Final documentation

MAURCO

06/06/2022

Marco Brown Cunningham A00822215

Mauricio Martinez Toledo A01173441

Index

Project Documentation and Description

a. Project Description

a.1. Scope and Purpose

The scope of the MAURCO program is to create a compiler and a virtual machine based on procedural programming. The semantics are quite similar to those in the C language. Functions, expressions, types, and multidimensional arrays are supported.
The purpose of this project is to transmit programming fundamentals, having the fundamental structure of good programming and structure will teach to new programmers the basics.

a.2. Requirement analysis and description of test cases

- The compiler must accept three types of data: Int, Float, String.
- The compiler must accept arrays of one and two dimensions.
- The compiler must process functions with parameters and accept recursion.
- Functions in the programming language must be Int, Float, or Void.
- The compiler must process expressions and assign values.
- The compiler must be able to receive information from the command line and print the results if needed.
- The compiler must accept a conditional loop.
- The MAURCO compiler must generate intermediate code represented by quadruples.
- The MAURCO virtual machine must be able to interpret the quadruples generated by the compiler and execute them.
- The virtual machine must manage the memory in the global scope and current scope, generating different memories depending on the instruction pointer.
- Data structures used during the project must be efficient for the correct functioning of the MAURCO project.
- The compiler must give information of different error types in the case they happened. Syntax, lexical, execution and semantic.

During the testing phase of MAURCO we assembled different tests to apply all the requirements of the compiler. First, variable declaration and arrays with their data types. Second void functions and return type functions. Cycles and conditionals were also tested to make iterative processes. At the end, function calling and recursion were tested and ended with proper results.

a.3. Description of the development of the project

During the development of the MAURCO project we worked on different repositories. In the first 3 weeks we used to work on repl.it due to the lack of knowledge and the need for real time collaboration. After that we moved the project to github, the list of commits begins in April. The commits can be found in our repository and the repl.it. Also every week we have delivered advancements of the project on Canvas with a small description of the tasks done during that time.

Week 1	Lexical and Syntax Analysis	During the first week of the project we generated the initial grammar based on the homework "LittleDuck". First we adapted the different diagrams that we had to our new grammar. Then these were converted by hand so they could be read by PLY. This week we did not have a GitHub repository but we worked on repl.it. Comments: -Declaration of classes, variables and functions -Read and write were declared -Basic types were declared and their expressions -Arrays and matrices were defined
Week 2	Basic Variable Semantics	We created a file named DirVar that makes the compilation of our program. We first instantiated it in week 2. Comments: -Grammar conflicts were corrected -The directory had variables and functions
Week 3	Basic Expressions Semantics	During this time, the generation and saving of quadruples was implemented. We added neural points to certain parts of the grammar so that they could be visible to us. We chose not to use Polish Vector due to its complexity. Comments: - The Github repository was created and now used as the main repository -Linear expressions were created and represented.
Week 4	Intermediate Code (Cycles)	In week 4 some linear expressions were not in the correct order, we spent some time fixing them. Quadruple generation for *while*, *if else* and *from to* was also implemented. Comments: - *From to* was separated from the program, it had conflicts with some go to statements - Some counter generations had an incorrect value by (-1 or 1) because of the instruction pointer or current quadruple.
Week 5	Intermediate Code (Functions)	This week was very difficult for us. We could not make the same amount of progress as previous weeks because of personal issues. We mostly fixed some bugs that happened

		in the intermediate code generation.
Week 6	Memory in Virtual Machine	We were able to create all function code generation, and also start the virtual machine.
Week 7	Array intermediate code	In week 7 arrays were finally finished. Temporal addresses were not in use at this time but the virtual machine was being instantiated. Comments: Arrays and matrices were not created in the same way as class. Neural points varied and we knew that arrays only had two elements.
Week 8	Documentation	During this time we were able to get addresses working correctly. No more values or strings were displayed on the terminal. Comments: -Even though we finally had something working, trying expressions we realized that we had a lot of problems in some queues.
Week 9	Final Delivery	This week was used for making everything else. Comments: -Queues were fixed -Params was fixed -Virtual Machine was completed -Documentation was completed -Recursion was fixed -Local and Global memory was separated -MAURCO was completed

Mauricio Martínez Toledo:
This was one of the most demanding projects I've had in my student life. Compiler design was, without doubt, one of the most challenging classes during university. This project helped me comprehend the amount of work that other people have done to implement some of the functionalities we take for granted when coding. This project tested my programming and debugging skills, patience, and resilience because several times we discovered that some of our implementations were wrong or that our "solutions" just provoked other bugs. In conclusion, this was both a challenging and satisfying experience. I'm proud of what we created as a team having no previous experience designing compilers and I'm sure that I'll remember this class for many years ahead.

Marco Brown C:

During my time in the compilers class (A full year) I had an amazing experience. Even though I had to take it twice, I can say it is one of my favorite classes of all of my career. It is truly a challenge, for the mind, the body and the soul. I felt like everytime something did not work I had to solve it, the complexity was so big that we had to check every part of the code just to know where the problem was. All of the knowledge we acquired during the years in ITC was extremely useful here. During this time in compilers I have understood how much I am in love with my career, and even though this project has some missing parts, I am extremely proud of it. Also, I am grateful to my teammate because we worked very well together and never allowed me to fall back on this project. Having taken this course twice (completely) I now understand that last semester I would have never finished my Individual project, it was a result of my lack of responsibility and overconfidence. Again, this was an amazing project to work on, I have learned a lot from this course and my teachers and I am sure that other classes (algorithms, networks, databases) will stay with me for a long time.

b. Language Description

b.1. Name of the language

The name “MAURCO” was chosen because it is a combination of the name Marco and Mauricio. It is similar in some ways to C but changes have been made so that MAURCO is unique.

b.2. Generic Description of the characteristics of the language

MAURCO is a procedural programming language.
Procedural programming offers the basic functionality with calls to modules. MAURCO, similarly to C, works as an extension of imperative programming . The procedures create different jumps between the program and carry out computational steps. Also MAURCO can compute all necessary operations for variables, functions and arithmetic expressions. Global variables are first defined, then functions and at the end there is a main function with all the operations and function calls.

b.3. List of errors that can happen during compilation and execution

Compile time errors

Name	**Description**
Lexical Error	It is raised when the input file contains a character that is not recognized by the lexer.
Syntax Error	It is raised when there is a statement that does not adhere to the defined syntactic structure of the MAURCO language.
Name Error	It is raised when trying to use a variable that is not declared in either local or global scope.
Dimension Error	It is raised when trying to access an array with the wrong number of

	dimensions.

Runtime errors

Name	**Description**
Index Error	It is raised when trying to access an index out of an array bounds.
Input Error	It is raised when the value given by a user in CLI is not compatible with the type of variable that the value is being assigned to.
Value Error	It is raised when there is division by zero.

c. Compiler Description

c.1. Computer setup, language and special libraries used in the development of the project.

A Windows computer with Python 3.6 or superior is needed for the project.
For the parsing process PLY 4.0 was used, it can be found and downloaded in (https://github.com/dabeaz/ply).
The only special library used in the project is functools, its *reduce* function was used for calculating the array length.

c.2 Description of the Lexical Analysis

Tokens used in MAURCO:

Token name	**Character or reserved word**
PROGRAM	program
MAIN	main
SEP_SEMICOLON	;

SEP_LBRACE	{
SEP_RBRACE	}
SEP_COMMA	,
SEP_LBRACKET	[
SEP_RBRACKET	]
SEP_LPAREN	(
SEP_RPAREN	)
OP_PLUS	+
OP_MINUS	-
OP_DIV	/
OP_MULT	*
OP_EQUAL	==
OP_NOTEQ	!=
OP_ASSIGN	=
OP_LT	<
OP_GT	>
REL_AND	&
REL_OR	|
INT	int
FLOAT	float
END	end
IF	if
THEN	then
ELSE	else
WHILE	while
DO	do
RETURN	return
VARS	vars
PRINT	print

