Evalúa que la expresión que se le asignará a la variable sea de tipo entero, generando los errores 34 y 35 en caso de que no se cumpla esta condición. Al proceder, genera un cuádruplo de asignación de dicho valor a la variable en la pila de operadores.

8. En for: Evalúa que la expresión meta también sea de tipo entero, generando el error 34 de lo contrario. Al proceder genera 4 cuádruplos para guardar el valor meta y el control, para comparar ambos valores, y un GotoF dependiente de la condición anterior.
9. En for: Genera un cuádruplo para sumarle 1 a la variable control, otro cuádruplo para actualizar la variable control, otro para asignarle este último valor a la variable del ciclo, y un último Goto al inicio del ciclo, actualizando el GotoF con la dirección del cuádruplo siguiente.



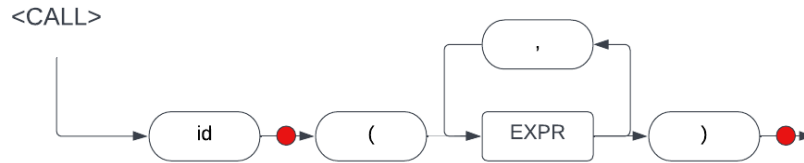
Puntos neurálgicos:

1. Evalúa si el nombre de la función ya fue utilizado para alguna variable u otra función, en caso de que si, genera los errores 25 y 26 respectivamente.
2. Si el tipo de la función no es void, genera una variable global homónima para guardar el valor de retorno, checando que haya espacio en memoria global, en caso de que no, arroja error 2. Finalmente, para toda función genera un nuevo registro en el directorio de funciones con el cuádruplo donde empezó y su tipo.
3. Itera sobre los parámetros, en caso de haber, creando nuevos registros en la tabla de variables de la función, y preparando el prototipado en el directorio de funciones.
4. Realiza el conteo de recursos de la función, resuelve llamadas recursivas (en caso de haber), resetea la memoria local y la tabla de variables, y genera un cuádruplo EndModule.



Puntos neurálgicos:

1. Si no se encuentra en una función tipada, genera el error 27. En caso de que la expresión de retorno no sea del mismo tipo que la función, genera el error 28. Finalmente, crea un cuádruplo de Return hacia la variable global homónima de la función.



Puntos neurálgicos:

1. Evalúa que la función llamada se encuentre dentro del directorio de funciones. Genera error 29 en caso de que no.
2. Evalúa los parámetros, generando los errores 30-33. Al proceder correctamente, crea el cuádruplo ARE con los recursos necesarios, los Parameter en caso de recibir alguno, el GoSub hacia la dirección de la función, y una asignación a temporal en caso de ser una función tipada.



Puntos neurálgicos:

1. Evalúa que ambos lados de la expresión sean del mismo tipo, generando un cuádruplo de asignación en caso de que si, o el error 35 en caso contrario.



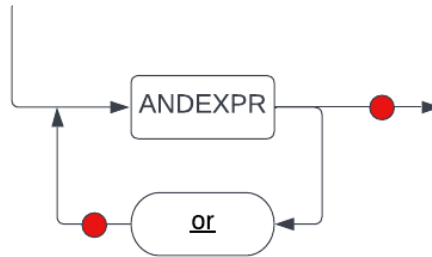
Puntos neurálgicos:

1. Evalúa que la función se trate de un arreglo, en caso de que no, prosigue al final del procesamiento del arreglo. Guarda también el tipo y el ambiente en donde se está declarando.
2. Evalúa la dimensión actual y verifica si existe una siguiente dimensión. En caso de que si procesa la dimensión actual, y regresa al mismo punto después de recibir ambas dimensiones.
3. Verifica cuántas dimensiones recibió, si fue arreglo o matriz, y basándose en eso determina el size de la variable.
4. Actualiza el contador de direcciones del tipo en base a la cantidad variables declaradas, y el tamaño si se trata de un arreglo o matriz.

