Compilador
Orientado a Objetos

Diseño de compiladores

Group 1

Equipo: Obj7

Lucía Cantú-Miller	A01194199	Elda G. Quiroga, M.C.
Fernando Carrillo	A01194204	Dr. Héctor Ceballos, PhD

Instituto Tecnológico y de Estudios Superiores de Monterrey

Monterrey, N. L. Miércoles 2 de junio de 2021.

INDICE

Descripción del Proyecto **2**
- Propósito y alcance del proyecto 2
- Análisis de Requerimientos y Principales Test Cases 2
- Descripción del proceso general seguido 3

Descripción del Lenguaje **6**
- Nombre 6
- Descripción genérica del lenguaje 7
- Listado de los posibles errores 7

Descripción del compilador **7**
- Equipo de cómputo, lenguaje y utilerías 7
- Descripción del Análisis de Léxico 7
- Descripción del Análisis de Sintaxis 8
- Descripción de Generación de Código Intermedio y Análisis Semántico 10
- Descripción del proceso de Administración de Memoria 26

Descripción de la máquina virtual **31**
- Descripción del proceso de Administración de Memoria 31

Pruebas de funcionamiento del lenguaje **36**

Documentación del código del lenguaje **45**
- Comentarios de Documentación e Implementación 45

Quick Reference Manual **49**

Video Demostración **53**

I. Descripción del Proyecto

Propósito y alcance del proyecto

Para este proyecto se nos dio una descripción de las características generales de un lenguaje básico a desarrollar que debe de soportar definición y manipulación de clases (estilo C++). El producto final a entregar es un compilador que pueda compilar de manera exitosa un lenguaje de programación orientado a objetos.

Análisis de Requerimientos y Principales Test Cases

Requerimientos

- Declaración de *n* variables globales y locales
- Declaración de *n* objetos
- Declaración de *n* funciones
- Existen variables de tipo int, float, char, arreglos de 1 o 2 dimensiones y objetos creados por el usuario
- Manejo de asignación de variables
- El tipo de retorno de las funciones deben de ser entero, flotante, char o void
- Se puede leer uno o más identificadores (o a una casilla o a un atributo) separados por comas.
- Se pueden escribir letreros y/ó resultados de expresiones separadas por comas.
- Se pueden utilizar estatutos de decisión. Dichos estatutos pueden o no contener un “sino”.
- Se pueden utilizar estatutos de repetición condicionales.
- Se pueden utilizar estatutos de repetición no condicionales.
- Existen operaciones aritméticas, lógicas y relacionales
- Existen identificadores, palabras reservadas, constantes enteras, constantes flotantes, constantes char y constantes string (letreros).

Test Cases

- Creación y manejo de objetos
- Creación de todo tipo de variables
- Creación y uso de funciones
- Asignación de variables
- Lectura correcta de un identificador
- Lectura correcta de varios identificadores
- Escritura de letreros
- Escritura de varios resultados
- Uso de estatutos de repetición condicionales
- Uso de estatutos de repetición no condicionales
- Implementación de operaciones aritméticas
- Implementación de operaciones lógicas
- Implementación de operaciones relacionales
- Uso de estatuto de decisión utilizando el “sino”
- Uso de estatuto de decisión sin utilizar el “sino”

- Implementación de llamadas recursivas
- Implementación de llamada a funciones “void”
- Implementación de llamada a funciones de tipo retorno
- Manejo de arreglos de 1 dimensión
- Manejo de arreglos de 2 dimensiones

Descripción del proceso general seguido

Para el desarrollo de este proyecto se estuvieron haciendo entregas de los avances cada semana. Este es el esquema de entregas que se siguió.

Semana	L	Mi	J	AVANCE	Contenido esperado de la entrega
Mzo-Abril (29 - 2)	29	**31**	2	--	**SEMANA SANTA**
Abril (5 - 9)	5	**7**	9	**#1**	Análisis de Léxico y Sintaxis *(**solo entregan quienes van a seguir los proyectos pre-diseñados o aquéllos que cuenten ya con propuesta de proyecto APROBADA **)*
Abril (12 - 16)	12	**14**	16	**#2**	Semántica Básica de Variables: Directorio de Procedimientos y Tablas de Variables
Abril (19- 23)	19	**21**	23	**#3**	Semántica Básica de Expresiones: Tabla de Consideraciones semánticas (Cubo Semántico) Generacion de Código de Expresiones Aritméticas y estatutos secuenciales: Asignación, Lectura, etc.
Abril (26 - 30)	26	**28**	30	**#4**	Generacion de Código de Estatutos Condicionales: Decisiones/Ciclos
Mayo (3 – 7)	3	**5**	7	**#5**	Generacion de Código de Funciones
Mayo (10 – 14)	10	**12**	14	**#6**	Mapa de Memoria de Ejecución para la Máquina Virtual Máquina Virtual: Ejecución de Expresiones Aritméticas y Estatutos Secuenciales
Mayo (17 – 21)	17	**19**	21	**#7**	Generacion de Código de Arreglos /Tipos estructurados Máquina Virtual: Ejecución de Estatutos Condicionales
Mayo (24 – 28)	24	**26**	28	**#8**	1era versión de la Documentación Generacion de Código y Máquina Virtual para una parte de la aplicación particular
Junio (31– 4)	31	**2**	4	**FINAL**	***ENTREGA FINAL DEL PROYECTO JUNIO 2, 12:00pm***

Diseño de Compiladores--PROGRAMACIÓN DE AVANCES. - Semestre Febrero-Junio 2021

# AVANCE	FECHA	AVANCE ENTREGADO
1	Abril 9	Avances: - Implementación de tokens y gramática descritos en el documento con los diagramas Pendiente: - Investigar y arreglar warnings de shift/reduce reduce/reduce Cambios personales a la descripción original - En lugar de llamar los tipos en español 'entero, flotante', se cambiaron a inglés 'int, float'

		- En lugar de declarar las variables como 'ids : tipo', se cambió el orden a 'tipo : ids'
2	Abril 16	Avances: - Implementación de Directorio de funciones y Tablas de variables. Las funciones son agregadas junto con su tipo al directorio, y las variables son agregadas a las tablas de variables en su respectivo scope junto con su tipo. Pendiente: - Investigar y arreglar warnings de shift/reduce reduce/reduce - Incluir atributo de valores en las tablas de variables, y agregar espacios de memoria para variables de 1 o 2 dimensiones (tipo arreglos)
3	Abril 23	Avances: - Implementación de Cubo Semántico y generación de cuádruplos para operaciones aritméticas con variables (sin tipo) y valores cte. Pendiente: - Investigar y arreglar warnings de shift/reduce reduce/reduce - Incluir atributo de valores en las tablas de variables, y agregar espacios de memoria para variables de 1 o 2 dimensiones (tipo arreglos)
4	Mayo 1	Avances: - generación de código para estatutos condicionales (If/else, while, for) Pendiente: - Investigar y arreglar warnings de shift/reduce reduce/reduce
5	Mayo 8	Avances: - Generación de código para declaración de funciones y llamadas a funciones. Falta hacer la creación de memoria ERA size. Pendiente: - Investigar y arreglar warnings de shift/reduce reduce/reduce - Arreglar detalle de asignación a chars se toman como ints.
6	Mayo 16	Avances: - Se agregó el main a directorio de variables con sus variables globales y se agregó la cantidad de variables temporales utilizadas por función - Se agregó funcionalidad de mapa de memoria que asigna direcciones a variables globales y locales - Se agregó funcionalidad de mapa de memoria que asigna direcciones al resto de las variables (constantes, temporales). Los cuádruplos ahora utilizan las direcciones virtuales. Se genera un archivo de código intermedio con DirFunc, Tabla Constantes,

		Cuádruplos con direcciones virtuales. Pendiente: - Investigar y arreglar warnings de shift/reduce reduce/reduce - Arreglar detalle de asignación a chars se toman como ints
7	Mayo 25	Avances: - Se agregó el administrador de memoria de la máquina virtual y función que parsea y guarda elementos del código intermedio generado por el compilador - Se agregaron instrucciones para manejar las operaciones de los cuádruplos de aritmética, impresión y lectura - Se agregaron instrucciones de condicionales y estatutos no lineales. también se agregó ejecución de funciones simples - Se agregó funcionalidad de parámetros. - Se ajustó el manejo de memoria de la máquina virtual para manejar recursión. Pendiente: - Investigar y arreglar warnings de shift/reduce reduce/reduce
Final	Junio 2	Avances: - Se agregó la declaración y el manejo de objetos. - Se arreglaron bugs. - Se finalizó el proyecto. - Se finalizó la documentación. - Se corrigió error de chars tomados como ints

Bitacoras				
Bitacora 1	Bitacora 2	Bitacora 3	Bitacora 4	Bitacora 5
Bitacora 6	Bitacora 7			

Proceso Seguido

Para el desarrollo de este proyecto ambos integrantes del equipo trabajamos juntos para definir la lista de trabajos que se deberían de desarrollar en la semana. Posteriormente, nos conectamos por Discord o Whatsapp para comentar las tareas a realizar y definir cómo lo íbamos a desarrollar. Era elemento clave que los dos estuviéramos en la misma página de cómo iban a funcionar las cosas en nuestro compilador para que todo lo que trabajáramos se complementara. Para el desarrollo del código hacíamos pair programming donde cada quien trabajaba en la sección del código que le correspondía y luego juntábamos los trabajos. También, en las ocasiones donde se nos complicaba un poco más el entender cómo íbamos a abordar un problema, lo que hacíamos era conectarnos a discord y a un liveshare para trabajar en el problema de manera simultánea.

Reflexiones

Fernando: Aunque fue un proyecto retador, disfruté trabajar en él ya que me permitió entender mucho mejor cómo es que funcionan los lenguajes de programación. Por otra parte siento que este proyecto logró poner en práctica una gran parte de mis conocimientos obtenidos de diferentes materias a lo largo de mi carrera. Este proyecto nos demandó pensar críticamente en un buen diseño, así como mantener buenas prácticas de programación de manera que pudiéramos agregar funcionalidades al programa a través de las semanas sin batallar ni pelear con el código previo. De igual manera este proyecto fomentó la comunicación, el trabajo en equipo y la administración de tiempo porque de otra forma sería muy complicado acabar el compilador. Me hubiera gustado tener más tiempo para implementar las funcionalidades más avanzadas del programa como arreglos y objetos, pero dado el calendario apretado tuvimos que avanzar lo que alcanzaramos a nuestra mejor habilidad.

Lucía: La planeación y desarrollo de este proyecto fue un gran reto que nos consumió mucho tiempo y energía. Sin embargo, fue un excelente método de aprendizaje puesto que nos seguido teníamos que tomarnos un tiempo para analizar a fondo lo que se tenía que lograr para saber cómo deberíamos de abordar el problema. En ocasiones nos era difícil encontrar un tiempo para trabajar en equipo, pero Fer y yo logramos hacer que las cosas funcionaran. Para lograr esto tuvimos que tener una comunicación constante y abierta para poder los dos estar en la misma página. A pesar de las dificultades este fue un proyecto que disfruté hacer y aparte, Fer fue un excelente compañero de trabajo.

II. Descripción del Lenguaje

Nombre

FerLu

Descripción genérica del lenguaje

El lenguaje de programación es similar al de C++. Este lenguaje implementa los conceptos definidos por la programación orientada a objetos. La intención es que se permita manipulación de objetos. También cuenta con facilidades de programación genérica que se centra más en los algoritmos que en los datos.

Listado de los posibles errores

- Un error de léxico o sintaxis
- Error de tipo de resultado después de una operación aritmética, relacional o lógica.
- Error al querer acceder a una función de un objeto que no existe.
- Error al querer acceder a un atributo de un objeto que no existe.
- Error al validar el tipo de variables al momento de la asignación.
- Resultado inválido al implementar una operación.
- La lectura de un identificador no es correcta.
- La impresión de un letrero no es correcta.
- La impresión de varios resultados no es correcta.
- No se validan los parámetros de una función correctamente.
- Inconsistencias en el valor de retorno de una función y en el valor que realmente regresa.
- Error al acceder a un espacio válido de un arreglo.
- Que te permita acceder a un espacio no válido de un arreglo.