INPUT	input

Regular definitions

Token	**Definition**
ID	`'[a-zA-Z_][a-zA-Z0-9_]*'`
newline	`'\n+'`
CTE_CHAR	`'\".*\"'`
CTE_FLOAT	`'\d*\.\d+'`
CTE_INT	`'\d+'`

c.3. Description of the Syntax Analysis

The language uses the following formal grammar where the rules are lowercase and the tokens or terminals are in UPPERCASE. This grammar has been modified to avoid ambiguity due to left recursion or left factoring.

```
programa : PROGRAM ID SEP_SEMICOLON provarsaux

provarsaux : vars profunctions
             | profunctions

profunctions : functions profunctions
               | principal END

principal : MAIN SEP_LPAREN SEP_RPAREN bloque

bloque : SEP_LBRACE estatuto bloqueaux

bloqueaux : estatuto bloqueaux
            | SEP_RBRACE

functions : typefun

typefun : nvfuntipid novoidnext
            | vfuntipid voidnext

nvfuntipid : INT FUNCTION ID
```

```
               | FLOAT FUNCTION ID
               | CHAR FUNCTION ID

vfuntipid : VOID FUNCTION ID

novoidnext : SEP_LPAREN paramsfun SEP_RPAREN varsfun SEP_LBRACE estfun
nvaux

nvaux : RETURN SEP_LPAREN hyper_exp SEP_RPAREN SEP_SEMICOLON SEP_RBRACE

voidnext : SEP_LPAREN paramsfun SEP_RPAREN voidvars

paramsfun : paramsfuncreate
              | ε

paramsfuncreate : typeparamfun ID paramsauxfun

paramsauxfun : paramsauxfuncreate
                 | ε

paramsauxfuncreate : SEP_COMMA typeparamfun ID paramsauxfun

typeparamfun : INT
                  | FLOAT
                  | CHAR

varsfun : varsnfun
               | ε

varsnfun : VARS varsauxfun

varsauxfun : idvarsfun
            | ε

idvarsfun : typefunp lista_ids varsauxfun

typefunp : INT | FLOAT

voidvars : varsfun SEP_LBRACE estfun RETURN SEP_SEMICOLON SEP_RBRACE

estfun : estatuto estfun
              | ε

vars : VARS varsaux

varsaux : type lista_ids varsaux
            | ε
```

```
lista_ids : ID listaidaux
listaidaux : decarr decaraux
               | decaraux

decaraux : SEP_COMMA lista_ids
             | SEP_SEMICOLON

decarr : SEP_LBRACKET CTE_INT SEP_RBRACKET auxdec

auxdec : SEP_LBRACKET CTE_INT SEP_RBRACKET
           | ε

accarraux : SEP_LBRACKET expresion SEP_RBRACKET
              | ε

type : INT
         | FLOAT

estatuto : asignacion
             | decision
             | escritura
             | llamadavoid
             | repeticion
             | lectura

asignacion : valasigaux OP_ASSIGN hyper_exp SEP_SEMICOLON

valasigaux : ID valasign_aux2

valasign_aux2 : acceso_array

args : hyper_exp argsaux
| ε

argsaux : SEP_COMMA hyper_exp argsaux
| ε

decision : IF SEP_LPAREN hyper_exp SEP_RPAREN THEN bloque decisionaux

decisionaux : ELSE bloque
                | ε

escritura : PRINT SEP_LPAREN escrituraaux

escrituraaux : letrero escaux2
                 | hyper_exp escaux2
```

```
escaux2 : SEP_COMMA escrituraaux
            | SEP_RPAREN SEP_SEMICOLON

letrero : CTE_CHAR

llamadavoid : ID SEP_LPAREN args SEP_RPAREN SEP_SEMICOLON

repeticion : repeticioncondicional

repeticioncondicional : WHILE SEP_LPAREN hyper_exp SEP_RPAREN DO bloque

lectura : INPUT SEP_LPAREN ID lectaux  SEP_RPAREN SEP_SEMICOLON

lectaux : SEP_COMMA ID lectaux
            | ε

hyper_exp : and_exp hyper_aux

hyper_aux : REL_OR hyper_exp
              | ε

and_exp : expresion andexpaux

andexpaux : REL_AND and_exp
              | ε
expresion : exp expresionaux

expresionaux : evaluators exp
               | ε

exp : termino expaux

expaux : gensumres expaux
           | ε

gensumres : OP_PLUS termino
              | OP_MINUS termino

termino : factor terminoaux

terminoaux :  genmuldiv terminoaux
               |  ε

genmuldiv : OP_MULT factor
              | OP_DIV factor
factor : SEP_LPAREN hyper_exp SEP_RPAREN
```

```
            | cteidcall
            | separadorarrfunc

cteidcall : CTE_INT
              | CTE_FLOAT

separadorarrfunc : asignacion_compleja
                     | cteidcall_atributo_metodo

asignacion_compleja : usfunc asignacion_funcion

usfunc: ID

asignacion_funcion : SEP_LPAREN args SEP_RPAREN

cteidcall_atributo_metodo : ID var_id_aux

var_id_aux : aux_accesoarray
             | ε

aux_accesoarray : acceso_array
                  | cteidcall_am_aux

cteidcall_am_aux : asignacion_funcion
                 | ε

acceso_array : mataccarraux
           | arraccarraux

mataccarraux : SEP_LBRACKET  hyper_exp SEP_RBRACKET SEP_LBRACKET
hyper_exp SEP_RBRACKET

arraccarraux : SEP_LBRACKET hyper_exp SEP_RBRACKET
```

c.4. Description of the Intermediate Code Generation and Semantic Analysis

Intermediate code operations

MAURCO operation codes have 1 to 4 elements and follow this format:
`Name, LeftOperand, RightOperand, Result`

Examples with different number of elements:
```
END, , ,
GOTO, , , 1
ERA, funcion, , ,
=, 1000, , 5000
+, 1000, 1001, 1002
```

It uses addresses with the following ranges:
Global Int = [5000,8000)
Global Float = [8000, 11000)

Locals initialization
Local Int = [11000, 13000)
Local Float = [13000, 15000)

Temporal global initialization
Temporal Int = [15000, 17000)
Temporal float = 17000

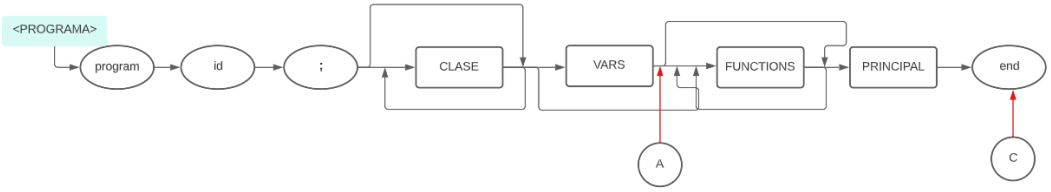
Constant initialization
Constant Int = [23000, 24000)
Constant Float = [24000 , 25,000]

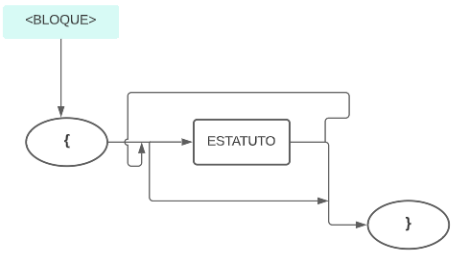
#Temporal pointer initialization
Temporal Pointer = [69000, 70000]

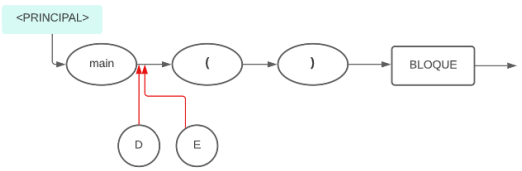
MAURCO uses the following operation codes when generating intermediate code:

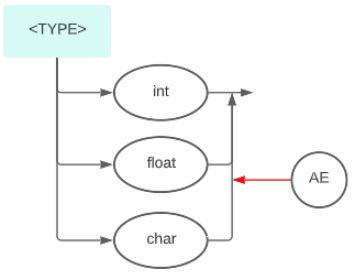
Name	Description
GOTO	It is an unconditional GOTO, it moves the instruction pointer in the virtual machine to the indicated quadruple in the *Result* element.
Arithmetic, logical and comparison operators +, -, *, / |, & ==, !=, <, >	They evaluate left and right operand using the first element in the quadruple, which contains the correspondent operation, and store the result in the *Result* element.
GOTOF	It is a conditional GOTO, it moves the instruction pointer in the virtual machine to the indicated quadruple in the *Result* element only when the *LeftOperand* is False.
PRINT	It shows in the console the content of an address, the content of a pointer address or a string.
RETURN	It indicates the variable that contains the value that a function returns. It is not generated by void functions.
ENDFUNC	It indicates the point where a function's instructions end.
GOSUB	It indicates that a subroutine is called. It moves the instruction pointer to the function it has in *leftOperand*.
VER	It verifies that the *leftOperand* value is between *rightOperand* and *Result* values. It raises an error is the condition is not met.
INPUT	It indicates that the user will input a value to some variable.
ERA	It indicates that there is a function call that requires its own memory.
PARAM	It indicates the value of the parameter in the *leftOperand* address, and assigns it to the parameter in the number of
=	It assigns the value of *leftOperand* address to the address in *Result*.