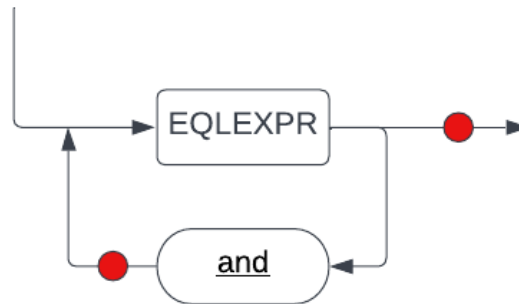
<EXPRESSION>



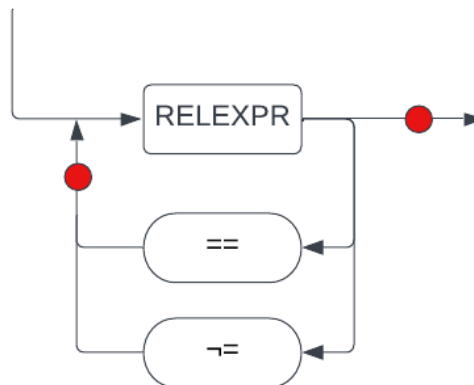
<EXPR>

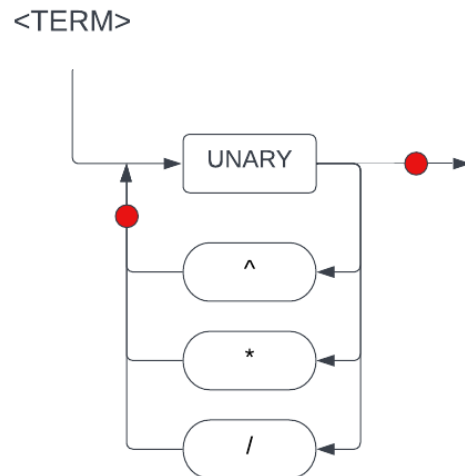
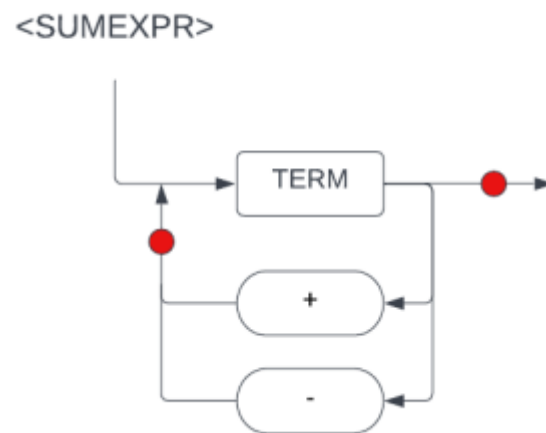
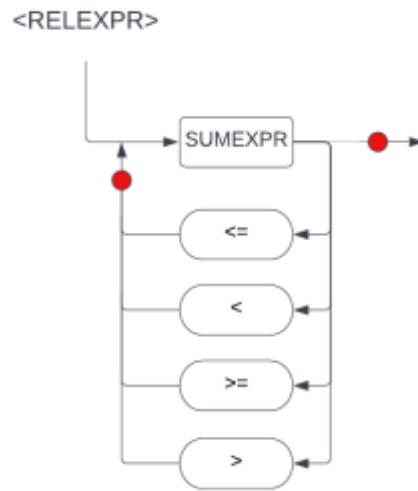


<ANDEXPR>



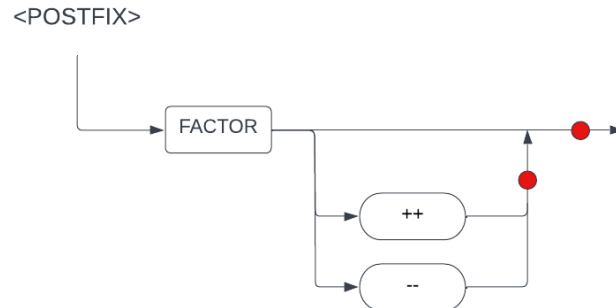
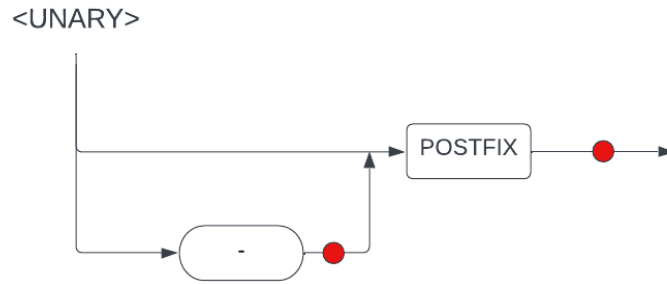
<EQLEXPR>