III. Descripción del compilador

Equipo de cómputo, lenguaje y utilerías

- Mac OS
- Windows
- C++
- Flex y Bison

Descripción del Análisis de Léxico

TOKEN	REGEX
si	si
entonces	entonces
sino	sino
mientras	mientras
hacer	hacer
desde	desde
hasta	hasta
programa	programa
principal	principal
clase	clase
hereda	hereda
variables	variables
funcion	funcion
n_int	int
n_float	float
n_char	char
n_void	void
regresa	regresa

lee	lee
escribir	escribir
atributos	atributos
metodos	metodos
asignador	=
mas	+
menos	-
mult	*
divi	/
op_or	|
op_and	&
op_rel	> | < | <> | >= | <= | ==
l_paren	(
r_paren	)
l_brace	{
r_brace	}
l_bracket	[
r_bracket	]
dos_puntos	:
punto_coma	;
coma	,
punto	.
flecha	->
digito	[0-9]
char	[a-z | A-Z]
cte_int	-?digito+
cte_float	-?digito*(\. digito+)? (E[+-]? int)?
cte_char	\'char\'
comentario	%%.*
cte_string	\"[^"]*\"
id_	char(char | digito | _)*

Descripción del Análisis de Sintaxis

Gramática Formal

PROGRAMA	-->	programa id_ ; DECLARACIONES principal ( ) { ESTATUTOS }
DECLARACIONES	-->	DECLARACION DECLARACIONES | epsilon
DECLARACION	-->	CLASE | VARIABLES | FUNCIONES
CLASE	-->	clase id_ HEREDA ; { ATRIBUTOS METODOS } ;
HEREDA	-->	hereda id_ | epsilon
ATRIBUTOS	-->	atributos VARS | epsilon
METODOS	-->	metodos FUNCIONES | epsilon
VARIABLES	-->	variables VARS
VARS	-->	VAR VARS_
VARS_	-->	VARS | epsilon
VAR	-->	TIPO_VAR : VAR_IDS ;

VAR_IDS	-->	id_ ARR VAR_IDS_
ARR	-->	[cte_int ARR_] | epsilon
ARR_	-->	, cte_int| epsilon
VAR_IDS_	-->	, VAR_IDS | epsilon
TIPO_SIMPLE	-->	n_int | n_float | n_char
TIPO_VAR	-->	TIPO_SIMPLE| id_
FUNCIONES	-->	FUNCION FUNCIONES_
FUNCIONES_	-->	FUNCIONES | epsilon
FUNCION	-->	funcion TIPO_RET id_ (PARAMETROS) ; FUNC_VARIABLES { ESTATUTOS }
FUNC_VARIABLES	-->	VARIABLES | epsilon
TIPO_RET	-->	TIPO_SIMPLE | n_void
PARAMETROS	-->	TIPO_SIMPLE : id_ PARAMETROS_ | epsilon
PARAMETROS_	-->	, PARAMETROS | epsilon
ASIGNACION	-->	id_ ASIG asignador EXP ;
ASIG	-->	. id_ | [EXP ASIG_] | epsilon
ASIG_	-->	, EXP | epsilon
LLAMADA_VOID	-->	id_ VOID_ATTR (FUNC_PARAM) ;
VOID_ATTR	→	-> id_ | epsilon
FUNC_PARAM	-->	EXP FUNC_PARAM_ | epsilon
FUNC_PARAM_	-->	, FUNC_PARAM | epsilon
DECISION	-->	si (EXP) entonces { ESTATUTOS } DECISION_
DECISION_	-->	sino { ESTATUTOS } | epsilon
CONDICIONAL	-->	mientras (EXP) hacer { ESTATUTOS }
NOCONDICIONAL	-->	desde id_ asignador EXP hasta EXP hacer { ESTATUTOS }
ESCRITURA	-->	escribe (PRINTABLE) ;
PRINTABLE	-->	E PRINTABLE_
PRINTABLE_	-->	, PRINTABLE | epsilon
E	-->	cte_string | EXP
LECTURA	-->	lee (READABLE) ;
READABLE	-->	id_ READABLE_
READABLE_	-->	, READABLE | epsilon
RETORNO	-->	regresa (EXP) ;
EXP	-->	EXP_Q
EXP_Q	-->	T_EXP EXP_
EXP_	-->	op_or EXP | epsilon
T_EXP	-->	G_EXP T_EXP_
T_EXP_	-->	op_and T_EXP | epsilon
G_EXP	-->	M_EXP G_EXP_
G_EXP_	-->	OPER_REL M_EXP | epsilon
OPER_REL	-->	op_rel
M_EXP	-->	T M_EXP_
M_EXP_	-->	+ M_EXP | - M_EXP | epsilon
T	-->	F T_

T_	-->	* T | / T | epsilon
F	-->	(EXP_Q) | CTE | id_ ID_A
ID_A	-->	ID_ATTR | ID_METHOD | [EXP ID_ARR] | epsilon
ID_ARR	-->	, EXP | epsilon
ID_ATTR	-->	. id_ | epsilon
ID_METHOD	-->	FLECHA LLAMADA_RET
FLECHA	-->	-> id_ | epsilon
LLAMADA_RET	-->	(FUNC_PARAM) | epsilon
CTE	-->	cte_int | cte_float | cte_char
ESTATUTOS	-->	ESTATUTO ESTATUTOS | epsilon
ESTATUTO	-->	ASIGNACION | LLAMADA_VOID | LECTURA | ESCRITURA | DECISION | CONDICIONAL | NOCONDICIONAL | RETORNO

Descripción de Generación de Código Intermedio y Análisis Semántico

El código intermedio está compuesto de **3** partes divididas por '**%%**':

La primera parte consta de las distintas funciones junto con la cantidad de variables que necesitan reservar a la hora de llamar al 'ERA'. El formato para cada función es:

1. Nombre de la función,
2. Dirección de cuádruplo inicial,
3. Núm. variables locales int,
4. Núm. variables locales float,
5. Núm. variables locales char,
6. Núm. variables temporales locales int,
7. Núm. variables temporales locales float,
8. Núm. variables temporales locales bool

Ej. dotProduct, 1, 4, 0, 0, 20, 0, 4

La segunda parte del archivo representa la Tabla de Constantes, donde cada línea indica una constante y su dirección de memoria.

Cte., dirección de memoria
Ej. 0, 30000
1, 30001
3.14, 34000

La tercera parte está compuesta por todos los cuadrupulos que se generaron al compilar el código fuente. En esta parte las operaciones son sustituidas por su valor numérico, y los operandos son sustituidos por sus direcciones virtuales. Cada línea contiene 4 elementos estos siendo:

Operación, operando1, operando2, resultado.
Ej. 1, 17000, 34002, 25002

Ejemplo de archivo codigo.cmm

```
matrixMult, 55, 5, 0, 0, 25, 0, 5
dotProduct, 1, 4, 0, 0, 20, 0, 4
multMat, 0, 30, 0, 0, 5, 0, 2
%%
18, 30005
9, 30004
1, 30001
2, 30003
3, 30002
0, 30000
%%
16, principal, -, 121
12, 30000, -, 15000
5, 15000, 27, 27000
14, 27000, -, 25
12, 30000, -, 15001
5, 15001, 27, 27001
14, 27001, -, 22
...
```

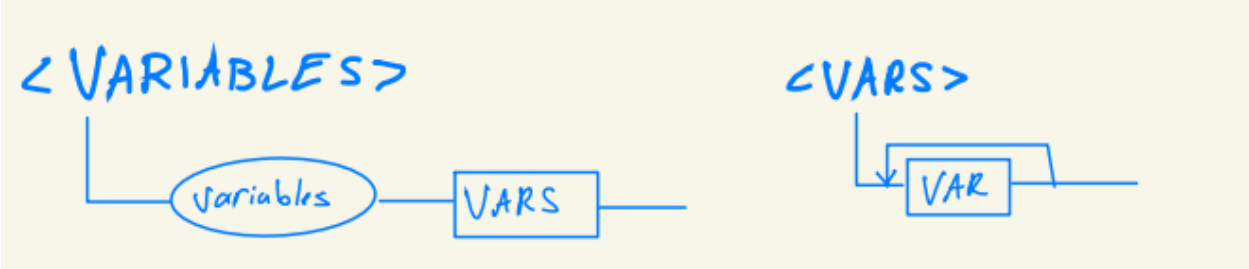
Mapa de operaciones con su código numérico

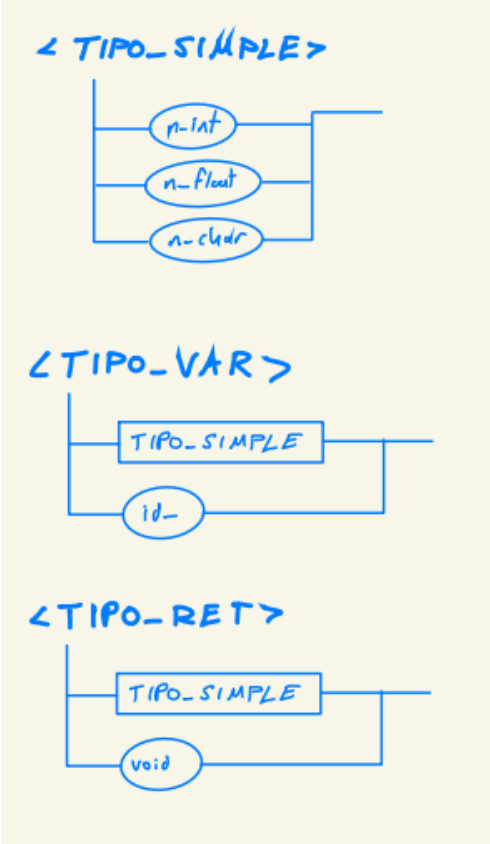
```
std::unordered_map<std::string, int> operatorMap {
        {"+", 0},
        {"-", 1},
        {"/", 2},
        {"*", 3},
        {">", 4},
        {"<", 5},
        {">=", 6},
        {"<=", 7},
        {"==", 8},
        {"<>", 9},
        {"&", 10},
        {"|", 11},
        {"=", 12},
        {"PARAMETER", 13},
        {"GotoF", 14},
        {"GOSUB", 15},
        {"GOTO", 16},
        {"ERA", 17},
```

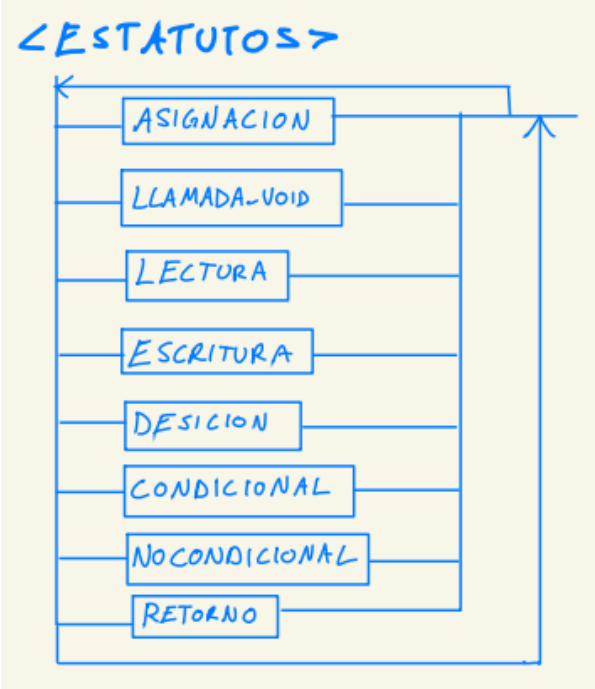
```
      {"ENDFUNC", 18},
      {"write", 19},
      {"read", 20},
      {"RETURN", 21},
      {"END", 22},
      {"VER", 23}
  };
```

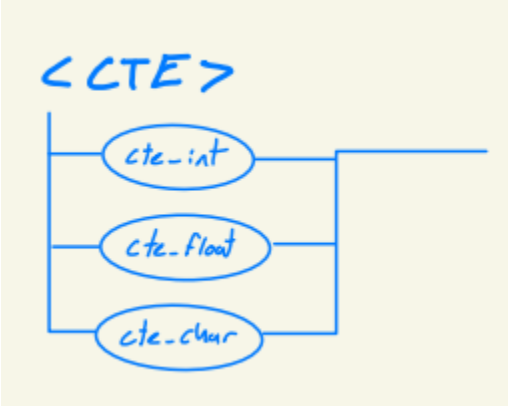
```
void Quad::generateIntermediateCode() {
    std::ofstream outFile;
    outFile.open("codigo.cmm");
    // func dir
    for (auto &entry : (*tablasDatos).funcDir) {
        FuncEntry &funcData = entry.second;
        outFile << entry.first << ", " << funcData.quadCont << ", ";
        // local vars used
        outFile << funcData.numLVar[0] << ", ";
        outFile << funcData.numLVar[1] << ", ";
        outFile << funcData.numLVar[2] << ", ";
        // temps used
        outFile << funcData.numTemp[0] << ", ";
        outFile << funcData.numTemp[1]<< ", ";
        outFile << funcData.numTemp[3] << "\n";
    }

    // const table
    outFile << "%%\n";
    for (auto entry : (*tablasDatos).constDir) {
        outFile << entry.first << ", " << entry.second << "\n";
    }

    // quads
    outFile << "%%\n";
    for (auto quad : memQuads) {
        outFile << quad[0] << ", " << quad[1] << ", " << quad[2] << ", "
<< quad[3] << "\n";
    }
    outFile.close();
}
```

Diagramas de Sintaxis:





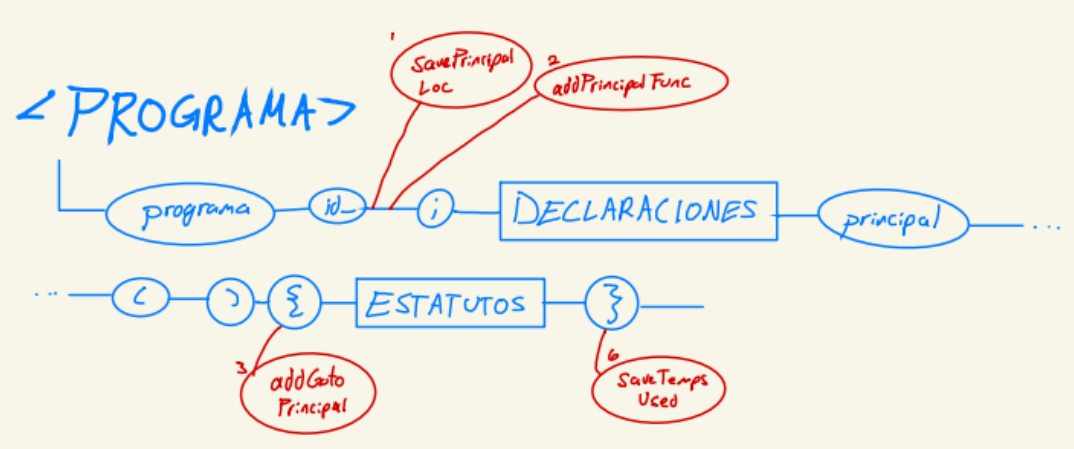




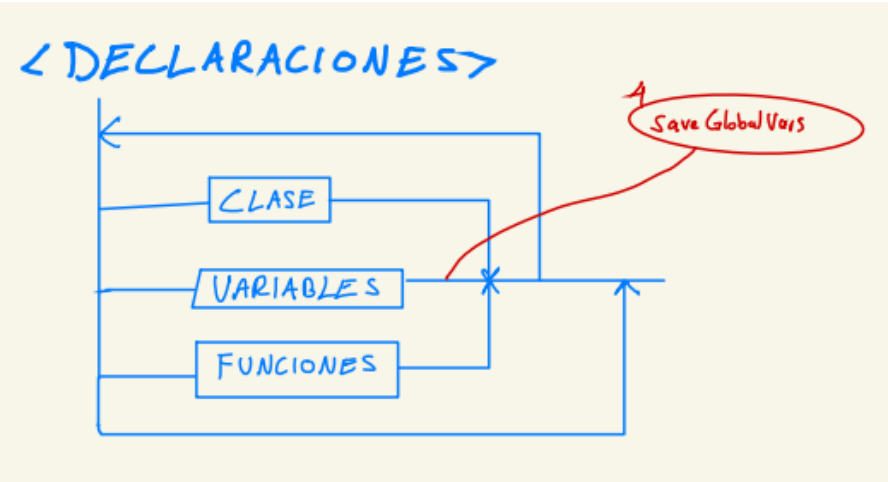
Diagramas completos:

https://drive.google.com/file/d/1UPVnM2Wg-w8ffcVbX5gvB63F-Z7RsihI/view?usp=sharing

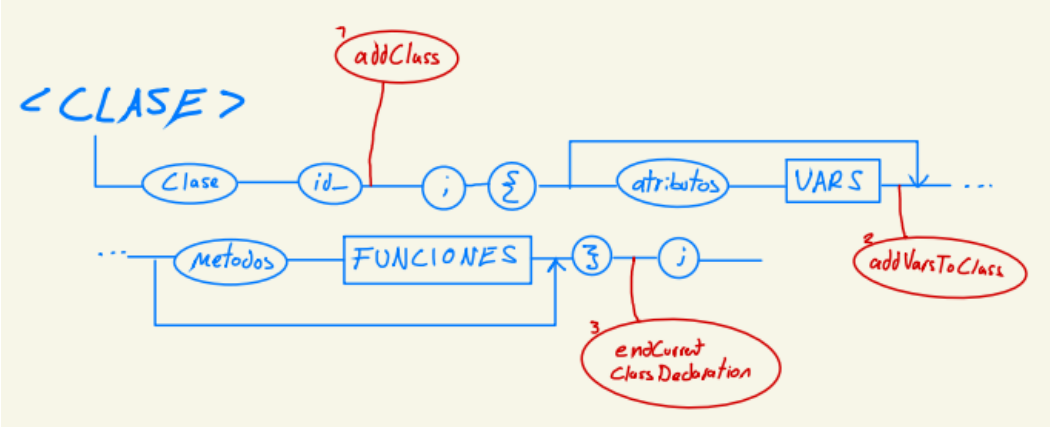
Acciones Semánticas y de Generación de Código



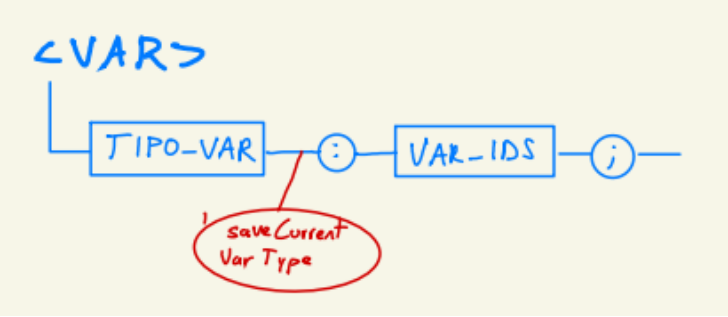
- PROGRAMA
    - **savePrincipalLoc**: Agrega al cuádruplo la instrucción para ir al programa principal y deja el salto pendiente.
    - **addPrincipalFunc**: Se agrega la función principal al directorio de funciones y se agrega la tabla de datos.
    - **addGotoPrincipalLoc**: Ya que se tiene dónde empieza el programa principal se regresa al cuádruplo para ir al programa principal y se especifica a donde se va el “GOTO”.
    - **saveTempsUsed**: Guarda la cantidad de temporales utilizadas de cada tipo por el main en el directorio de funciones.
    -



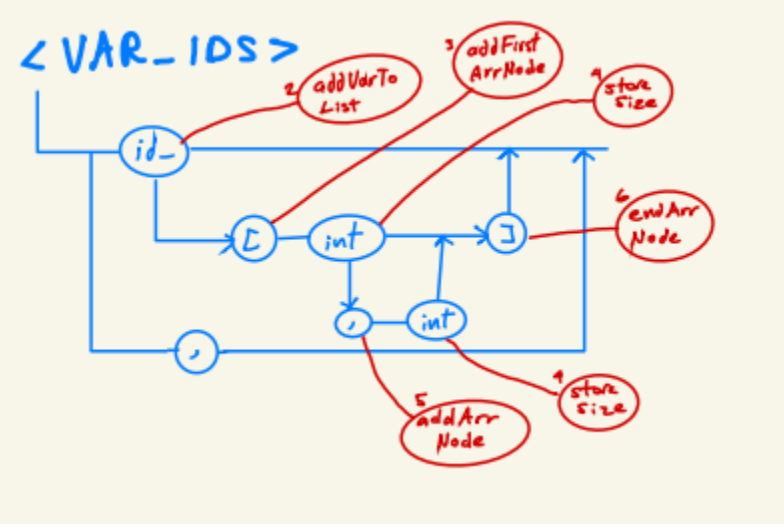
- DECLARACIONES
    - **saveGlobalVars**: Se guardan las variables globales declaradas en el directorio de funciones.



- CLASE
  - **addClass**: Verifica que no exista la clase y la agrega al directorio de clases
  - **addVarsToClass**: Verifica que no existan las variables a declarar y agrega los atributos al directorio de variables.
  - **endCurrentClassDeclaration**: regresa el valor de currentClassDecl a “main”.

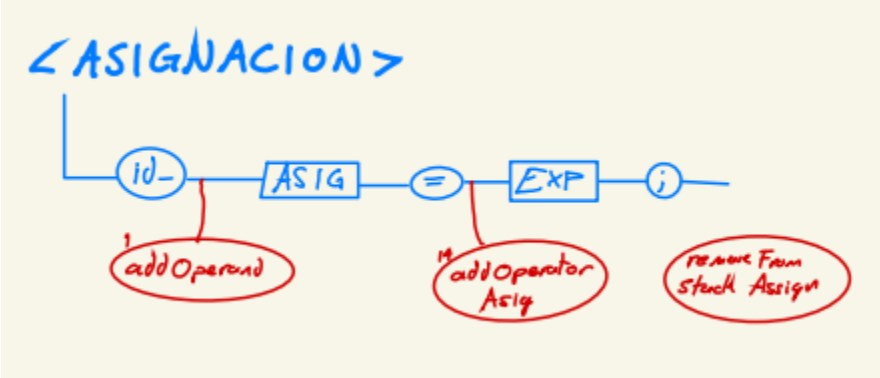


- VAR
  - **saveCurrentVarType**: guarda el tipo de las variables que se van a declarar en una misma línea.

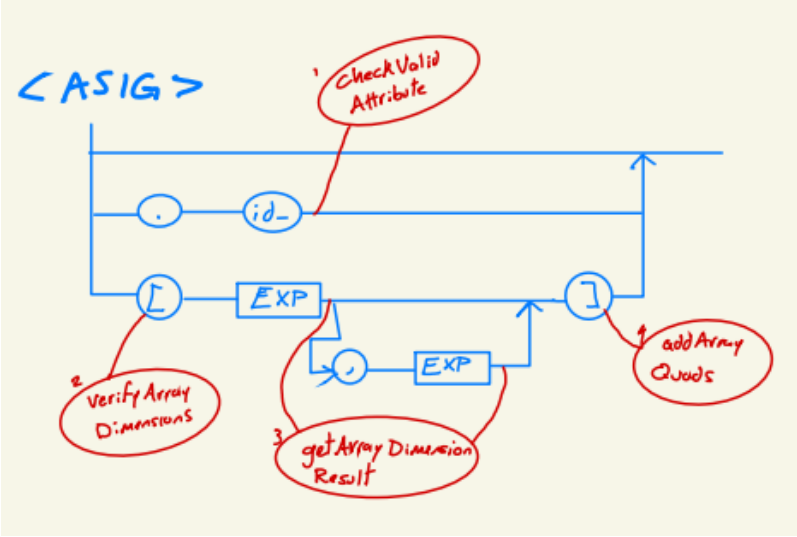


- VAR_IDS

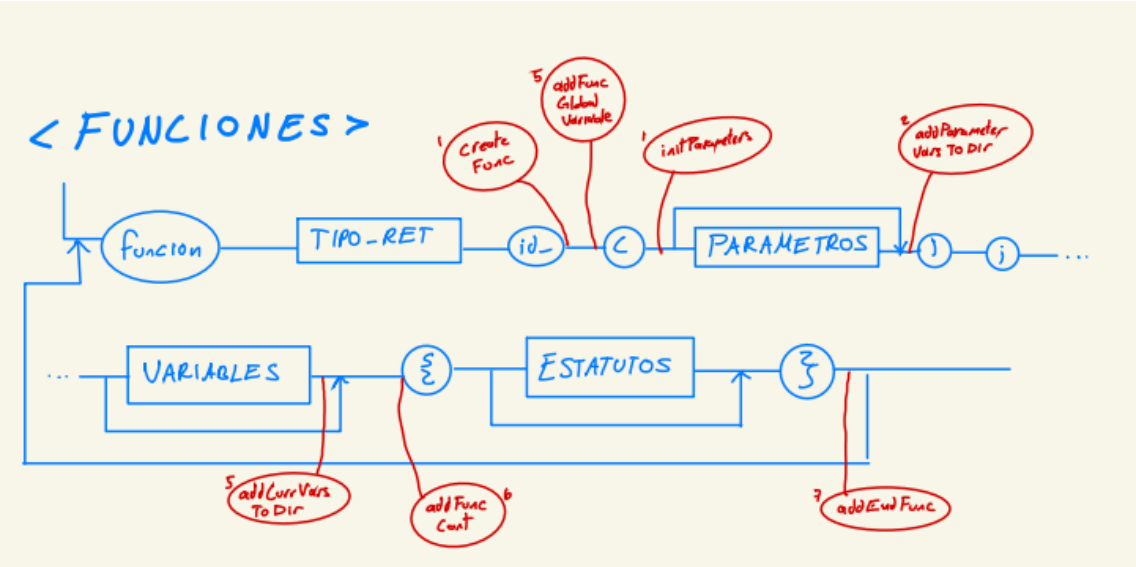
- **addVarToList**: checa que la variable no existe previamente y la guarda en una lista de variables que al terminar la línea serán agregadas a su tabla de variables respectiva.
- **addFirstArrNode**: crea un nodo donde se guardará la información de la primera dimensión del arreglo, y guarda R = 1.
- **storeSize**: le asigna la dimensión o el tamaño del al nodo de arreglo, y calcula R = R * tamaño.
- **addArrNode**: agrega un nuevo nodo donde se guardará la información de la segunda dimensión.
- **endArrNode**: calcula el valor de m para cada dimensión y asigna m = 0 al último nodo de arreglos.