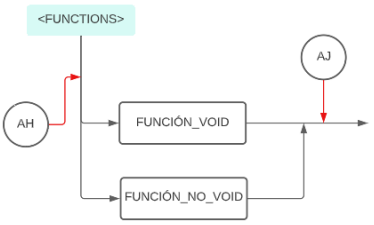
The neural points of our compiler can be seen in the next pages. Also you can find it HERE. The explanation of our neural points is at the end of the section.

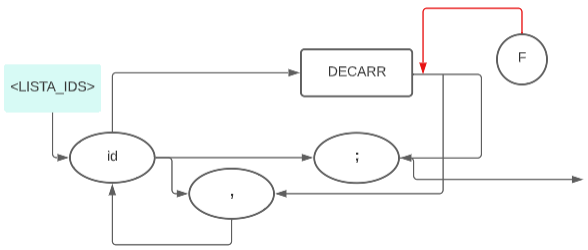

<PROGRAMA>
program
id
;
CLASE
VARS
FUNCTIONS
PRINCIPAL
end
A
C

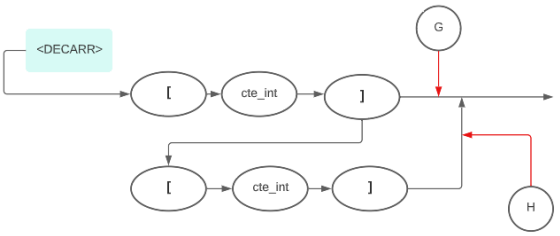

<BLOQUE>
{
ESTATUTO
}

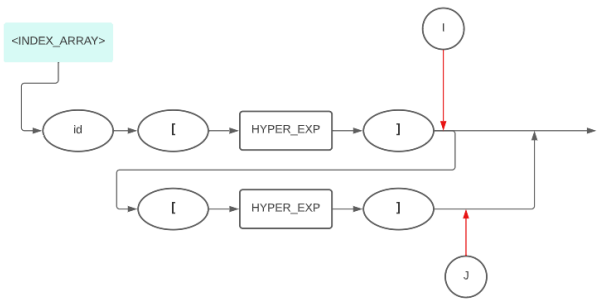

<PRINCIPAL>
main
(
)
BLOQUE
D
E

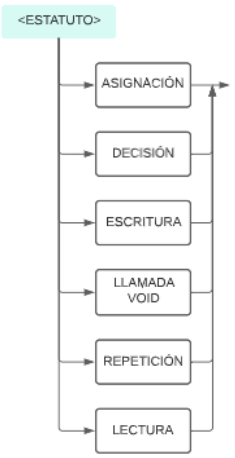

<TYPE>
int
float
char
AE

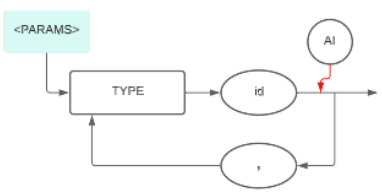

<FUNCTIONS>
AH
FUNCIÓN_VOID
FUNCIÓN_NO_VOID
AJ

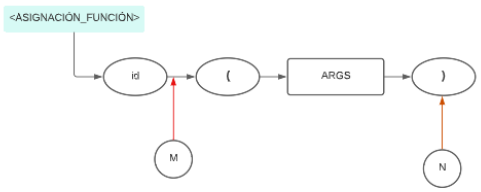

<LISTA_IDS>
id
,
;
DECARR
F

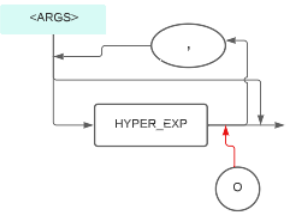

<DECARR>
[
cte_int
]
[
cte_int
]
G
H

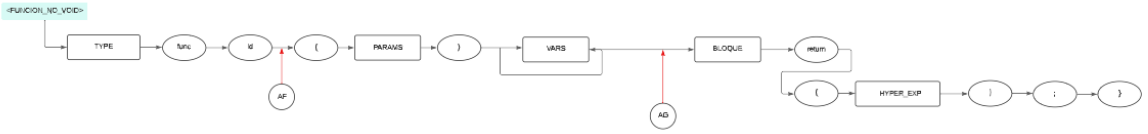

<INDEX_ARRAY>
id
[
HYPER_EXP
]
[
HYPER_EXP
]
I
J

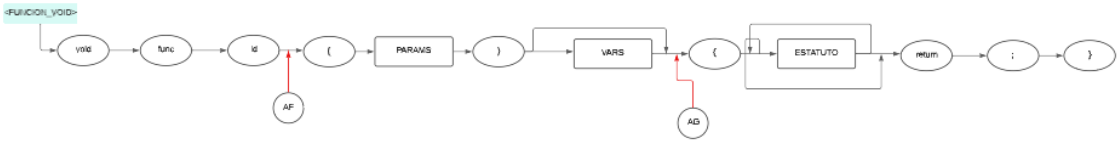

<ESTATUTO>
ASIGNACIÓN
DECISIÓN
ESCRITURA
LLAMADA VOID
REPETICIÓN
LECTURA

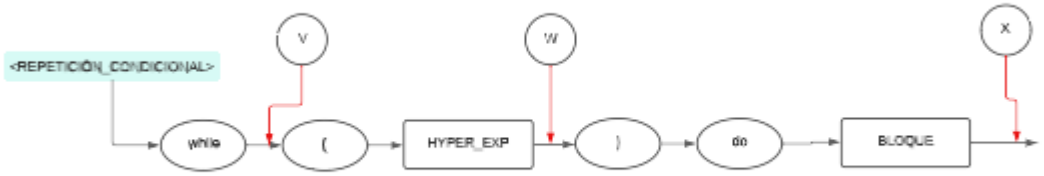

<PARAMS>
TYPE
id
,
AI

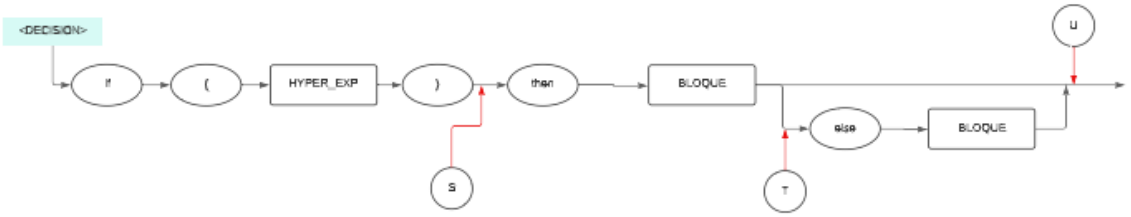

<ASIGNACIÓN_FUNCIÓN>
id
(
ARGS
)
M
N

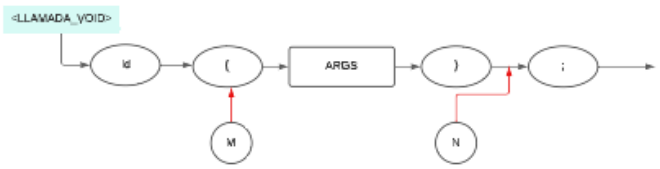

<ARGS>
,
HYPER_EXP
O

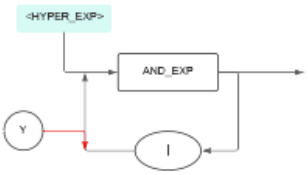

<FUNCION_NO_VOID>
TYPE
func
id
(
PARAMS
)
VARS
BLOQUE
return
(
HYPER_EXP
)
;
}
AF
AG