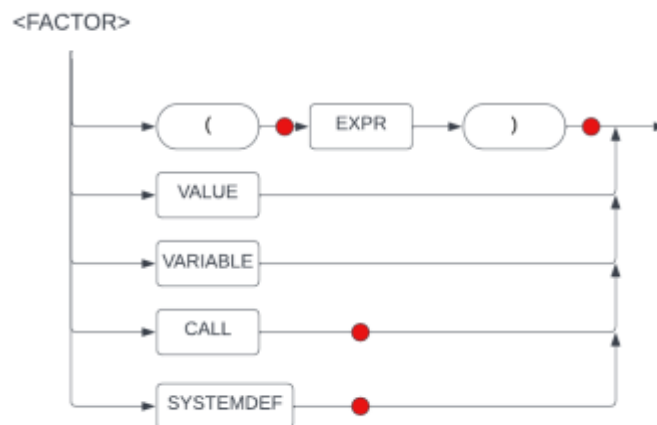
Puntos neurálgicos para toda expresión binaria:

1. Agrega el operador a la pila de operadores.
2. Evalúa si los operandos son de tipos que pueden operar entre sí y, en caso de serlo, genera el cuádruplo relevante en base al operador. De lo contrario, genera error 37, 38 o 39.



Puntos neurálgicos para toda expresión binaria:

1. Agrega el operador a la pila de operadores.
2. Evalúa si el operando es de tipos int o float, en caso de serlo, genera el cuádruplo relevante en base al operador. De lo contrario, genera error 40 o 41.



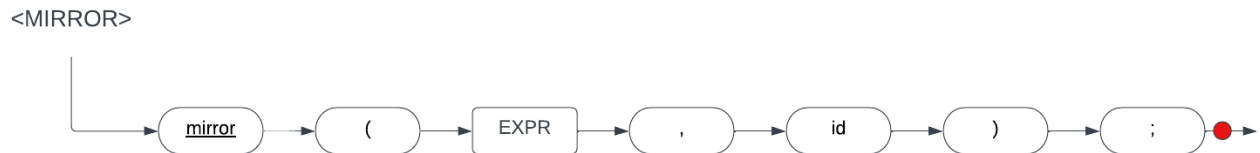
Puntos neurálgicos para toda expresión binaria:

1. En paréntesis: Agregar el paréntesis al tope de la pila de operadores.
2. En paréntesis: Remover el paréntesis al tope de la pila de operadores.
3. En call: Agregar a la pila de operadores la temporal atada al retorno de la función.
4. En systemdef: Evalúa la semántica de la variable recibida para comprobar si es un arreglo y genera un cuádruplo de la función estadística correspondiente.