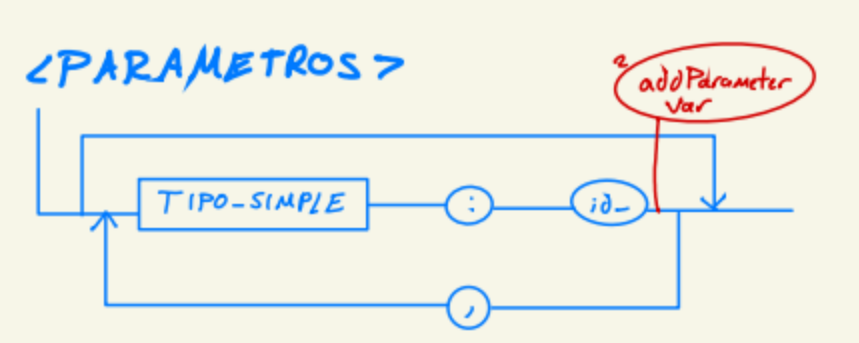
- ASIGNACIÓN
  - **addOperand**: Verifica que exista una variable en el scope local o global y agrega el operando a la pila de operandos y agrega el tipo a la pila de tipos.
  - **addOperatorAsig**: Agrega el operando '=' a la pila de operadores.
  - **removeFromStackAssign**: Elimina el operador de asignación de la pila de operadores y crea el quad que asigna la variable del lado izquierdo a la expresión del lado derecho.



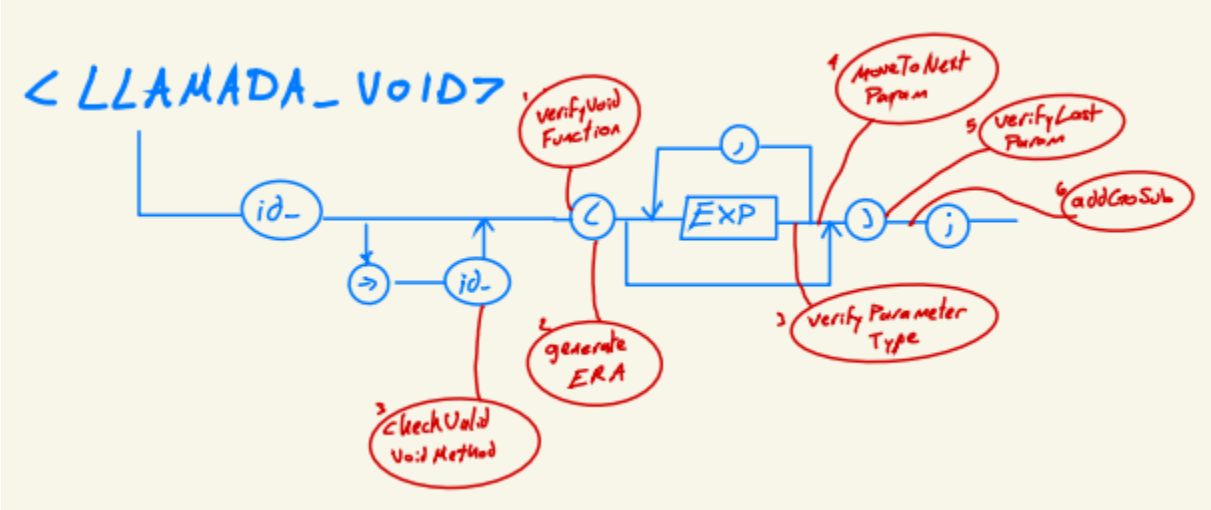
- ASIG
    - **checkValidAttribute**: Verifica que el atributo del objeto exista.
    - **verifyArrayDimensions**: busca la variable tipo arreglo y verifica que tenga dimensiones.
    - **getArrayDimensionResult**: asigna el resultado de una expresión al índice de una dimensión del arreglo. Ej a[exp]
    - **addArrayQuads**: agrega los cuádruplos necesarios para hacer el cálculo de la dirección virtual de memoria de una casilla de un arreglo



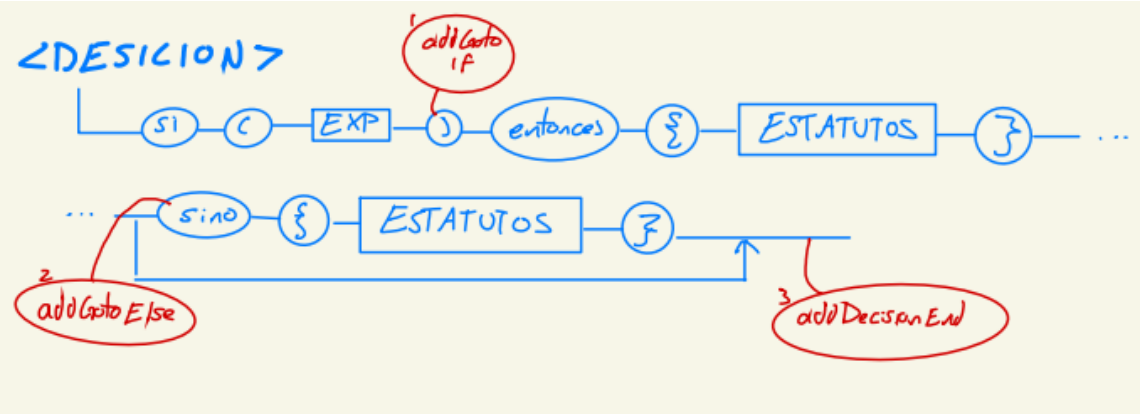
- FUNCIONES
    - **createFunc**: Verifica que no exista una función con el mismo nombre. Si no existe se crea la función y se agrega al directorio de funciones.
    - (sub programa) **addFuncGlobalVar**: agrega la función como variable global en el directorio de funciones para la función que representa el main.
    - (sub parámetros) **initParameterVars**: Inicializa el vector donde se guardarán los parámetros declarados en la firma de una función.
    - **addParameterVarsToDir**: Agrega los parámetros encontrados como variables locales a la tabla de variables de la función, y guarda la lista de tipos que utiliza la función (firma).
    - **addCurrLVarsToDir**: Agrega las variables locales declaradas a la tabla de variables de la función que se está desarrollando.
    - **addFuncCont**: guarda la dirección del cuádruplo inicial de los estatutos de la función
    - **addEndFunc**: Elimina la tabla de variables de la función que ya se terminó de declarar, guarda la cantidad de temporales utilizados en el desarrollo y agrega el fin de la función al cuádruplo.



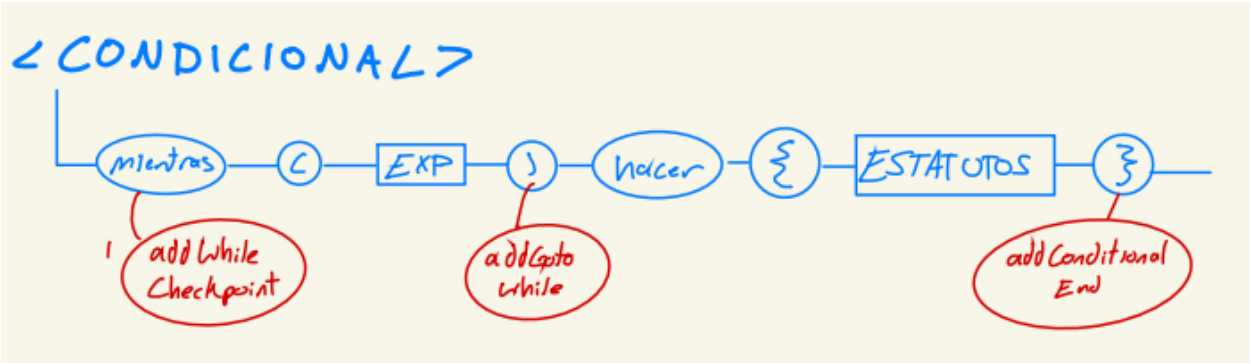
- PARAMETROS
    - **addParameterVar**: Comprueba que el nombre del parámetro no esté repetido y en caso de no ser así se agrega la información del parámetro en la lista de parámetros para la función.



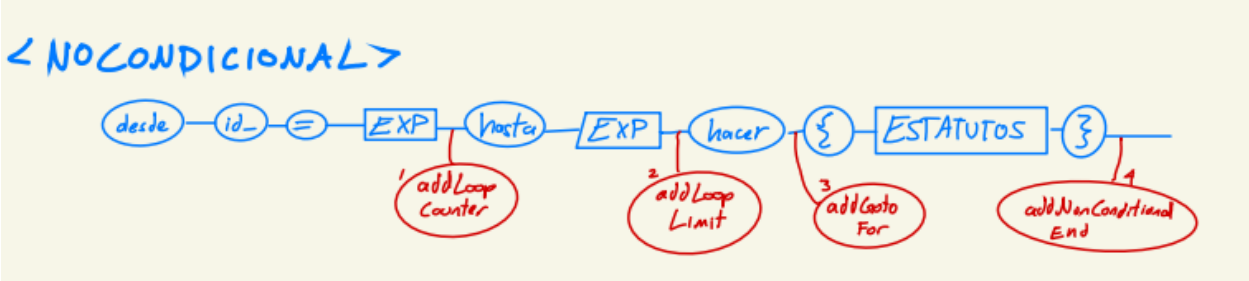
- LLAMADA_VOID
    - **verifyFunction**: Verifica que exista la función en el directorio de funciones y agrega la función a la pila de llamadas.
    - **generateEra**: Agrega al cuádruplo la instrucción de ERA y agrega un fondo falso.
    - **verifyParameterType**: Verifica que el parámetro recibido concuerde con los parámetros de la función a llamar.
    - **moveToNextParam**: Trae el siguiente parámetro.
    - **verifyLastParam**: Verifica que todos los parámetros necesarios se han recibido.
    - **addGoSub**: Agrega al cuádruplo la instrucción de GOSUB para la función.



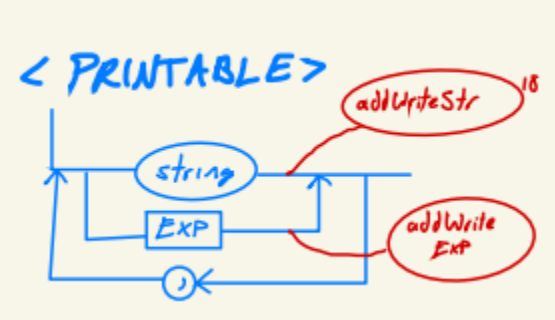
- DESICIÓN
  - **addGoToIf**: Si el tipo de expresión de la pila de operandos es “bool”, agrega un cuádruplo “GoToF” que depende del booleano y agrega la posición de este en la pila de saltos.
  - **addDecisionEnd**: Marca la decisión como terminada y regresa a llenar el “GoTo”.
  - **addGoToElse**: En caso de que exista un ‘else’ agrega una instrucción “GoTo” al cuádruplo y guarda el salto mientras se obtiene a dónde se irá el “GoTo” regresa a llenar el “GoToF”.



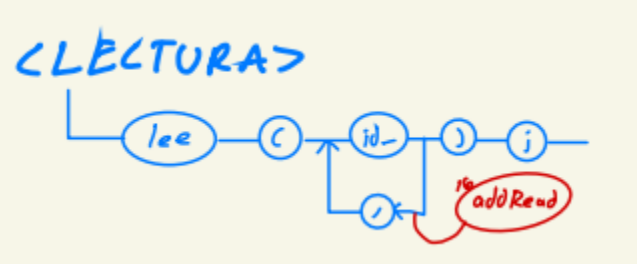
- CONDICIONAL
  - **addWhileCheckpoint**: Guarda el inicio del while en la pila de saltos.
  - **addGotoWhile**: Si el tipo de expresión es “bool” agrega una instrucción “GoToF” al cuádruplo y guarda el salto mientras se obtiene a donde se va el “GoToF”.
  - **addConditionalEnd**: Regresa a completar la instrucción del “GoToF” y agrega un “GOTO” al inicio de la condicional.