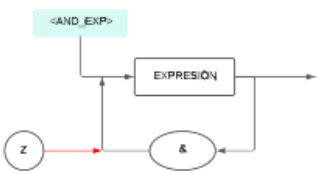

<FUNCION_VOID>
void
func
id
(
PARAMS
)
VARS
{
ESTATUTO
return
;
}
AF
AG

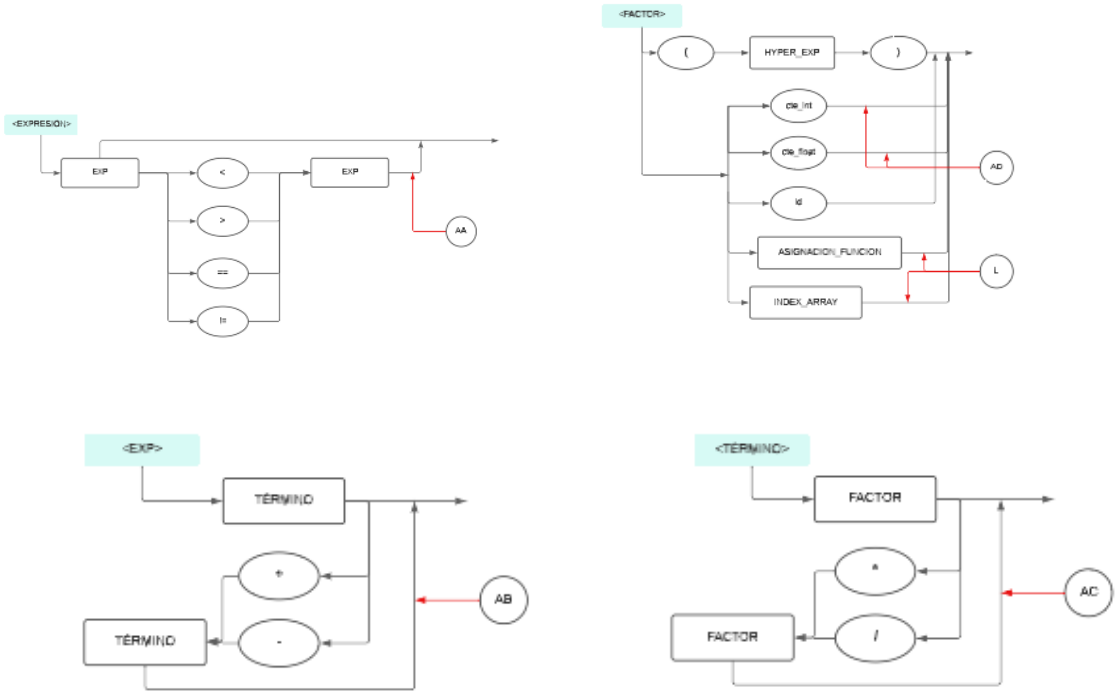

<REPETICIÓN_CONDICIONAL>
while
(
HYPER_EXP
)
do
BLOQUE
V
W
X


<DECISIÓN>
if
(
HYPER_EXP
)
then
BLOQUE
else
BLOQUE
S
T
U


<LLAMADA_VOID>
id
(
ARGS
)
;
M
N


<HYPER_EXP>
AND_EXP
Y
|


<AND_EXP>
EXPRESIÓN
Z
&



Here is a summary of the Neural Points:

- A = Get from id stackVariables and push to global variables.
- B = Push type and ID to StackVariables.
- C = End quadruples.
- D = Push main to currFunction, and add it to global variables.
- E = Add in the first quadruple the main starting point.
- F = Put id and the length (if is array) in a queue
- G = Put value of array length.
- H = Put value of matrix length.
- I = Push hyper exp to values of arrExp.
- J = Push hyper exp to values of arrExp.
- K = Add operator, create assignment quadruple.
- L = Get the type of the id.
  Add id and type to the stack.
  Gets the address depending on global or local variables.
  If it is not an array the value is changed to the id address.
  If it is an array the operand stack is popped to get the address.
  If it has one dimension it verifies in quad it with arrExp and then it sums the base direction of the ID in quads.
  If it has two dimensions the first dimension of the address is verified in quad with the arrExp on the first dimension.
  The offset is multiplied by the second dimension of the matrix and added to a quad.
  The second dimension is verified in quad and sums the first operation of the matrix with the second offset.
  Finally the matrix gets its own base direction and adds it in quad.
- M = Verify that the global variable exists, if it exists create the ERA of the function.

- N = Get the parameters of the function.
  If the parameters are the same length create a quad GOSUB of the function.
  If the function is not VOID : get the address of the global variable of the function and create the quads that assign the return value of the function.
- O = validate the current parameter with the parameters type in the function and add it to a PARAM quad.
- P = create quad of input.
- Q = create quad of print of expressions.
- R = create quad of print of chars.
- S = create quad GOTOF with no value.
- T = create quad GOTO with no value.
- fill GOTOF with the current counter of quads.
- U = Fill GOTO quad with current counter of quads to skip else.
- V = append to jump stack.
- W = create GOTOF with Operands and blank space.
- X = Create GOTO that goes to the conditional of the while.
- fill the GOTOF with the next quad counter.
- Y = Create quad of or.
- Z = Create quad of and.
- AA = Create quad of comparison (evaluators).
- AB = Create quad of SUM/SUB.
- AC = Create quad of MUL/DIV.
- AD = Add to TypeStack and to Operand Stack the address of the value.
- AE = Get the type.
- AF = Add function to Functions Directory.
- If not VOID add Function to Global Variable.
- AG = Add variables and parameters from variableStack to local Function with addresses, and add the params to the function. Add to the function the quads starting point.
- AH = reset counters of local variables and reset counter of temporal variables.
- AI = Add to variableStack type and ID of parameters.
- AJ = Add to the quads the endFUNC of the function.

Table of compatibility among types and operators

In the assignment operator the left operand always keeps its type and if possible it casts the assigned value to the right one.

Left operand	Right operand	+	-	*	/	<	>	&	|	==	!=	=
int	int	int	int	int	float	int	int	int	int	int	int	✔ int
int	float	float	float	float	float	int	int	int	int	int	int	✖
int	char	err	err	err	err	err	err	err	err	err	err	✖
float	float	float	float	float	float	int	int	int	int	int	int	✔ float
float	int	float	float	float	float	int	int	int	int	int	int	✔ float
float	char	err	err	err	err	err	err	err	err	err	err	✖
char	char	err	err	err	err	err	err	err	err	err	int	✔
char	int	err	err	err	err	err	err	err	err	err	err	✖
char	float	err	err	err	err	err	err	err	err	err	err	✖

c.5. Detailed description of the memory administration process during compilation

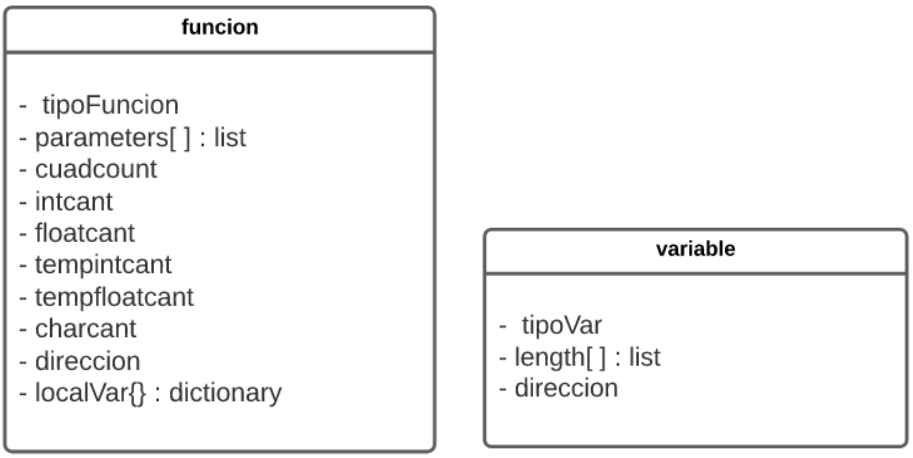
During compile time there are several important data structures. They are listed below:

Name	Description
variable	It is a class for storing variable properties. It has the type, length (in case it is an array), and assigned address.
funcion	It is a class for storing function properties. It contains the type, a list with the type of its parameters, the number of variables needed for its local memory by type, an assigned address for non-void functions, and a dictionary with its variable objects.
dirGlobalVar	It is a dictionary containing all the global variable objects. It allows fast and easy search using a global variable name as a key.
dirFunciones	It is a dictionary containing all the declared function objects (that also includes the `main`