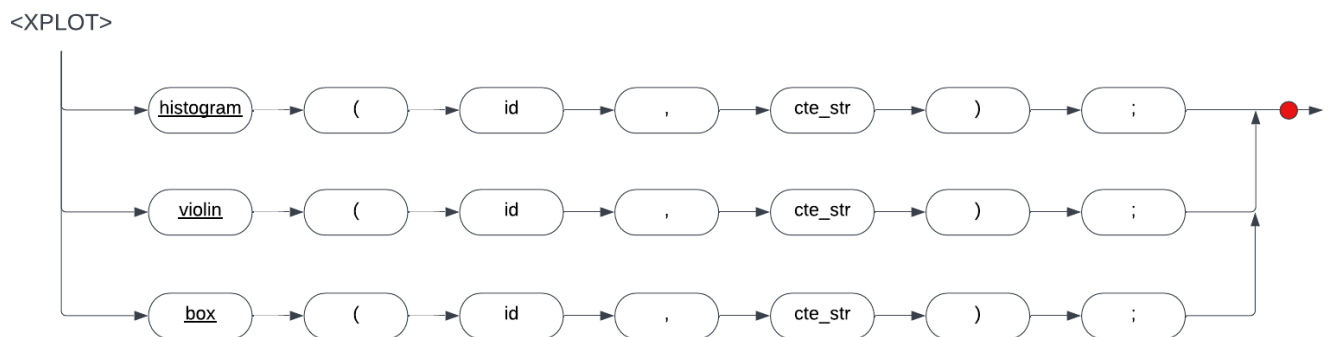
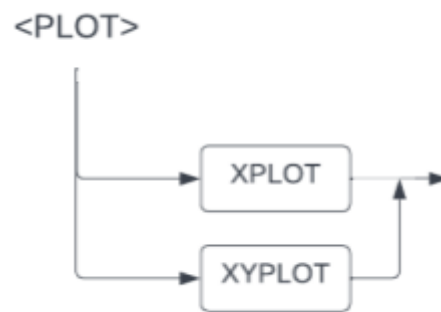
Puntos neurálgicos:

1. Evalúa que el mensaje sea de tipo string y que la variable exista, y genera un cuádruplo de input.



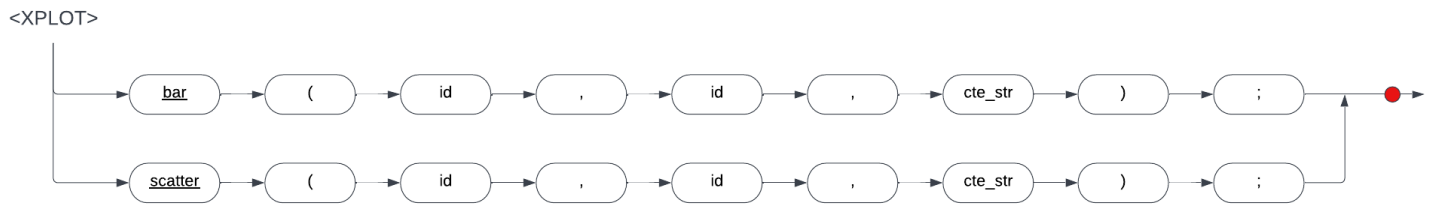
Puntos neurálgicos:

1. Evalúa que el nombre del archivo sea de tipo string y que la variable exista, y genera un cuádruplo de mirror.



Puntos neurálgicos:

1. Evalúa que la variable exista y verifica que el valor recibido sea una constante string, y genera un cuádruplo del tipo histograma, violin o boxplot.



Puntos neurálgicos:

1. Evalúa que las variables existan y verifica que el valor recibido al final sea una constante string, generando un cuádruplo del tipo scatter plot o bar plot.

Administración de Memoria en Compilación

Para administrar la memoria en la compilación se utilizó una tabla de variables, en donde se incluyó información pertinente para cada variable generada, como el ID de la variable, su tipo, su dirección virtual y su tamaño. Esto con el fin de poder realizar la verificación semántica, de comprobar que una variable existiera, checar el scope, el tamaño y dimensiones de la variable, así como su tipo.

Similarmente, se generó un directorio de funciones que almacena la información más importante de una función, como su tabla de variables, el prototipado de sus parámetros formales, su tipo y su ID, así como el cuádruplo donde empieza. La información anterior sirve para hacer los saltos de cuádruplo hacia el comienzo de una función, verificar expresiones donde se incluyen llamadas a funciones, así como verificar la existencia de una función y separar la memoria necesaria al momento de llamarla.

Los cuádruplos y la fila de cuádruplos se generaron para poder procesar las instrucciones recibidas en el código fuente y pasar la información a ejecución. Estas estructuras permiten la ejecución del código, así como la verificación semántica de ciertos tipos en la máquina virtual.