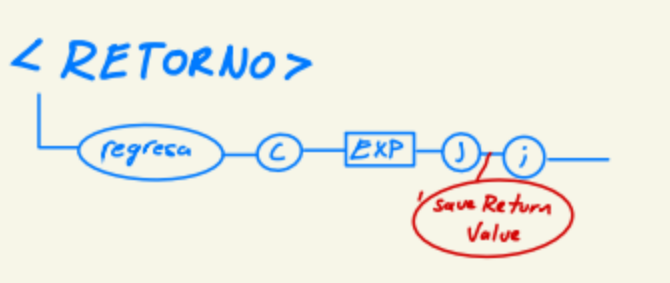
- NOCONDICIONAL
  - **addLoopCounter**: Si el contador es de tipo “int” se agrega a la memoria local la variable que guarda el conteo del “loop” y se le asigna su valor inicial.
  - **addLoopLimit**: se agrega el cuádruplo donde se compara el contador del “loop” con una expresión que indica el límite y se checa que el contador sea menor o igual al límite.
  - **addGotoFor**: Si el tipo de expresión es “bool” agrega una instrucción “GoToF” al cuádruplo y guarda el salto en la pila de saltos mientras se obtiene a donde se va el “GoToF”.
  - **addConditionalEnd**: Regresa a completar la instrucción del “GoToF” agrega un “GOTO” al inicio del loop.



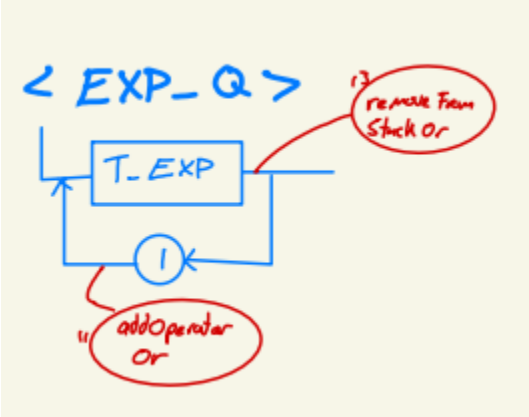
- PRINTABLE
  - **addWriteStr**: Agrega al cuádruplo una instrucción “write” con la información a imprimir de tipo string.
  - **addWriteExp**: Agrega al cuádruplo una instrucción “write” con la expresión a imprimir.



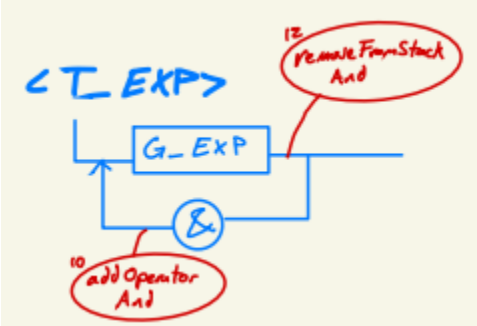
- LECTURA
  - **addRead**: Agrega al cuádruplo una instrucción “read” con la variable a leer.



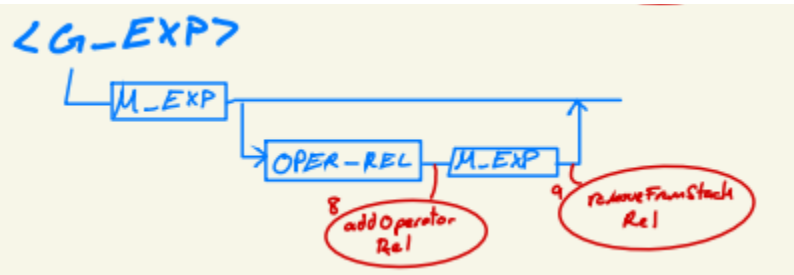
- RETORNO
    - **saveReturnValue**: Verifica que la función tenga un valor de retorno y lo regresa. Se agrega al cuádruplo una instrucción “return” con la variable a regresar.



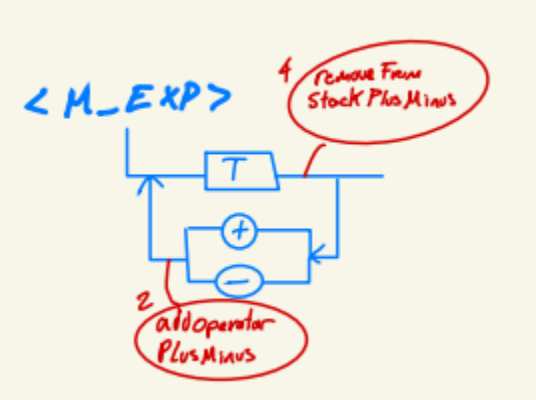
- EXP_Q
    - **addOperatorOr**: Agrega el operador “or” a la pila de operadores.
    - **removeFromStackOr**: Elimina el operador “or” a la pila de operadores.



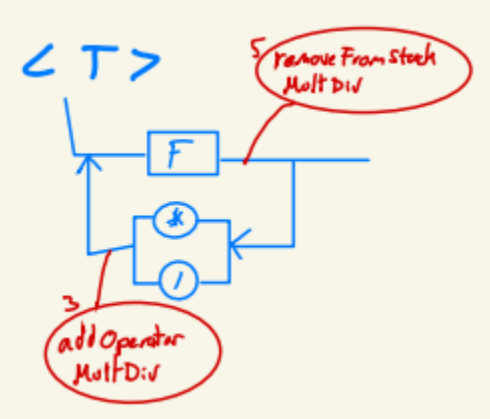
- T_EXP
    - **addOperatorAnd**: Agrega el operador “and” a la pila de operadores.
    - **removeFromStackAnd**: Elimina el operador “and” a la pila de operadores.



- G_EXP
    - **addOperatorRel**: Agrega el operador relacional “<, >, …” a la pila de operadores.
    - **removeFromStackRel**: Elimina el operador relacional a la pila de operadores.

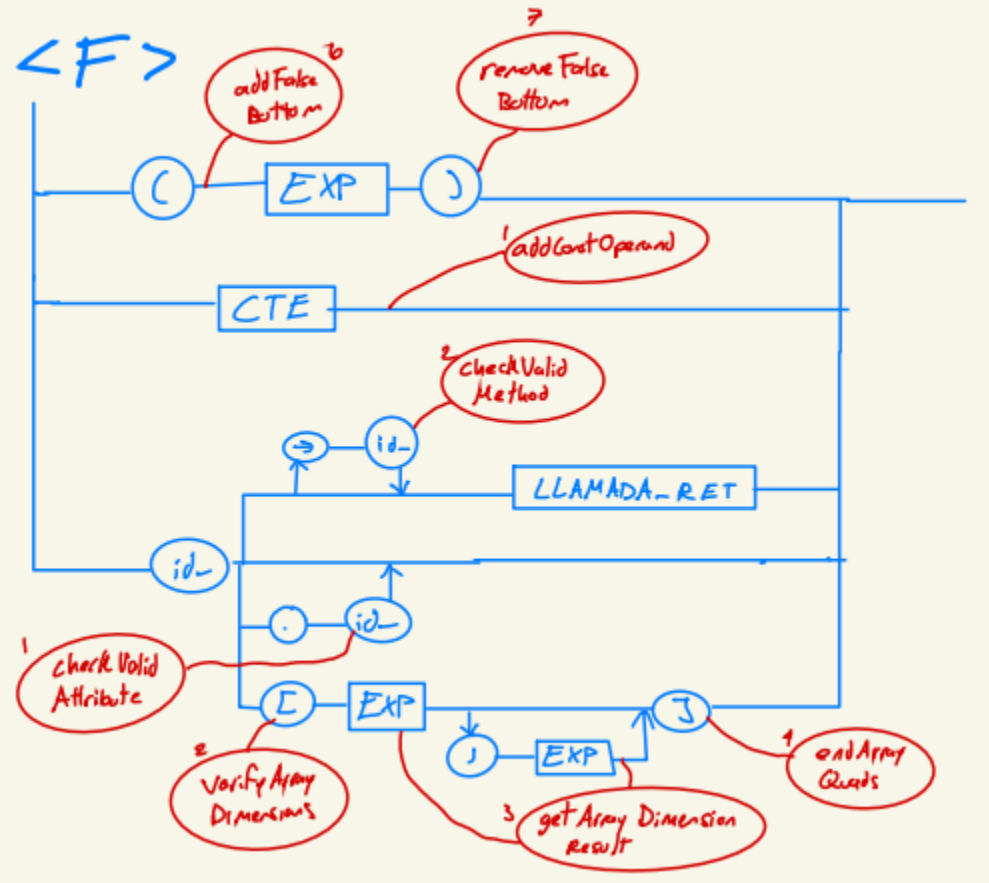


- M_EXP
    - **addOperatorPlusMinus**: Agrega el operador de suma o resta a la pila de operadores.
    - **removeFromStackPlusMinus**: Elimina el operador de suma o resta a la pila de operadores.

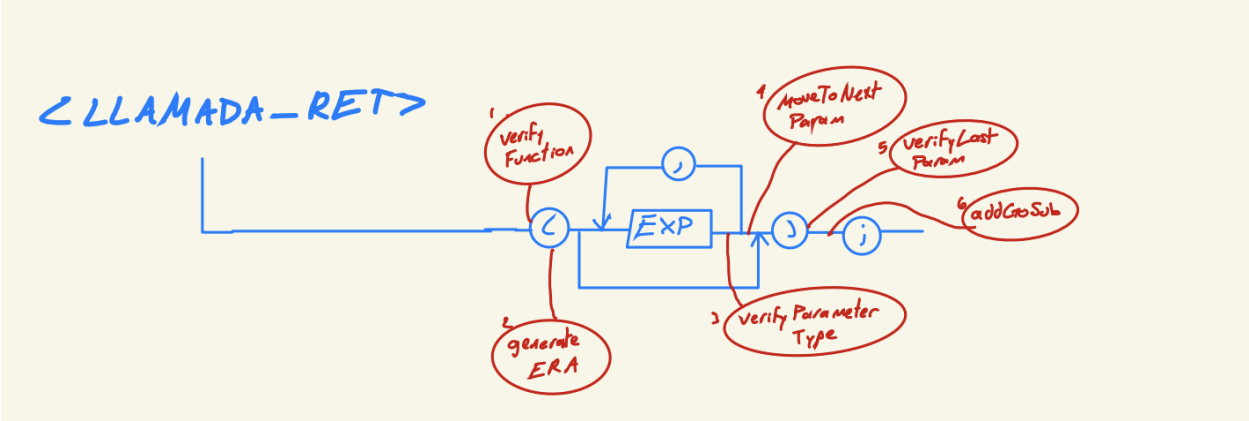


- T
    - **addOperatorMultDiv**: Agrega el operador de multiplicación o división a la pila de operadores.

- **removeFromStackMultDiv**: Elimina el operador de multiplicación o división a la pila de operadores.



- F
  - **addFalseBottom**: Agrega un fondo falso.
  - **removeFalseBottom**: Elimina el fondo falso.
  - **addConstOperand**: Agrega a la Tabla de Constantes una variable constante.
  - **checkValidAttribute**: Verifica que el atributo del objeto exista.
  - **verifyArrayDimensions**: busca la variable tipo arreglo y verifica que tenga dimensiones.
  - **getArrayDimensionResult**: asigna el resultado de una expresión al índice de una dimensión del arreglo. Ej a[exp]
  - **addArrayQuads**: agrega los cuádruplos necesarios para hacer el cálculo de la dirección virtual de memoria de una casilla de un arreglo
  - **addOperand**: Verifica que exista una variable en el scope local o global y agrega el operando a la pila de operandos y agrega el tipo a la pila de tipos.



- LLAMADA_RET
  - **verifyFunction**: Verifica que exista la función en el directorio de funciones y agrega la función al pila de llamadas.
  - **generateEra**: Agrega al cuádruplo la instrucción de ERA y agrega un fondo falso.
  - **verifyParameterType**: Verifica que el parámetro recibido concuerde con los parámetros de la función a llamar.
  - **moveToNextParam**: Trae el siguiente parámetro.
  - **verifyLastParam**: Verifica que todos los parámetros necesarios se han recibido.
  - **addGoSub**: Agrega al cuádruplo la instrucción de GOSUB para la función.
  - **addReturnValue**: Guarda el valor de retorno de una llamada a una función en una variable temporal.