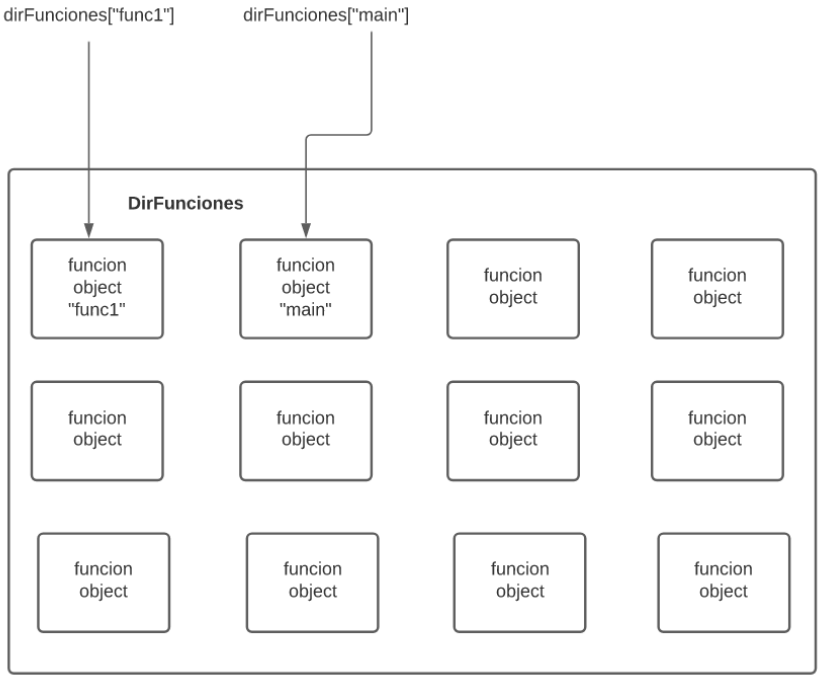
	function). It allows fast and easy search using a function name as a key.

semanticCube	It is a dictionary of dictionaries. It can be indexed using strings. It contains the compatibility among types and operations and returns a type or an error string.
	Example:
	`print(semantic['int']['+']['int'])`
	returns an “int” string
	meanwhile
	`print(semantic['int']['=']['float'])`
	returns an “error” string

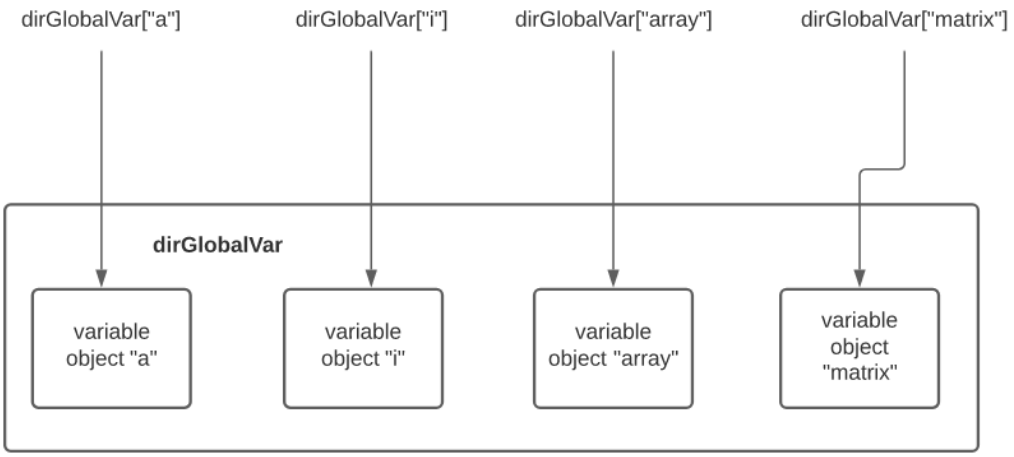
pOperandos	It contains all the operands being used by the expressions.
pOperadores	It contains all operators being used by the expressions.
pTipos	It contains the type of each one of the operands being used by the expressions.
psaltos	It is used to represent where there is a non linear statement. Elements are pushed and popped from here for filling GOTO quads after finalizing the parsing of some nonlinear statements, such as *if else* and *while*.
cuads	It is a list of tuples that contains all the generated quadruples. It also contains the quad number and it is shared with the virtual machine for executing all operations.
currID	It is a queue that stores variable names in multiple declaration such as `int x, y, z;`
currTypeID	It is a queue that stores variable types and id in multiple declaration such as `int x, y, z;`
dirConstantes	It is a dictionary that uses a constant, either float or int, as a key and saves virtual addresses as values.



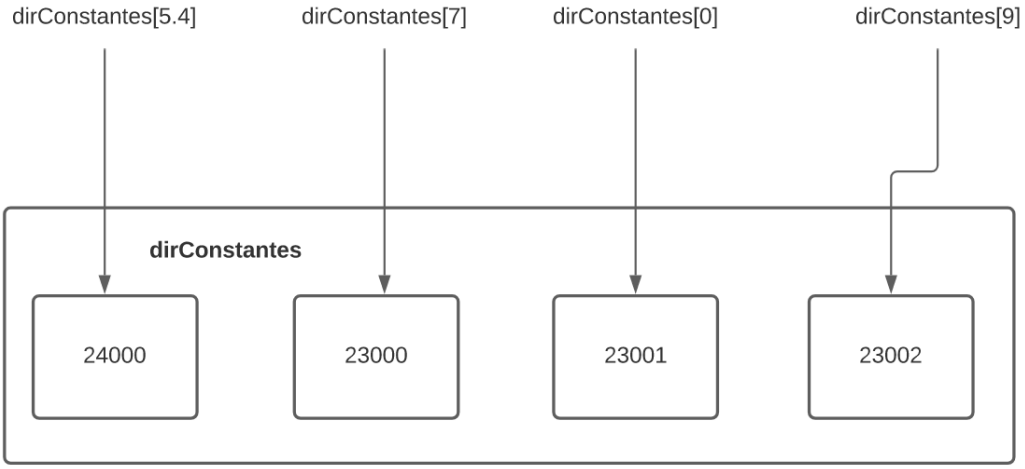
On the left: class diagram of a ‘funcion’ object; on the right: class diagram of a ‘variable’ object.



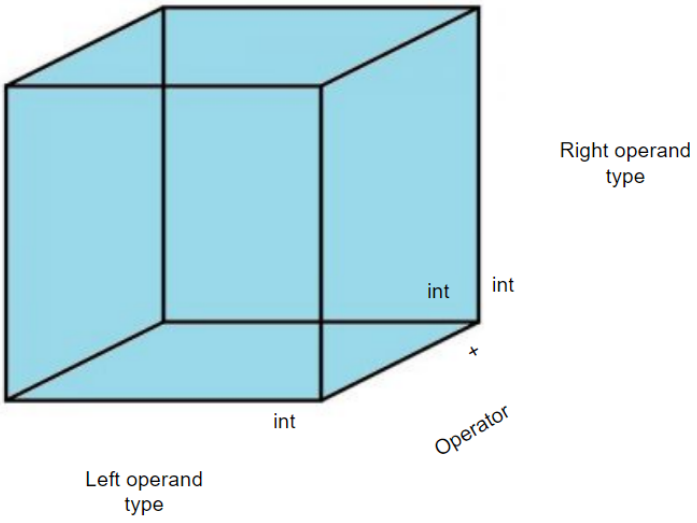
Representation of a hashmap of the functions directory of a MAURCO file.