Para el manejo de operadores y los saltos en la generación de cuádruplos, se crearon las estructuras de pilas de saltos y de operadores. Así mismo se utilizaron la pila de dimensiones y la de tipos para comprobaciones semánticas de variables.

Tabla de consideraciones semánticas

Operadores	Tipos permitidos
++, --, ^, *, /, +, -	int x int, int x float, float x float
<, <=, >, >=	int x int, int x float, float x float
==, !=	int x int, int x float, float x float, string x string
and, or	bool x bool

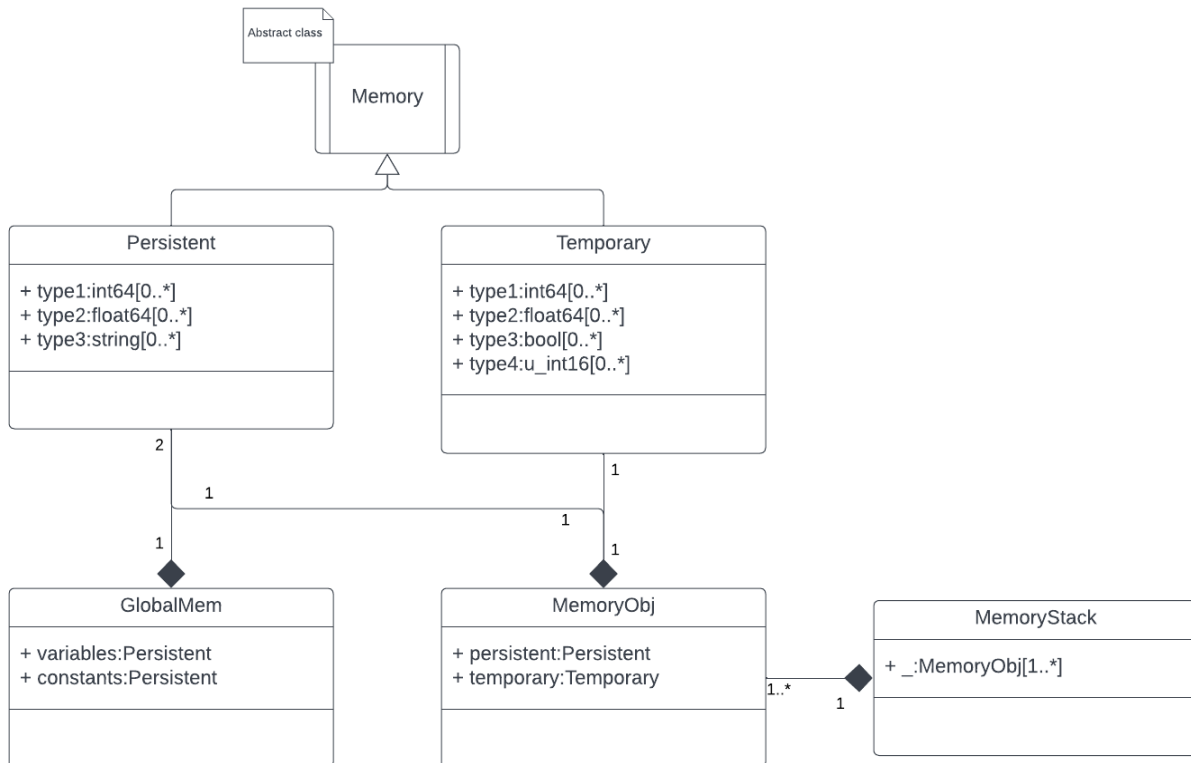
EJECUCIÓN DE ALICE

La ejecución de Alice se realizó utilizando 2 idiomas: Julia, que fue el lenguaje en el que se desarrolló la máquina virtual y las estructuras de datos de la memoria virtual, y Bash, que se utilizó para programar una especie de pseudo-repl o ejecutor que ata la compilación y la ejecución a una automáticamente, ya sea recibiendo los archivos a compilar como argumentos o a través de la interfaz repl. Por lo anterior dicho, para correr Alice, será necesario tener instaladas la última versión de Julia, así como correr el ejecutor en una terminal o ambiente que pueda procesar código escrito en Bash. Así mismo, se necesitará tener instalados los módulos StatsBase, PlottyJS y JSON para Julia para poder ejecutar el código recibido del compilador en la máquina virtual.