Tabla de Consideraciones Semánticas

left_op	right_op	+	-	/	*	>	<	>=	<=
int	int	int	int	int	int	bool	bool	bool	bool
int	float	float	float	float	float	bool	bool	bool	bool
int	char	err	err	err	err	err	err	err	err
float	float	float	float	float	float	bool	bool	bool	bool
float	char	err	err	err	err	err	err	err	err
char	char	err	err	err	err	err	err	err	err
bool	int	err	err	err	err	err	err	err	err
bool	float	err	err	err	err	err	err	err	err
bool	char	err	err	err	err	err	err	err	err

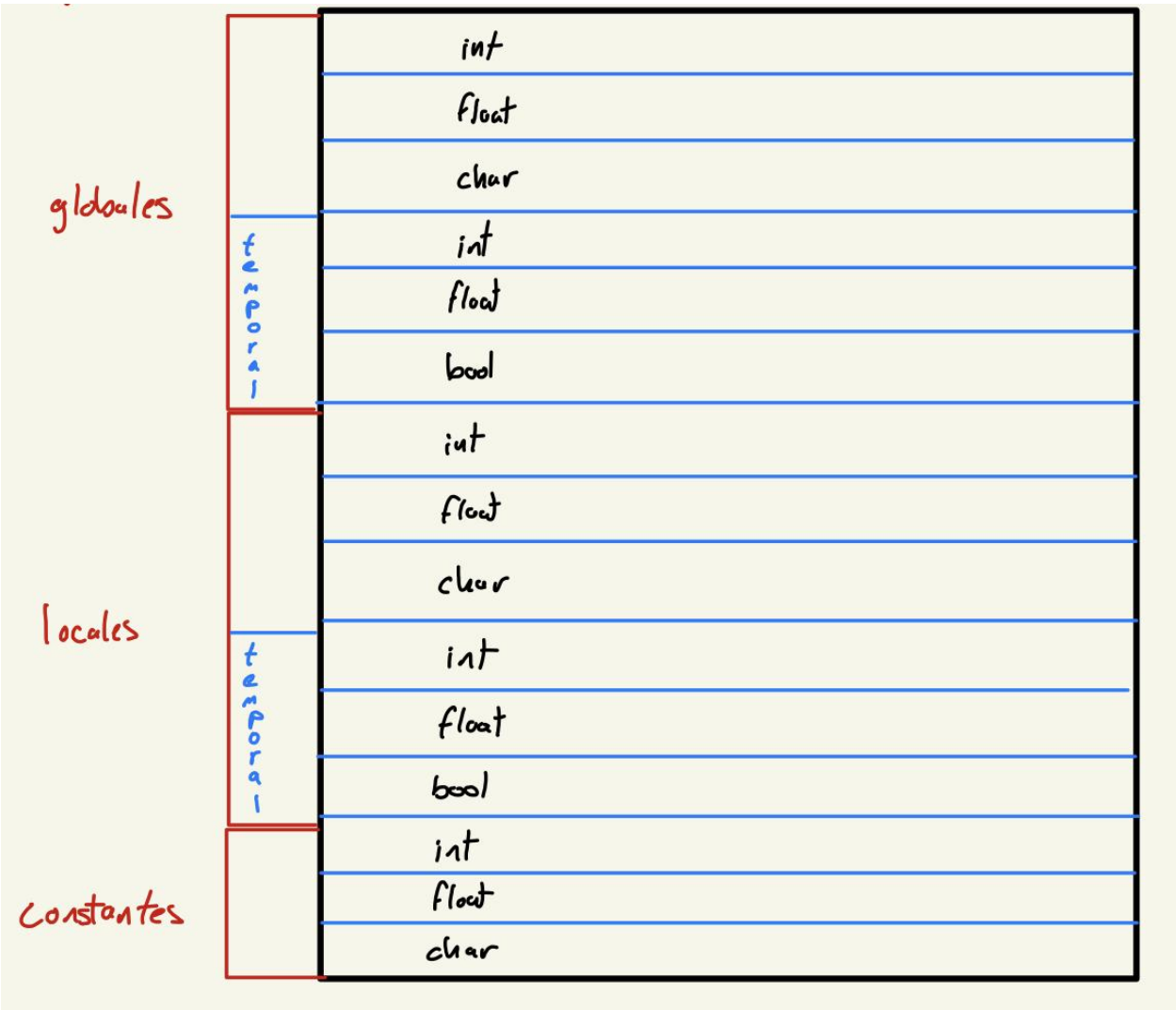
bool	bool	err	err	err	err	err	err	err	err

left_op	right_op	==	<>	&	|	=
int	int	bool	bool	err	err	int
int	float	bool	bool	err	err	err
int	char	err	err	err	err	err
float	float	bool	bool	err	err	float
float	char	err	err	err	err	err
char	char	bool	bool	err	err	char
bool	int	err	err	err	err	err
bool	float	err	err	err	err	err
bool	char	err	err	err	err	err
bool	bool	bool	bool	bool	bool	bool

Descripción del proceso de Administración de Memoria

Para asignar una dirección de memoria a cada variable en el proceso de compilación, se siguió el siguiente procedimiento:

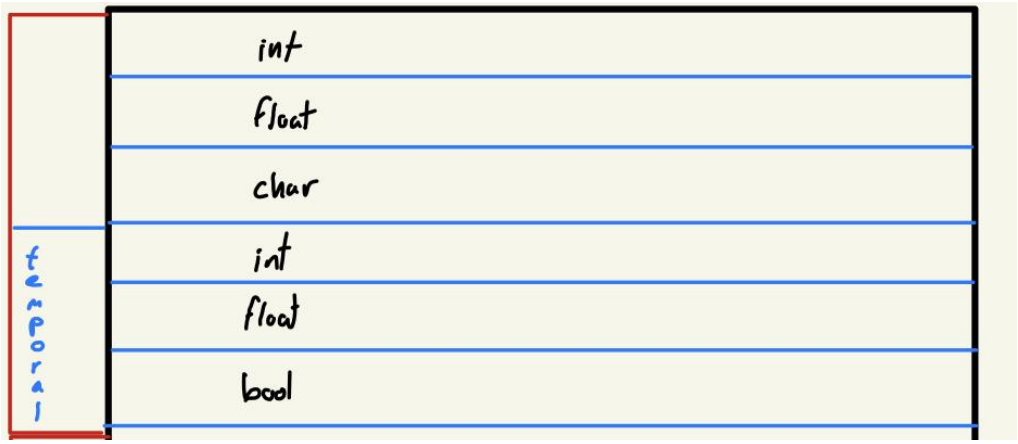
Mapa de Memoria



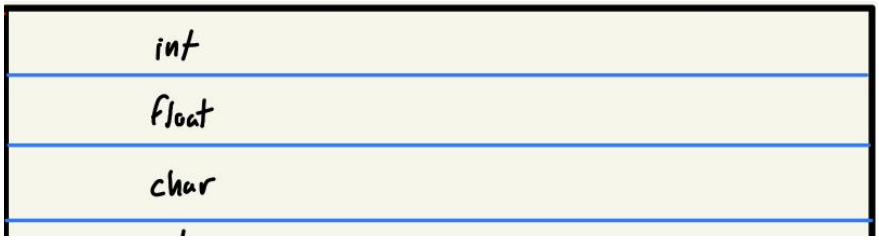
Se diseñó un mapa de memoria donde se describen las diferentes particiones que tiene la memoria. Esta se divide principalmente en 3 diferentes bloques:

- Memoria para variables Global: contiene variables declaradas y utilizadas en el main o principal
- Memoria para variables Local: contiene variables declaradas y utilizadas dentro de funciones y métodos
- Memoria para Constantes: contiene constantes declaradas en cualquier parte del código.

Hay una segunda división para las variables ‘Globales’ y ‘Locales’ que indica si una variable es temporal. Estas variables son utilizadas para cálculos intermedios en los cuádruplos.



La tercera y última división se crea dependiendo del tipo de variable ya sea ‘int’, ‘float’, ‘char’, o ‘bool’. Cabe destacar que no hay ‘chars’ tipo ‘temporal’ ya que no existen las operaciones con tipos ‘char’, y que el tipo ‘bool’ solamente existe temporalmente para facilitar operaciones lógicas en estatutos condicionales.



Para cada partición en memoria se calculó una dirección inicial correspondiente a su orden en el Mapa de Memoria. Adicionalmente cada partición tiene un tamaño máximo de casillas que pueden ser utilizadas.

Por ejemplo, las variables globales / no temporales / tipo int, inician en la dirección ‘0’ y tienen un máximo de 2,000 casillas disponibles. Con la dirección inicial y su tamaño máximo sabemos que la primera y última dirección son 0 y 1,999.

Sabiendo esto se calculó la dirección inicial de la siguiente partición: variables globales / no temporales / tipo float, con dirección inicial ‘2,000’ y un máximo de 2,000 casillas disponibles. Así sucesivamente para todas las particiones se definieron las posiciones iniciales y rango máximo de casillas disponibles.

Ya que se definieron las casillas, el siguiente paso fue crear la administración de memoria para que se puedan otorgar posiciones de memoria conforme se vayan ocupando en la generación de cuádruplos. Para solucionar este problema se inicializan contadores para cada partición, creando en un total 15 contadores:

- Globales
    - Int
    - Float

    - Char
    - Temporal int
    - Temporal float
    - Temporal char
- Locales
    - Int
    - Float
    - Char
    - Temporal int
    - Temporal float
    - Temporal char
- Constantes
    - Int
    - Float
    - Char

Cada contador tiene la función de guardar la cantidad de casillas que se han ocupado, así como ayudar a calcular la dirección de la siguiente casilla disponible en caso de que se necesite reservar otro espacio.

Cada vez que se necesita reservar un espacio de memoria para un cierto tipo, se llama la función correspondiente 'reverse*Tipo*Memory'. Esta función se encarga de calcular el siguiente espacio de memoria disponible con ayuda del contador para el tipo y la dirección de memoria inicial, así como incrementar el contador de espacios utilizados.

Existen 4 variaciones de la 'reverse*Tipo*Memory', en la cual pueden variar los parámetros de esta.

- reverse*Int*Memory(scope, isTemp, size): se necesita el scope para saber si se reservará memoria de la partición global o local. También se necesita saber si es temporal o no para reservar en su partición respectiva. Últimamente se puede indicar el tamaño de memoria que se necesita en caso de que la variable sea tipo arreglo.
  Dentro de la función:
    - Se valida que el contador de memoria para la partición no exceda el tamaño máximo,
    - Se calcula la dirección de memoria sumando la dirección inicial con el contador de memoria
    - Se incrementa el contador de memoria para la partición utilizada

- reverse*Float*Memory(scope, isTemp, size): se necesita el scope para saber si se reservará memoria de la partición global o local. También se necesita saber si es temporal o no para reservar en su partición respectiva. Últimamente se puede indicar el tamaño de memoria que se necesita en caso de que la variable sea tipo arreglo.
  Dentro de la función:

    - Se valida que el contador de memoria para la partición no exceda el tamaño máximo,
    - Se calcula la dirección de memoria sumando la dirección inicial con el contador de memoria
    - Se incrementa el contador de memoria para la partición utilizada

- reverse*Char*Memory(scope, size): esta función no tiene como parámetro 'isTemp' ya que no existen los chars temporales. Solamente le importa si la variable es global o local, y si es tipo arreglo.
  Dentro de la función:
    - Se valida que el contador de memoria para la partición no exceda el tamaño máximo,
    - Se calcula la dirección de memoria sumando la dirección inicial con el contador de memoria
    - Se incrementa el contador de memoria para la partición utilizada

- reverse*Bool*Memory(scope): esta función solamente necesita saber el scope de la variable ya que los booleanos siempre serán temporales y siempre tendrán tamaño 1 ya que no hay arreglos tipo 'bool'.
  Dentro de la función:
    - Se valida que el contador de memoria para la partición no exceda el tamaño máximo,
    - Se calcula la dirección de memoria sumando la dirección inicial con el contador de memoria
    - Se incrementa el contador de memoria para la partición utilizada

Ej. se va a reservar memoria para variables globales tipo float:
...
variables
    float : a, b[10], c;
…

Dirección inicial de variables globales / no temporales / tipo float: 2,000
Tamaño máximo para partición: 2,000
Contador para variables globales / no temporales / tipo float: 0

1. Encuentra la variable 'a'
    a. Se llama la función 'reserveFloatMemory(scope:global, isTemp:false, size: 1)'
    b. Checa que el contador+size para esta partición (1) no sea mayor a 2,000
    c. Calcula la dirección sumando el contador (0) + dirección inicial (2,000) = 2,000
    d. Suma 'size' (1) al contador
    e. Regresa la dirección 2,000
2. Encuentra la variable 'b[10]'
    a. Se llama la función 'reserveFloatMemory(scope:global, isTemp:false, size: 10)'

b. Checa que el contador+size para esta partición (11) no sea mayor a 2,000
c. Calcula la dirección sumando el contador (1) + dirección inicial (2,000) = 2,001
d. Suma ‘size’ (10) al contador
e. Regresa la dirección 2001

3. Encuentra la variable c
    a. Se llama la función ‘reserveFloatMemory(scope:global, isTemp:false, size: 1)’
    b. Checa que el contador+size para esta partición (12) no sea mayor a 2,000
    c. Calcula la dirección sumando el contador (11) + dirección inicial (2,000) = 2,011
    d. Suma ‘size’ (1) al contador
    e. Regresa la dirección 2,011

IV. Descripción de la máquina virtual

Descripción del proceso de Administración de Memoria

Se diseñó un mapa de memoria donde se describen las diferentes particiones que tiene la memoria. Esta se divide principalmente en 3 diferentes bloques:

- Memoria para variables Global: contiene variables declaradas y utilizadas en el main o principal
- Memoria para variables Local: contiene variables declaradas y utilizadas dentro de funciones y métodos
- Memoria para Constantes: contiene constantes declaradas en cualquier parte del código.

Para la **memoria local** se utilizará un vector dinámico que incrementa y decrementa en espacios de memoria dependiendo del espacio requerido con cada llamada a las funciones.