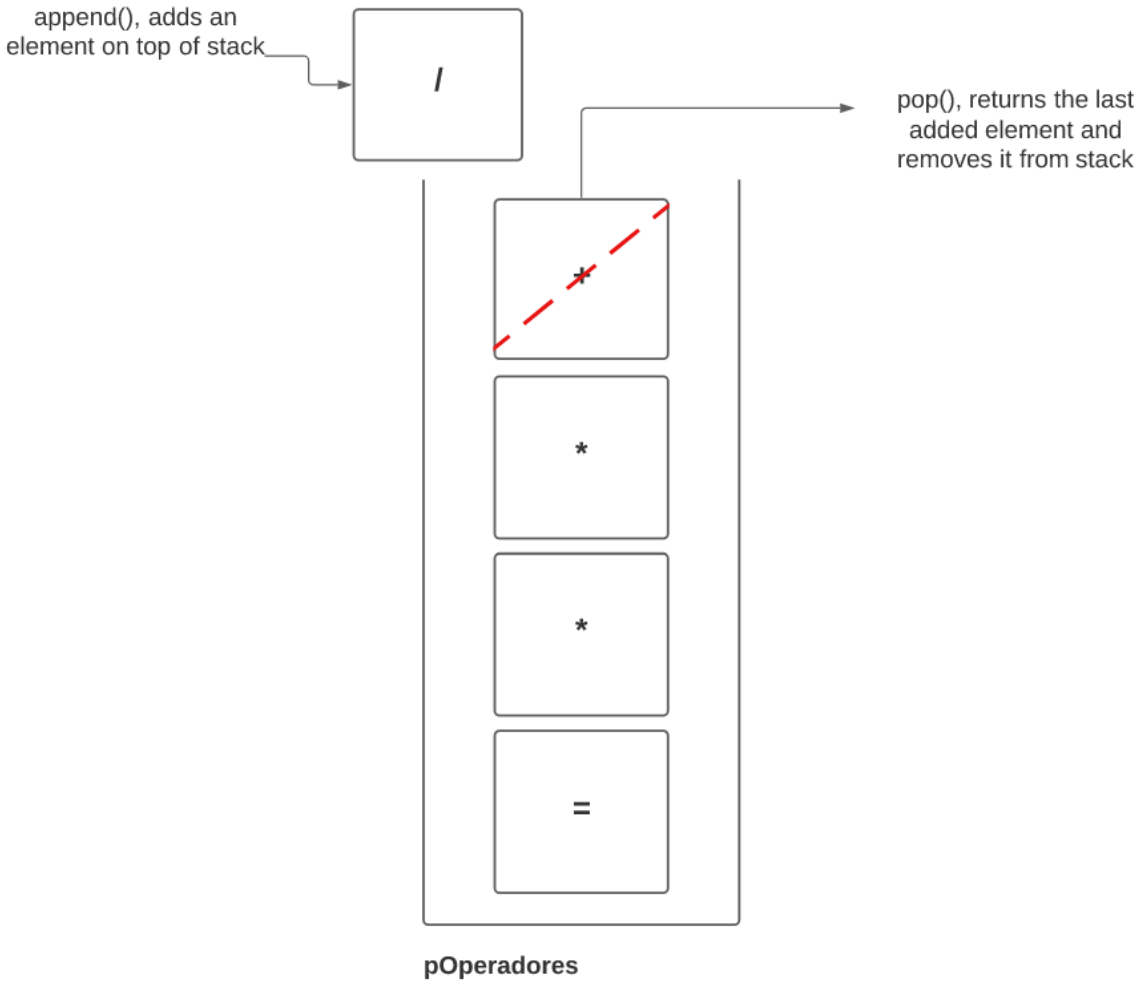
Representation of a hashmap of the global variable directory.



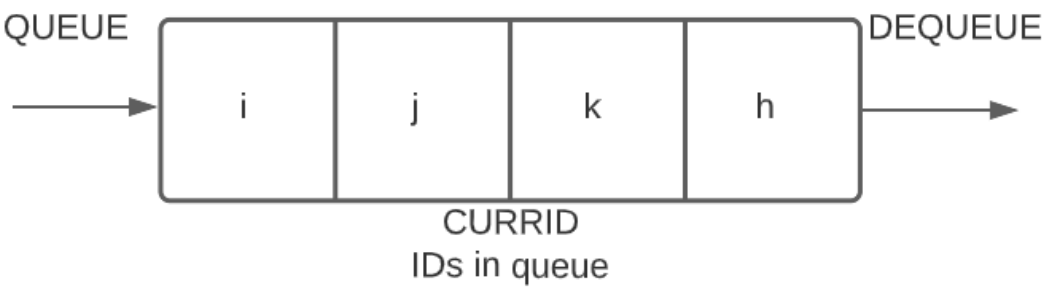
Representation of a hashmap of the constants directory.

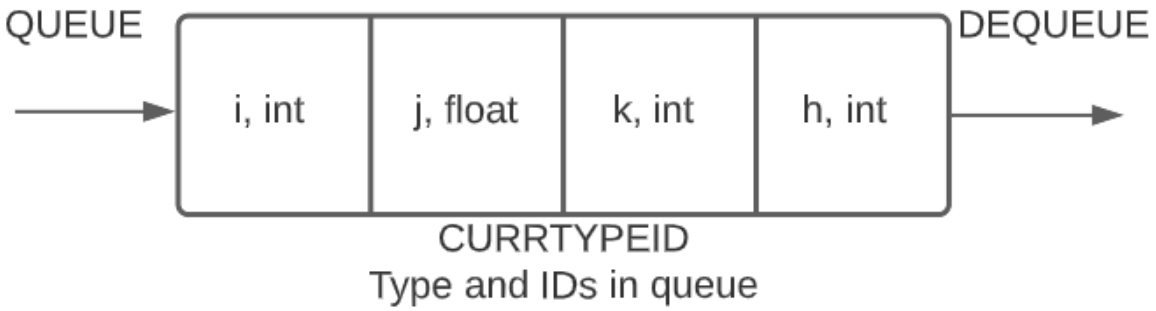


Simplified representation of the semantic cube where a dimension represents the left operand type, the second dimension represents the operator and the third one represents the right operand type. The content of a cell represents the result of a certain operation.

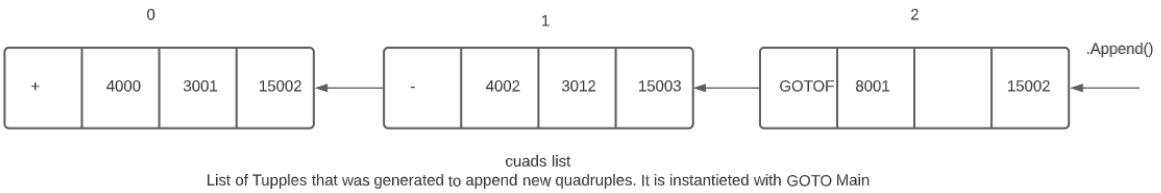


Representation of a pOperadores stack.
pOperandos, psaltos and pTipos work similarly to pOperadores; but they use their respective content (psaltos uses integers, pOperandos uses addresses, and pTipos uses strings).





Queues were used to simulate the correct order of assignment during compilation due to some order difference in syntax.



The quads list is instantiated with GOTO Main, after that each new operation is appended at the end of the list. The tuples are always of size 4 with (operation, left operand, right operand and result).

ArrayLength was created to keep track of the current size of the array in compilation, once it is assigned to a variable it is reseted to nothing.

Dictionaries were the ideal data structure for storing functions and variables because the Python 3.6 implementation uses a hashmap to find the desired object using the key. Also, Python dictionaries are iterable which means that the find and write operations on them are quite efficient and easy to program.

pOperandos, pOperadores, pTipos and pSaltos are lists that simulate the functionality of a stack. Two basic operations are performed:

- append(), which adds an element on top.
- pop(), which gets the last element from the list and erases it from the list.

A stack is used due to having quick add and get operations and being of LIFO approach.

d. Virtual Machine Description

d.1. Computer setup, language and special libraries used(different than compiler)

The virtual machine can be found in a file named `vm.py`.
For the virtual machine, a Windows computer with Python 3.6 or superior is needed. There is no need for any additional libraries.

d.2. Detailed description of the process of the memory administration in execution

The virtual machine uses two types of memory: **local** and **global**.

It uses addresses with the following ranges:
Global Int = [5000,8000)
Global Float = [8000, 11000)

Locals initialization
Local Int = [11000, 13000)
Local Float = [13000, 15000)

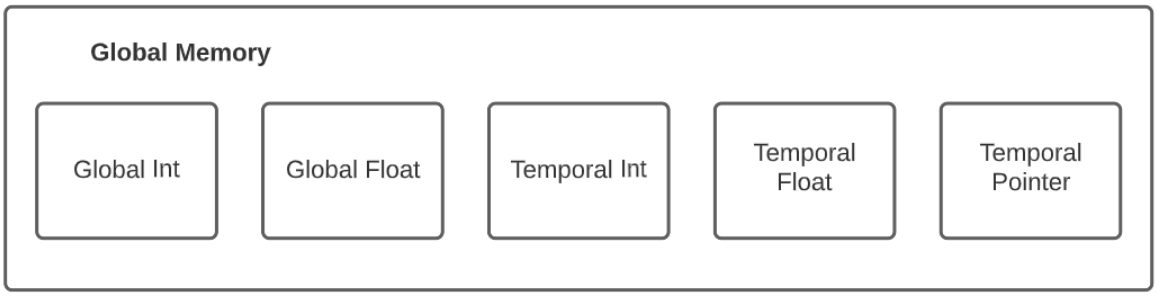
Temporal global initialization
Temporal Int = [15000, 17000)
Temporal float = 17000

Constant initialization
Constant Int = [23000, 24000)
Constant Float = [24000 , 25,000]

#Temporal pointer initialization
Temporal Pointer = [69000, 70000]

The global memory consists of a list of 5 lists.