Administración de Memoria en Ejecución

La memoria de la máquina virtual de Alice fue creada utilizando varias clases que representan tanto la memoria global como la memoria local y temporal del lenguaje. Estas últimas se almacenan también dentro de un stack de memoria que funge como herramienta para procesar cambios de scopes hacia otras funciones y el main.

Las direcciones recibidas del compilador se traducen a través de un offset con las direcciones bases pertinentes a índices dentro de un vector vacío que funciona a manera de una *linked list* con un tipo específico que dependerá de la dirección recibida, así como el objeto de memoria que se utilizará para almacenar el valor recibido. A continuación se mostrará una representación gráfica de las estructuras anteriormente mencionadas:



PRUEBAS DEL FUNCIONAMIENTO

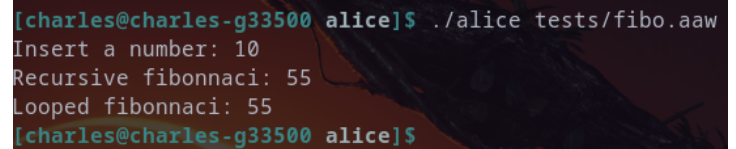
Fibonacci (recursivo e iterativo)

```
begin fibo:
  module loop_fibonnaci::int(n::int):
    let a,b,c,x::int;
    a <- 0;
    b <- 1;
    if n == 0 then:
      return a;
    else:
      if n == 1 then:
        return b;
      else:
        for x <- 2:(n+1):
          c <- a + b;
          a <- b;
          b <- c;
        end
        return b;
      end
    end
  end
end

module re_fibonacci::int(n::int):
  if n <= 1 then:
    return n;
  end
  return re_fibonacci(n-1) + re_fibonacci(n-2);
end

main:
  let x::int;
  input("Insert a number:", x);
  print("Recursive fibonnaci:", re_fibonacci(x));
  print("Looped fibonnaci:", loop_fibonnaci(x));
end
endprog
```

Resultados de la prueba:



```
[charles@charles-g33500 alice]$ ./alice tests/fibo.aaw
Insert a number: 10
Recursive fibonnaci: 55
Looped fibonnaci: 55
[charles@charles-g33500 alice]$
```

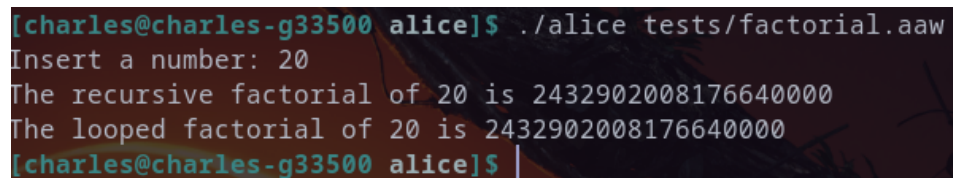
Factorial (recursivo e iterativo)

```
begin facto:
  module loop_factorial::int(n::int):
    let res::int;
    res <- 1;
    while n > 0:
      res <- res * n;
      n--;
    end
    return res;
  end

  module re_factorial::int(n::int):
    if n == 1 then:
      return n;
    end
    return n * re_factorial(n-1);
  end

  main:
    let x::int;
    input("Insert a number:", x);
    print("The recursive factorial of", x, "is", re_factorial(x));
    print("The looped factorial of", x, "is", loop_factorial(x));
  end
endprog
```

Resultados de la prueba:



```
[charles@charles-g33500 alice]$ ./alice tests/factorial.aaw
Insert a number: 20
The recursive factorial of 20 is 2432902008176640000
The looped factorial of 20 is 2432902008176640000
[charles@charles-g33500 alice]$ |
```