Cada vez que se encuentre una instrucción ERA, se crearán los espacios correspondientes al final del vector de tal manera que funciona como una pila de memoria donde se guardan todas las variables de las funciones activas. Por lo mismo, cuando se encuentre un ENDFUNC se liberarán los espacios asignados a la función que acaba de terminar.

Ej. de llamada simple

Memoria Local para Ints

Vacio

Se llama una función f1(int a, int b) y con el ERA se crean dos espacios nuevos

_ | _

Se pasan los parámetros ‘a’ y ‘b’:

a | b

Termina la función y se eliminan los espacios con ENDFUNC

Vacio

Para acceder y guardar las variables en su espacio relativo cuando hay muchas llamadas, se creará también una pila de offsets donde se guarda la dirección inicial relativa al contexto de memoria de la función. Cada vez que se encuentra un ERA, se agrega a la pila de “offsets" el tamaño actual del vector de memoria, y luego se crean los espacios de memoria. De esta manera al buscar una dirección para una función, se le puede sumar su offset o inicio relativo y que a partir de ahí busque las variables para una función

Ej. de llamada anidada en una función

Memoria Local para Ints

Vacio

Offsets de memoria Local para Ints

Vacio

Se llama una función f1(int a, int b) y con el ERA se crean dos espacios nuevos

_ | _

Offsets de memoria Local para Ints

0

Se pasan los parámetros ‘a’ y ‘b’:

a | b

Offsets

0

Se llama una función f2(int x) que tiene una variable local ‘y’. Con el ERA se crean dos espacios nuevos

a | b | _ | _

Offsets. Se crea nuevo offset ‘2’ ya que las direcciones para la función ‘f2’ se encuentran desplazadas 2 espacios en el vector de memoria local ints.

0 | 2

Se pasa el parámetro ‘x’ y la variable ‘y’:

a | b | x | y

Offsets.

0 | 2

(En este punto si quiere acceder a la variable ‘x’ con dirección local 5000, su posición local relativa sería 0, pero como tiene un offset de 2 en el vector de memoria local, se le suma el offset y termina siendo la posición 2.)

Termina la función f2 y se liberan los espacios con ENDFUNC

a | b

Offsets. Se libera el offset de ‘f2’

0

Termina la función f1 y se liberan los espacios con ENDFUNC

Vacio

Offsets.

Vacio

Cada vez que se llama un ‘ERA’, se crea el espacio de memoria para esa función. Habrá casos donde haya que acceder variables dentro del vector que corresponden a otro contexto.
Por ejemplo, cuando se encuentra dentro de una llamada y se tiene una expresión como la siguiente:
Dentro de ‘f1’ -> f3(a, 5)

Nos encontramos dentro de la llamada a f1, por lo que ya existe un espacio para esta función

a | b

Offsets.

0

Después se encuentra otra llamada a una función f3 y crea su memoria y se asigna un nuevo offset

a | b | _ | _

Offsets.

0 | 2

Aquí surge un problema ya que a la hora de asignar el parámetro ‘a’ del contexto de la función actual, se tiene como offset el valor ‘2’, por lo que buscará la variable ‘a’ en las dos casillas recientemente creadas que se encuentran vacías.
En ese caso lo correcto sería cambiar el offset actual al contexto de la función activa ‘f1’ y que tenga el valor ‘0’. De esta manera al buscar la variable ‘a’ la encontraría fácilmente en las primeras dos casillas.

Por esta razón se agregó otra pila que guarda el orden de los contextos. De esta manera se puede cambiar de contextos después de haber creado el ERA para una llamada a una función que tiene como parámetros variables de la función local actual.

Esta pila de contextos funcionaria de la siguiente manera:
Cada valor al tope de la pila representa el índice que debería de estar activo en el vector de offsets.

Cuando se encuentra un Gosub, se agrega a la pila de contextos el offset más reciente
Cuando se encuentra un ENDFUNC, se hace un Pop de la pila de contextos y regresa al contexto previo.

Nos encontramos dentro de la llamada a f1, por lo que ya existe un espacio para esta función

a | b

Offsets de memoria Local para Ints, con contexto activo [0]

0

Después se encuentra otra llamada a una función f3 y crea su memoria y se asigna un nuevo offset

a | b | _ | _

Con contextos disponibles [0, 1]

0 | 2

Se pasan los parámetros ‘a’ y se asigna la variable ‘5’:

a | b | a | 5

Con contextos disponibles [0, 1]

0 | 2

Se hace un Gosub a ‘f3’

a | b | a | 5

Con contexto activo [0, 1]

0 | 2

Acaba la función ‘f3’ y se liberan sus variables

a | b

Se hace pop al contexto de la función ‘f3’, con contexto activo [0]

0

Termina la función f1 y se liberan los espacios con ENDFUNC

Vacio

Se elimina el offset de ‘f1’ y el contexto activo

Vacio

Para la **memoria global** se sigue un proceso similar, la única diferencia es que esta memoria se crea una sola vez al iniciar el programa y no aumenta o decrementa en tamaño. Por esta misma razón para la memoria global no se necesita tener una variable que mantenga el tanto de offsets, y solamente se necesita el espacio de memoria virtual y la dirección de la partición inicial para acceder a la casilla correspondiente.

Ej. para particiones de variables globales tipo int, float y char

Memoria global int

Vacio

Memoria global float

Vacio

Memoria global char

Vacio

Al leer el ERA de main observamos que necesitan:

- 3 ints
- 2 floats
- 2 chars

Por lo que se separan los espacios correspondientes

Memoria global int

_ | _ | _

Memoria global float

_ | _

Memoria global char

_ | _

Conforme se vayan asignando sus valores en principal o dentro de alguna función, estos se irán actualizando.
Por ejemplo si tenemos en el código fuente:

```
y = 10.0;
```

Y lo traducimos a sus direcciones virtuales

```
4001 = 35000
```

Para accesar la casilla que representa ‘y’:

1. Se calcula su posición relativa restando la posición inicial de las variables globales flotantes de la posición original: 4001 - 4000
2. Este resultado (1) nos indica que ‘y’ es la segunda variable en el vector de memoria para esa partición

Memoria global float

```
(0)_ | (1)y
```

De esta manera podemos acceder para recuperar o guardar cualquier valor de las variables globales.

V. Pruebas de funcionamiento del lenguaje

Fibonacci:

Código fibonacci sin recursión.

```
programa fibonacci;
variables
   int : x, i, f[50];

principal() {
        escribe("Cantidad de elementos de la serie fibonacci:" );
        lee(x);

        f[0] = 0;
        f[1] = 1;

        desde i = 2 hasta x + 1 hacer
        {
                f[i] = f[i - 1] + f[i - 2];
        }

        desde i = 0 hasta x hacer
        {
                escribe(f[i]);
        }
}
```

Extracto de código intermedio

```
fibonacci, 0, 54, 0, 0, 12, 0, 2
%%
2, 30004
49, 30003
50, 30002
1, 30001
0, 30000
%%
16, principal, -, 1
19, -, -, "Cantidad de elementos de la serie fibonacci:"
20, -, -, 0
23, 30000, 30000, 30003
0, 30000, 30004, 6000
12, 30000, -, (6000)
23, 30001, 30000, 30003
0, 30001, 30004, 6001
12, 30001, -, (6001)
12, 30004, -, 52
0, 0, 30001, 6002
5, 52, 6002, 12000
14, 12000, -, 26
23, 52, 30000, 30003
0, 52, 30004, 6003
```

Ejecución

```
"Cantidad de elementos de la serie fibonacci:"
Lectura: 10
0
1
1
2
3
5
8
13
21
34
---End---
(base) $
```

Fibonacci Recursivo:

Código fibonacci utilizando recursión.

```
programa fibonacci;
variables
   int : x;

funcion int fibR(int : x);
{
        si (x <= 1) entonces {
                regresa(x);
        } sino {
      regresa(fibR(x - 1) + fibR(x - 2));
```

```
        }
        %%escribe("DESPUES");
}

funcion void imprimeSequenciaFib(int : n);
{
        escribe(0);
        desde a = 1 hasta n hacer {
                escribe(fibR(a));
        }
}

principal() {
        escribe("Cantidad de elementos de la serie fibonacci:" );
        lee(x);
        imprimeSequenciaFib(x);
}
```

Extracto de código intermedio

```
1   imprimeSequenciaFib, 18, 2, 0, 0, 2, 0, 1
2   fibR, 1, 1, 0, 0, 5, 0, 1
3   fibonacci, 0, 2, 0, 0, 0, 0, 0
4   %%
5   2, 30002
6   1, 30001
7   0, 30000
8   %%
9   16, principal, -, 31
10  7, 15000, 30001, 27000
11  14, 27000, -, 5
12  21, 1, -, 15000
13  16, -, -, 17
14  17, fibR, -, -
15  1, 15000, 30001, 21000
16  13, 21000, -, param0
17  15, fibR, -, 1
18  12, 1, -, 21001
19  17, fibR, -, -
20  1, 15000, 30002, 21002
21  13, 21002, -, param0
22  15, fibR, -, 1
23  12, 1, -, 21003
24  0, 21001, 21003, 21004
```

Ejecución

```
"Cantidad de elementos de la serie fibonacci:"
Lectura: 10
0
1
1
2
3
5
8
13
21
34
---End---
(base) $
```

Factorial:

Código factorial sin recursión.

```
programa factorial;
variables
   int : x, i, res;

principal() {
        escribe("Numero factorial a calcular:");
        lee(x);
        res = 1;

        desde i = 2 hasta (x+1) hacer{
                res = res * i;
        }

        escribe("Solución: ", res);
}
```

Ejecución

```
(base) $ ./maquina
"Numero factorial a calcular:"
Lectura: 5
"Solución: "
120
---End---
```

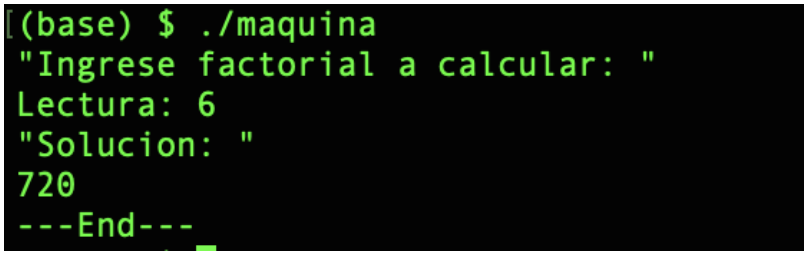
Factorial Recursivo:

Código factorial utilizando recursión.

```
programa factorial;
variables
   int : x, res;
```

```
funcion int factorialR(int : n);
{
        si (n == 0) entonces {
      regresa(1);
        } sino {
              regresa (n * factorialR(n - 1));
        }
}

principal() {
        escribe("Numero factorial a calcular:");
        lee(x);
        res = factorialR(x);
        escribe("Solución: ", res);
}
```

Ejecución



Sort:

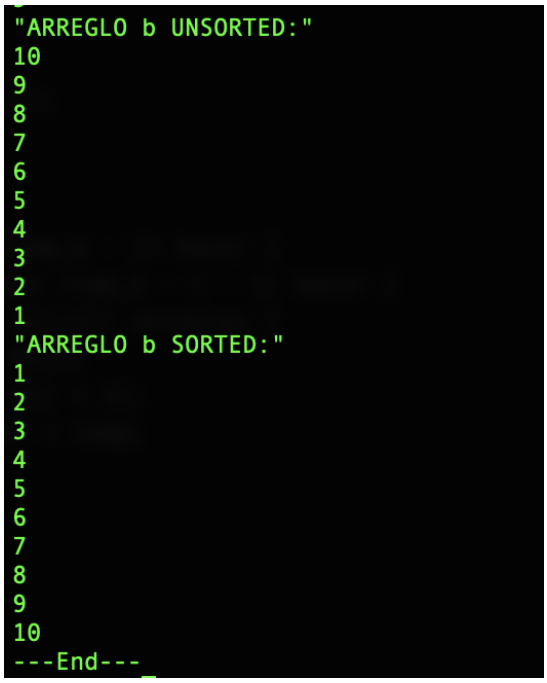
Código de función BubbleSort suponiendo que se tiene un arreglo global ‘b[10]’ y tam_b = 10

```
funcion void bubbleSort();
variables
   int : temp;
{
   desde i = 0 hasta (tam_b - 1) hacer {
      desde j = 0 hasta (tam_b - i - 1) hacer {
         si (b[j] > b[j+1]) entonces {
            temp = b[j];
            b[j] = b[j + 1];
            b[j + 1] = temp;
         }
      }
   }
}
```

Ej. se tiene b[10] = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1];

Se llama bubbleSort desde principal
bubbleSort();
Resulta en b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Ejecución



Find:

Código de función Find suponiendo que se tiene un arreglo global ‘a[10]’ y tam_a = 10

```
funcion int findInt(int : x);
variables
   int : idx;
{
   idx = -1;

   desde i = 0 hasta tam_a hacer {
      si (a[i] == x) entonces {
         idx = i;
      }
   }

   regresa(idx);
}
```