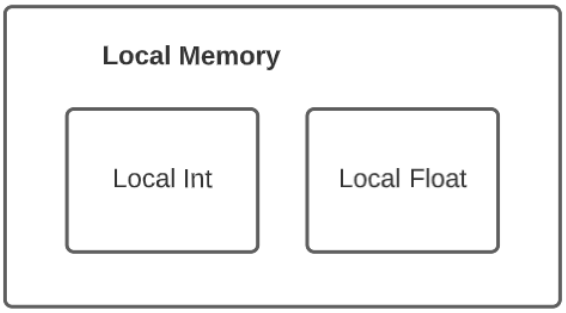
- The first list inside the global memory contains all the global variables of type int.
- The second list inside the global memory contains all the global variables of type float.
- The third list contains all the temporal variables of type int, either local or global.
- The fourth list contains all the temporal variables of type int, either local or global.
- The fifth list contains temporal variables of type pointer (they contain an address instead of a value).



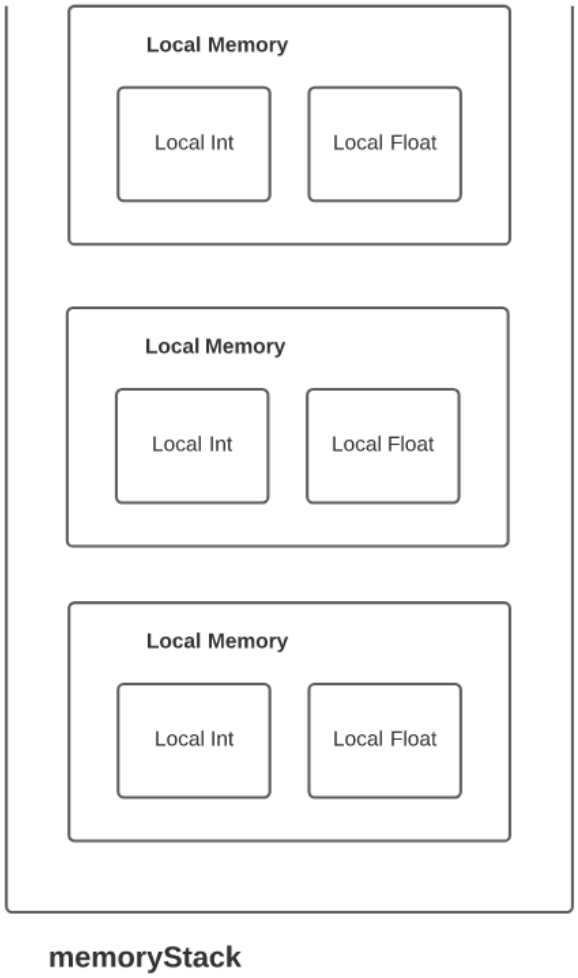
For the local memory the virtual machine uses two main data structures: a **stack** and a **list of lists**.
The list of lists contains the local memory for each function call and the stack saves it as a way to keep track of values for each call when using recursion.

A local memory object consists of 2 lists:

- The first one contains all the local variables of type int specific to that function call.
- The second one contains the float type values specific to that function call.



Finally, the memory stack is a list with stack operations that saves the memory before each recursive call. It can be represented like this, where at the end of the last recursive call the actual memory is popped so that the new memory is used for further operations.



As previously mentioned, the virtual memory receives only virtual directions that are processed by the virtual machine to obtain the actual value. The virtual machine performs two basic operations: **read** and **write**.

Both read and write depend on the `getTypeScopeByAddress(num)` function for it translates the address number to a string that indicates the type and scope of the variable or constant.

```python
def getTypeScopeByAddress(num):
    '''
    Devuelve un string con el tipo y scope de la variable en la dirección num
    '''

    if num >= GLOBAL_INT_START and num < GLOBAL_FLOAT_START:
        return 'gi'
    elif num >= GLOBAL_FLOAT_START and num < LOCAL_INT_START:
        return 'gf'
    elif num >= LOCAL_INT_START and num < LOCAL_FLOAT_START:
        return 'li'
    elif num >= LOCAL_FLOAT_START and num < TEMPORAL_INT_START:
        return 'lf'
    elif num >= TEMPORAL_INT_START and num < TEMPORAL_FLOAT_START:
        return 'tgi'
    elif num >= TEMPORAL_FLOAT_START and num < 19000:
        return 'tgf'
    elif num >= CONSTANT_INT_START and num < CONSTANT_FLOAT_START:
        return 'ci'
    elif num >= CONSTANT_FLOAT_START and num < 25000:
        return 'cf'
    elif num >= TEMPORAL_POINTER_START and num < 70000:
        return 'tp'
```

Image of `getTypeScopeByAddress(num)` implementation

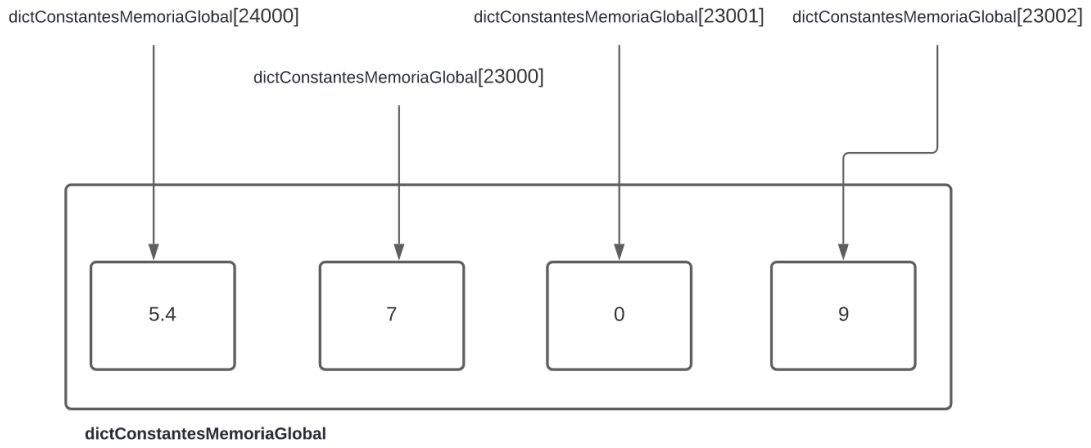
The **read** operation consists of a switch statement that depending on the string sent by `getTypeScopeByAddress(num)` gets a value from either the local or global memory. Its implementation is in the image below:

```python
def readMemory(num):
    aux = getTypeScopeByAddress(num)

    if aux == 'gi':
        return MemoriaGlobal[0][num - GLOBAL_INT_START]
    elif aux == 'gf':
        return MemoriaGlobal[1][num - GLOBAL_FLOAT_START]
    elif aux == 'li':
        if memoryStack:
            return memoryStack[-1][0][num - LOCAL_INT_START]
    elif aux == 'lf':
        if memoryStack:
            return memoryStack[-1][1][num - LOCAL_FLOAT_START]
    elif aux == 'tgi':
        return MemoriaGlobal[4][num - TEMPORAL_INT_START]
    elif aux == 'tgf':
        return MemoriaGlobal[5][num - TEMPORAL_FLOAT_START]
    elif aux == 'ci':
        return getConstant(num)
    elif aux == 'cf':
        return getConstant(num)
    elif aux == 'tp':
        return MemoriaGlobal[6][num - TEMPORAL_POINTER_START]
```

`readMemory(num)` also uses `getConstant(num)` which is a function that contains the constant dictionary with the addresses as keys, it returns a value based on the address it receives.

```
dictConstantesMemoriaGlobal = dict((v,k) for k,v in dirConstantes.items())
def getConstant(key):
    return dictConstantesMemoriaGlobal[key]
```



Graphical representation of the *dictConstantesMemoriaGlobal* dictionary.