Funciones de data science

```
begin DS:
  let x,y::int[20];

  main:
    let i::int;

    mirror("tests/test.mrr", x);
    mirror("tests/test.mrr", y);

    print("Values inside the array:");
    for i <- 0:20:
      print(x[i]);
    end

    print();
    print("Description of array x:");
    range(x);
    print("Size:", size(x));
    print("Mean:", mean(x));
    print("Mean²:", mean(x)^2);
    print("Median:", median(x));
    print("Mode:", mode(x));
    print("Variance:", variance(x));
    print("Standard Deviation", std(x));
    print("Sum of contents:", sum(x));
    print("Smallest value:", min(x));
    print("Largest value:", max(x));
    print();

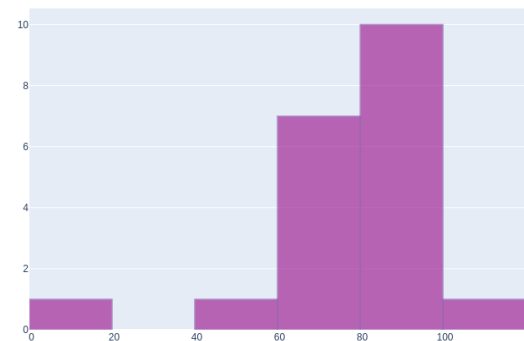
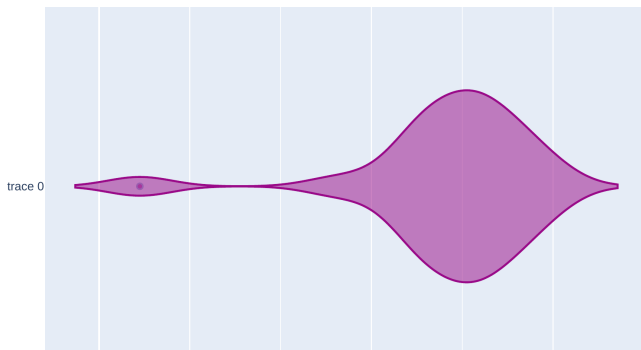
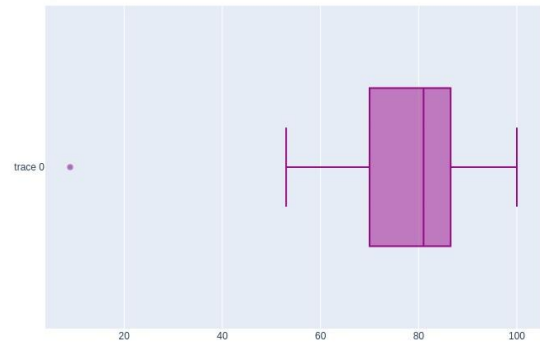
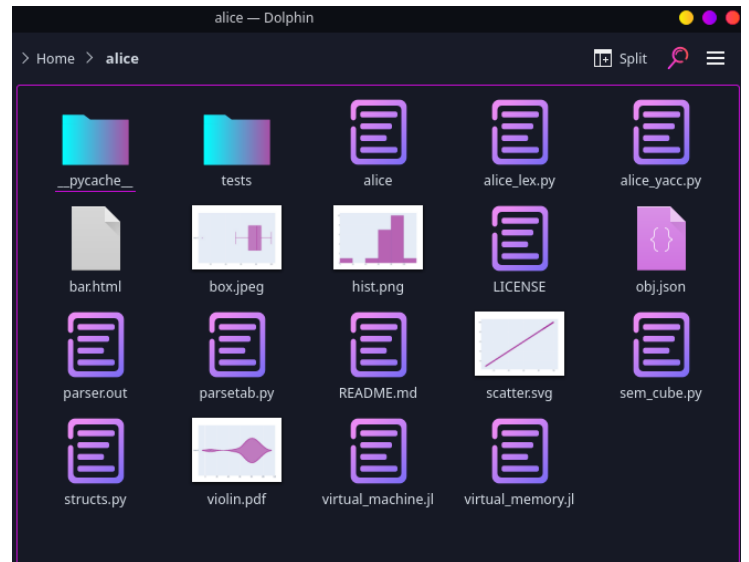
    print("---Graphing section---");
    histogram(x, "hist.png");
    box(x, "box.jpeg");
    violin(x, "violin.pdf");
    bar(x, y, "bar.html");
    scatter(x, y, "scatter.svg");
  end
endprog
```