Ej. a[10] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
Se llama Find desde principal

Find(3);
Regresa ‘2’
Find(11);
Regresa ‘-1’

Extracto de código intermedio

```
    bubbleSort, 15, 3, 0, 0, 14, 0, 3
    findInt, 1, 3, 0, 0, 2, 0, 2
    sort_find, 0, 28, 0, 0, 12, 0, 5
    %%
    12, 30005
    9, 30004
    10, 30003
    -1, 30002
    1, 30001
    0, 30000
    %%
    16, principal, -, 51
    12, 30002, -, 15001
    12, 30000, -, 15002
    5, 15002, 10, 27000
    14, 27000, -, 13
    23, 15002, 30000, 30004
    0, 15002, 30000, 21000
    8, (21000), 15000, 27001
    14, 27001, -, 10
    12, 15002, -, 15001
    0, 15002, 30001, 21001
    12, 21001, -, 15002
    16, -, -, 3
    21, 23, -, 15001
    18, -, -, -
    12, 30000, -, 15001
    1, 22, 30001, 21000
    5, 15001, 21000, 27000
    14, 27000, -, 50
    12, 30000, -, 15002
```

Ejecución

```
(base) $ ./maquina
"Ingresa el número deseas buscar: "
Lectura: 3
"Se encontro en el indice: "
2
```

```
(base) $ ./maquina
"Ingresa el número deseas buscar: "
Lectura: 11
"No se encontro en el arreglo 'a'"
```

Multiplicación de matrices:

Código de función matrixMult suponiendo que se tienen 3 arreglos a, b, c todos con dimensiones [3, 3] y tam = 3;

```
funcion void matrixMult();
{
   desde i = 0 hasta tam hacer {
      desde j = 0 hasta tam hacer {
         desde k = 0 hasta tam hacer {
            a[i, j] =  a[i, j] + b[i, k] * c[k, j];
         }
      }
   }
}
```

Si matriz b

0	0	0
1	1	1
2	2	2

y matriz c

0	1	2
0	1	2
0	1	2

Resultado matriz a

0	0	0
0	3	6
0	6	12

Extracto de código intermedio

```
matrixMult, 55, 5, 0, 0, 25, 0, 5
dotProduct, 1, 4, 0, 0, 20, 0, 4
multMat, 0, 30, 0, 0, 5, 0, 2
%%
18, 30005
9, 30004
1, 30001
2, 30003
3, 30002
0, 30000
%%
16, principal, -, 121
12, 30000, -, 15000
5, 15000, 27, 27000
14, 27000, -, 25
12, 30000, -, 15001
5, 15001, 27, 27001
14, 27001, -, 22
23, 15000, 30000, 30003
3, 15000, 30002, 21000
23, 15001, 30000, 30003
0, 21000, 15001, 21001
0, 21001, 30004, 21002
12, 30003, -, (21002)
23, 15000, 30000, 30003
3, 15000, 30002, 21003
23, 15001, 30000, 30003
```

Ejecución

```
(base) $ ./maquina
0
0
0
"------"
0
3
6
"------"
0
6
12
"------"
---End---
```

VI. Documentación del código del lenguaje

Comentarios de Documentación e Implementación

La función addVarsToDir de la TablaSimbolos tiene como objetivo guardar todas las variables del programa con su dirección virtual asignada. Esta función no tiene valor de retorno. Para lograr guardar las variables recibe funcId (el identificador de la función actual) y un vector con las variables a guardar en la tabla de variables para la función correspondiente. En caso de que exista una variable tipo objeto, se hace una llamada recursiva para guardar los atributos del objeto de tal manera de que se guarda la variable compuesta por el nombre del 'objeto.attributo'.

```
void TablaSimbolos::addVarsToDir(std::string funcId, std::vector<VarEntry>
vars)
{
   if ((*funcDir)[funcId].varDir == nullptr)
   {
         std::cout << "ERROR: Empty variable directory for function: " <<
funcId << "\n";
      return;
   }

   // guarda direcciones virtuales
   // memory: local or global, varType, non temp
   for (VarEntry &var : vars)
   {
      int memAddr = -1;
      std::string scope = "local";
      if (currentFuncDecl == "main")
         scope = "global";

      std::vector<ArrNode> arrNodes = var.arrNodes;
      bool notArray = arrNodes.empty();
      int size = 1;
      for (ArrNode &node : arrNodes)
      {
         size *= node.size;
      }

      int type = cubo.typeMap[var.varType];

      // caso especial para tipo objetos
```

```
    if (type == 0 && var.varType != "int")
        type = 4;

    if (type != 4)
    {
        if (scope == "global")
            (*funcDir)[programName].numLVar[type] += size;
        else
            (*funcDir)[currentFuncDecl].numLVar[type] += size;
    }

    switch (type)
    {
    case 0:
    {
        memAddr = memoria->reserveIntMemory(scope, false, size);
        break;
    }
    case 1:
    {
        memAddr = memoria->reserveFloatMemory(scope, false, size);
        break;
    }
    case 2:
    {
        memAddr = memoria->reserveCharMemory(scope, size);
        break;
    }
    case 4:
    {
        // checar que exista clase
        if ((*clasesDir).count(var.varType) == 0)
        {
            std::cout << "ERROR: Clase no existe\n";
            return;
        }

        std::string className = var.varType;
        ClassEntry &classEntry = (*clasesDir)[className];
```

```
            std::vector<VarEntry> classVariables;
            for (auto classVar : classEntry.varTable)
            {
                classVariables.push_back({var.varName + "." + classVar.first,
classVar.second, -1, {}});
            }

            // agrega objeto a directorio de funciones pero no ocupa memoria
                    (*(*funcDir)[funcId].varDir)[var.varName]  =  {var.varName,
var.varType, -1, {}};

            addVarsToDir(funcId, classVariables);
        }
        }

        if (memAddr == -1 && type != 4)
        {
                std::cout << "ERROR: cannot reserve space for variable " +
var.varName + "\n";
        }

        var.memoryAddr = memAddr;
    }

    for (VarEntry var : vars)
    {
        (*(*funcDir)[funcId].varDir)[var.varName] = var;
    }
}
```

La función de getNextAvail de Quad tiene un valor de retorno de tipo string y tiene como parámetro una variable “type” de tipo string. Esta función es llamada cada vez que alguna función necesite una temporal. La función de getNextAvail le asigna un espacio de memoria a esa temporal y la guarda en la tabla de variables. También guarda un contador para cada tipo de temporal para que se pueda hacer el cálculo de ERA correctamente para la función correspondiente.

```
std::string Quad::getNextAvail(const std::string type) {
    if (sTemps.empty()) {
                    std::cout  <<  "ERROR:  Contador  temporal  vacío  para
disponibilidad\n";
        return "err";
```

```
    }

    std::string avail =  "t" + std::to_string(sTemps.back());
    sTemps.back() += 1;

    int tempType = cubo.typeMap[type];
    std::string currentFunc = (*tablasDatos).currentFunc;
    std::vector<int> &tempTypes = sTempTypes.back();
    tempTypes[tempType]++;

    // asigna direccion virtual a temp
    int memAddr = -1;
    std::string scope = "local";
    if (currentFunc == "main") scope = "global";

    switch (tempType) {
        case 0: // int
            memAddr = memoria->reserveIntMemory(scope, true, 1);
            break;
        case 1: // float
            memAddr = memoria->reserveFloatMemory(scope, true, 1);
            break;
        case 3: // bool
            memAddr = memoria->reserveBoolMemory(scope);
            break;
    }

    VarEntry var { avail, type, memAddr , {}};
    if (currentFunc == "main") {

(*(*tablasDatos).funcDir[(*tablasDatos).programName].varDir)[avail] = var;
    } else {
        (*(*tablasDatos).funcDir[currentFunc].varDir)[avail] = var;
    }

    return avail;
}
```

VII. Quick Reference Manual

Primero se tiene que compilar el código fuente.
Para lograr esto se tiene que crear un archivo nombre.txt en el directorio ./compilador.
Cuando lo quieras compilar puedes correr el comando en la terminal
./parser < ./nombreArchivo

Esto generará un archivo 'codigo.cmm' con el código intermedio

Para ejecutar el código habrá que navegar al directorio de ./maquina_virtual
Aquí se puede correr el comando ./maquina para ejecutar el código compilado

* Las secciones en itálicas son opcionales (pudiera o no venir).
* Las palabras y símbolos en bold/negritas son reservadas y el %% indica comentario

Inicio y estructura de programa:

Programa Nombre_prog ;
<Declaración de Clases>
<Declaración de Variables Globales>
<Definición de Funciones>
principal()
{
    <estatutos>
}

Declaración de variables globales:

variables %%Palabra reservada
    tipo **:** lista_ids**;**

donde
tipo = puede ser entero, flotante, char y id (donde este es el nombre de una clase previamente declarada).
lista_ids = identificadores separados por comas.
Cada identificador puede ser tipo regular, o tipo arreglo de 1 o 2 dimensiones.
    Regular - x
    Una dimension - x[10]
    Dos dimensiones - x[4, 4]
Ej.
    variables
        int a, b[5], c, d[10, 10];
        float x, y, z[10];

Declaración de funciones globales:

<tipo-retorno> **funcion** nombre_módulo **(** *<parámetros>* **);**
<Declaración de Variables Locales>
{
<estatutos> %% El lenguaje soporta llamadas recursivas.
}

Los **<parámetros>** siguen la sintaxis de la declaración de variables de tipo simple y únicamente son de entrada.
Ejemplo de parámetros. (int: a, float: b, char: c)

<tipo-retorno> puede ser de cualquier tipo simple soportado (entero, flotante, char) o void (si no regresa valor). En caso de que la función no sea void, se necesita incluir un estatuto 'regresa'.

Ej.

```
int f1(int: x, int: y);
variables
        float z;
{
        z = x * y * 3.14;
        regresa(z);
}
```

Declaración de clases:

Clase id; %% id siendo el nombre de la clase
{
atributos < *declaración de atributos* > %%sigue la sintaxis de una declaración de variables
metodos < *declaración de métodos* > %% sigue la sintaxis de una declaración de funciones.
};

Ej.

```
Clase carro;
{
        atributos
                int : numPasajeros;
                float: velocidadMaxima;
        metodos
                funcion float calculaVelocidadMaxima(int: pesoDePasajeros);
```

```
                {
                        regresa(pesoDePasajeros * 0.6);
                }
};
```

Uso de Clases:

Declaración de objeto

tipo_objeto **:** nombre_variable**;**

Ej. carro: toyota;

Acceso a atributo

Nombre_variable**.**nombre_atributo**;**

Ej. toyota.numPasajeros;

Acceso a función

Nombre_variable**->**nombre_funcion**(**<*parametros*>**)**;

Ej. toyota->calculaVelocidadMaxima(pesoDePasajeros);

Lectura:

lee(variable**);**

Ej. lee(x);

Nota* la variable a leer debe de estar previamente declarada.

Escritura:

Una expresión

escribe(variable**);**

Ej. escribe(2 + 2);

Múltiples expresiones

escribe(“letrero”**,** variable**);**

Ej. escribe(“2 + 2 = “, 2 + 2);

Aritmética:

Suma: suma entre dos valores numéricos que resulta en un valor numérico
operando **+** operando

Resta: resta entre dos valores numéricos que resulta en un valor numérico
operando **-** operando

Multiplicación: multiplicación entre dos valores numéricos que resulta en un valor numérico
operando ***** operando

División: división entre dos valores numéricos que resulta en un valor numérico
operando **/** operando

Operador comparativo: comparación entre dos valores numéricos que resulta en un valor booleano

Mayor que	>
Menor que	<
Mayor o igual	>=
Menor o igual	<=
No igual que	<>
Igual que	==

Or: operación entre dos valores booleanos que resulta en un valor booleano
operando **|** operando

And: operación entre dos valores booleanos que resulta en un valor booleano
operando **&** operando

Paréntesis: encapsula una o muchas operaciones
(expresión **)**

If/Else:

If basico:

si (expresión booleana**) entonces {**
    <estatutos>
}

Ej.

```
si (a < b) entonces {
    escribe("a es menor a b");
}
```

If else:

si (expresión booleana**) entonces {**
    <estatutos>
} sino {
    <estatutos>
}

Ej.

```
si (a < b) entonces {
        escribe("a es menor a b");
} sino {
        escribe("a no es menor a b");
}
```

Loops:

Loop while:

mientras (expresión booleana**) hacer {**
    <estatutos>
}

Ej.

```
mientras (x < y & y < z) hacer {
        x = x + 2;
        y = y + 1;
}
```

Loop for:

desde id **=** (exp) **hasta** (exp) **hacer {**
    <estatutos>
}

Ej.

```
desde i = 0 hasta (k + 1) hacer {
        a[i] = i * 2;
}
```

VIII. Video Demostración

Se encuentran en el folder de VideoDemos