The **write** operation consists of another switch statement that uses `getTypeScopeByAddress(num)` and depending on the string it receives assigns a value to a specific memory address, i.e. a certain index on either the actual local memory (top of the memory stack) or the global memory.

```
def writeOnMemory(value, addr):

    aux = getTypeScopeByAddress(addr)
    if aux == 'gi':
        MemoriaGlobal[0][addr - GLOBAL_INT_START] = value
    elif aux == 'gf':
        MemoriaGlobal[1][addr - GLOBAL_FLOAT_START] = value
    elif aux == 'li':
        if memoryStack:
            memoryStack[-1][0][addr - LOCAL_INT_START] = value
    elif aux == 'lf':
        if memoryStack:
            memoryStack[-1][1][addr - LOCAL_FLOAT_START] = value
    elif aux == 'tgi':
        MemoriaGlobal[4][addr - TEMPORAL_INT_START] = value
    elif aux == 'tgf':
        MemoriaGlobal[5][addr - TEMPORAL_FLOAT_START] = value
    elif aux == 'tp':
        MemoriaGlobal[6][addr - TEMPORAL_POINTER_START] = value
```

e. Evidence of the language working

e.1 Implementation of an iterative Fibonacci function

```
program fibonacci1;

int func fibonacciIterativo(int n)
vars
int aux, numAnterior, numAntesAnterior, numActual;
{
  numAnterior = 0;
  numAntesAnterior = 0;
  numActual = 1;
  while (n > 1) do{
    numAntesAnterior = numAnterior;
    numAnterior = numActual;
    numActual = numAntesAnterior + numAnterior;
    n = n - 1;
  }
  return(numActual);

}


main(){

    print("--------------------");
    print("fibonacciIterativo(10)", fibonacciIterativo(10));
    print("--------------------");

}
end
```

Intermediate code generation of the fibonacci1 program(quadruples):

```
cuad      op        v1                  v2        vf
----      --        ---                 ---       ---
0         GOTO                                    15
1         =         23000                         11002
2         =         23000                         11003
3         =         23001                         11004
4         >         11000               23001     15000
5         GOTOF     15000                         13
6         =         11002                         11003
7         =         11004                         11002
8         +         11003               11002     15001
9         =         15001                         11004
10        -         11000               23001     15002
11        =         15002                         11000
12        GOTO                                    4
13        RETURN                                  11004
14        ENDFUNC
15        PRINT                                   --------------------
16        PRINT                                   fibonacciIterativo(10)
17        ERA       fibonacciIterativo
18        PARAM     23002                         1
19        GOSUB     fibonacciIterativo
20        =         5000                          15003
21        PRINT                                   15003
22        PRINT                                   --------------------
23        END
```

Execution of fibo1 program

```
Mau@Mauricio MINGW64 ~/OneDrive/Documentos/Repos/MAURCO (funcsandvm)
$ python main.py
--------------------
fibonacciIterativo(10)
55
--------------------
```

e.2 Implementation of a recursive Fibonacci function

```
program fibonacci2;

vars{
    int i;
    float valor, y;
    char msg;
    int x;
    int arr[4], f, vector[10];
    int mat[2][3];
    int a, b, size, aux, marco;
}

int func fibonacciRecursivo(int n)
vars
int aux, aux2, aux3;
float x;
{
    if (n < 3) then{
        aux = 1;

    } else{
      aux2 = fibonacciRecursivo(n-1);
      aux3 = fibonacciRecursivo(n-2);
      aux = aux2 + aux3;
    }

    return(aux);
}


main(){

    print("--------------------");
    print("fibonacciRecursivo(10)", fibonacciRecursivo(10));
    print("--------------------");

}
end
```

Intermediate code generation of the fibonacci2 program(quadruples):

```
Mau@Mauricio MINGW64 ~/OneDrive/Documentos/Repos/MAURCO (funcsandvm)
$ python main.py



cuad      op        v1                  v2        vf
----      --        ---                 ---       ---
0         GOTO                                    21
1         <         11000               23000     15000
2         GOTOF     15000                         5
3         =         23001                         11001
4         GOTO                                    19
5         ERA       fibonacciRecursivo
6         -         11000               23001     15001
7         PARAM     15001                         1
8         GOSUB     fibonacciRecursivo
9         =         5028                          15002
10        =         15002                         11002
11        ERA       fibonacciRecursivo
12        -         11000               23002     15003
13        PARAM     15003                         1
14        GOSUB     fibonacciRecursivo
15        =         5028                          15004
16        =         15004                         11003
17        +         11002               11003     15005
18        =         15005                         11001
19        RETURN                                  11001
20        ENDFUNC
21        PRINT                                   --------------------
22        PRINT                                   fibonacciRecursivo(10)
23        ERA       fibonacciRecursivo
24        PARAM     23003                         1
25        GOSUB     fibonacciRecursivo
26        =         5028                          15006
27        PRINT                                   15006
28        PRINT                                   --------------------
29        END
```

Execution of fibo2 program

```
Mau@Mauricio MINGW64 ~/OneDrive/Documentos/Repos/MAURCO (funcsandvm)
$ python main.py
--------------------
fibonacciRecursivo(10)
55
--------------------
```

User Manual

g. Quick Reference Manual

Maurco compiles in a linear fashion, so everything that will be used must be declared before using it in the next order: program name, global variables, functions, local variables for the functions and finally the main module and its end. Some parts can be empty but they must have the keyword in order to work.

Steps for for compiling and running a MAURCO file:

1. Add a `file.m` to the `Testing files` directory.
2. Open a terminal
3. Run `python MAURCO file.m`
4. The terminal will run the `file.m` and if the syntax is correct it will be executed and the generated quadruples will be shown.

Initial declarations:

- All programs must have the keyword “program” and a name that represents the program (ID) and a semicolon after it.
  program Ejemplo;
- Global variable declaration must have first the keyword “vars” then “{“ and at the end “}”. Global variable declaration is optional and it can also be empty.
  vars{ }
- Functions declaration also must have keywords first should be the type of function that is declared “int”, “float”, “char” or “void” then the keyword “func” and then the ID followed by “(“ then the parameters (in the next section) and “)”
  void func helloWorld(params **)**
- Local variables must be declared with the keyword “vars” followed by any quantity of variables, then “{“ , any number of statements and at the end of the function “}”
  vars
  int msg; { statements returns **}**
- Parameters must be declared with the type of variable, then the name followed by a comma if necessary repeating type and ID. Parameters can be empty.
  int i, float s, int f
- Returns for functions
  Functions of type int and float must have a keyword “return” followed by “(“ a hyper expression and “)” ending with “;”. Returns can not have another function called in them due to memory calculations, only arithmetic expressions.
  int, float: **return(f);**
  void: **return;**
- Variable declarations in all sections must have a type followed by any number of id with commas between them ending with “;”.
  int i, j , p;

- Array and matrix declarations can be placed in normal variable declarations. After the id in brackets there must be the size of the array or matrix. Only one and two dimensional arrays are accepted.
  int arr[3], matrix[2][4]
- After the declaration of program name, global variables and functions, there must be a main program that executes statements. The keyword “main” followed by “(“ “)” “{“ any number of statements and “}” “end”.
  main () {
  …statements…
  } end
  Statements
  - ❖ Assign is used for assigning a value to a variable. This value can be an expression, another variable, a constant or the return of a function (float or char only). The types must match in order to work. Assignments for arrays must have the ID and in the brackets of the size there must be an index between 0 and the size of the array or matrix. Every assignment must end with “;”
    f = hw1(3);
    mat[0][1] = i * 3;
    x = 2;
  - ❖ Function calling for void type does not return a value, so it must be placed only by the function name, parameters and at the end of the function “;”. Functions with a return value can not be placed like this.
    printThis(8);
  - ❖ Input statements must also be declared in a specific way. It should be the “input” keyword next to “(“ and the name of the variable you want the user to enter. If there's more than one variable “,” must be placed between ids, at the end there must be “);”
    input(x,y,j);
  - ❖ print statements write in the terminal an specific value or set of values that the user wants. It functions similarly to input, first the “print” statement “(“ id or a char. For more prints a “,” has to be placed followed by a char or id ending with “;”.
    print(“hola”, h);
  - ❖ Conditional statements IF ELSE work based on a hyper expression. First the keyword “IF” must be placed, right after in parenthesis “(“ “)” a condition must be placed. The keyword “then” must follow with “{“ and statements between them “}”. If there is no alternative the conditional if ends with the last “}”, if there is alternative the keyword “else” must also have “{“ “}” and statements between them ending with the last “}”.
    if(i < 3) then{
    i = i+1;
    }
    else{
    a = (x+5);
    }
  - ❖ Lastly, the while cycle evaluates a condition and repeats the statements in the while cycle until it stops being true. First the “while” keyword must appear

then the condition in parenthesis “(“ condition “)” then the keyword “do” and “{“ “}” with statements between them. For the cycle to work properly the condition must change.

```
while(i<3) do {
    i = (i+1);
  }
```