Resultados de la prueba:

```
[charles@charles-g33500 alice]$ ./alice tests/ds.aaw
Values inside the array:
9
70
93
53
92
85
75
70
68
88
76
70
77
85
82
80
96
100
85

Description of array x:
Range: 9 - 100
Size: 20
Mean: 76.8
Mean2: 5898.24
Median: 81.0
Mode: 70
Variance: 377.6421052631578
Standard Deviation 19.433015856092894
Sum of contents: 1536
Smallest value: 9
Largest value: 100

---Graphing section---
Generating histogram plot...
Plot created as hist.png.
Generating box plot...
Plot created as box.jpeg.
Generating violin plot...
Plot created as violin.pdf.
Generating bar plot...
Plot created as bar.html.
Generating scatter plot...
```



Find

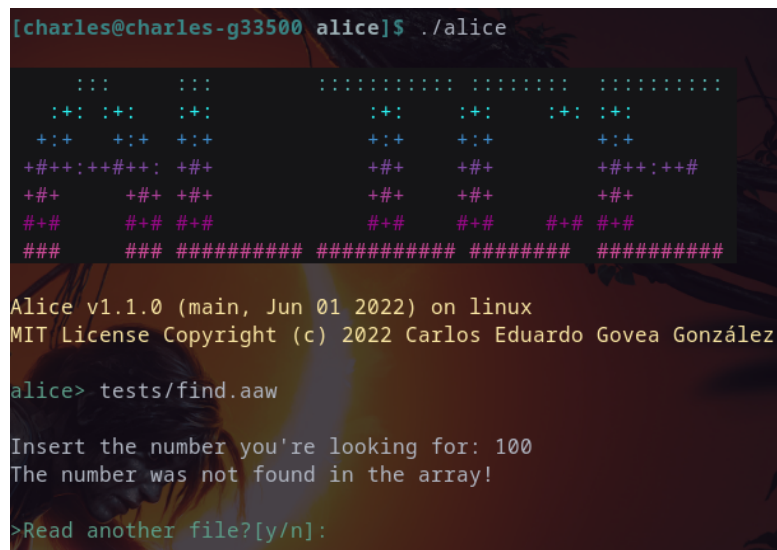
```
begin findtest:
  let arr::int[20];

  main:
    let res, y, num::int;
    res <- -1;
    mirror("tests/test.mrr", arr);
    input("Insert the number you're looking for:", num);

    for y <- 0:size(arr):
      if num == arr[y] then:
        res <- y;
      end
    end

    if res == -1 then:
      print("The number was not found in the array!");
    else:
      print("The number was found at index:", res);
    end
  end
endprog
```

Resultados de la prueba:



```
[charles@charles-g33500 alice]$ ./alice

      :::      :::      :::::::::::::: :::::::::::::: ::::::::::::::
    :+: :+: :+:      :+:      :+:      :+:      :+:
  +:+ +:+ +:+      +:+      +:+      +:+      +:+
+#++:++#++: +#+      +#+      +#+      +#++:++##
+#+      +#+ +#+      +#+      +#+      +#+
#+#      +#+ +#+      +#+      +#+      +#+      +#+
###      ### ##### ##### ##### ##### #####

Alice v1.1.0 (main, Jun 01 2022) on linux
MIT License Copyright (c) 2022 Carlos Eduardo Govea González

alice> tests/find.aaw

Insert the number you're looking for: 100
The number was not found in the array!

>Read another file?[y/n]:
```

La prueba falló, no encuentra valores que si se encuentran dentro del arreglo