Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

Diseño de Compiladores TC3048

MIR Programming Language

Andrés Carlos Barrera Basilio

A00815749

Elda Guadalupe Quiroga González
Héctor Gibrán Ceballos Cancino

June 6 2022, Monterrey Nuevo León

Table of Contents

Technical Description and Documentation of Project

Description of Project

Purpose

The purpose of the compiler design project is to create a simulacrum of a low-level compiler, abstracting the need of memory, token, and pointer handling, to teach us diverse things about the nature of creating, understanding, and working of a language. In this project, you must construct your compiler from the formal grammar to the lexical analyzer, to the parser, till the virtual machine that will compile your language. This will be done by

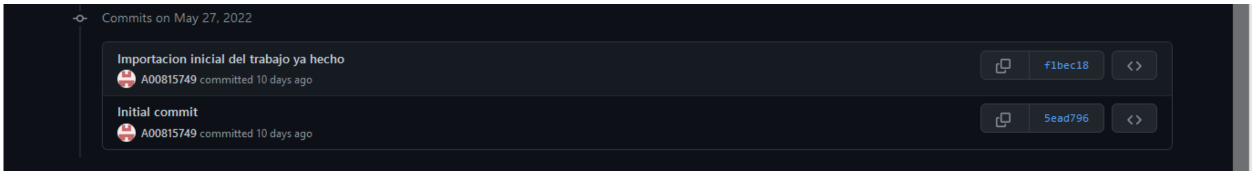
Test Cases and Project Requirements

We were provided with a document that described the specific requirements in the language, also guiding with a generic format of a program writing on that specific language. The language follows a standard structure for programming languages, giving us a specific list that we must complete for general approval:

- Variable declarations in a local and global context
- Function declarations
- Void Functions and return functions
- Handling of Int, Float and char variables
- Expression Handling
- Diverse statutes:
  - Assign, in which an identifier (from now on written as id) is assigned with a given value, in both simple variables, vectors and function calls
  - Read, in which the user can read from input, and store data in the defined id.
  - Write, in which a user can print on terminal the value of the id
  - Decision statutes, in which a user can follow the standard structure of an if-else decision, by reading expressions.
  - Looping statues, which include:
    - Conditional: While structure
    - No Conditional: for structure
- Arithmetic, Logical and Relational operations for the appropriate ids
- And special functions that accept a list of numbers

Description of General Development Process of the Project

The development process was started by the start of May, by revising the state of the past semester project and setting up the formal grammar of the language, leaving the lexer part ready for work, jointly with the Semantic Cube. This was done in the past Github folder, unfortunately leading to a late creation of a new Github project this semester.



Commits on May 27, 2022

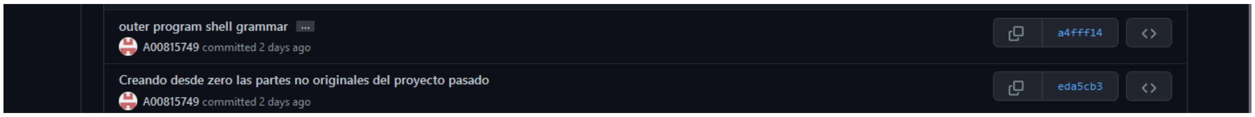
Importacion inicial del trabajo ya hecho
A00815749 committed 10 days ago
f1bec18

Initial commit
A00815749 committed 10 days ago
5ead796

5ead7960d165c122194c8f87abd5e591cb4c243f

f1bec189194824f383ca77e3e206ddfc96680e30 SHA commits from the git

Due to unforeseen circumstances at work, the entire month was too busy to commit time to the project, leading to all of the work being pushed to these last days of the semester.



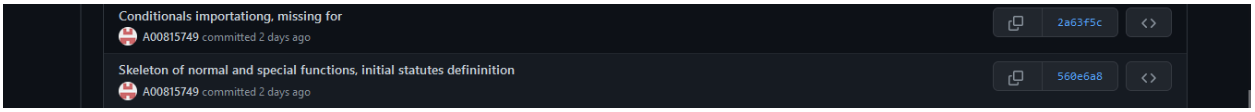
outer program shell grammar …
A00815749 committed 2 days ago
a4fff14

Creando desde zero las partes no originales del proyecto pasado
A00815749 committed 2 days ago
eda5cb3

eda5cb34b54b21332581393b03090294c301f6b8

a4fff14d12136ee8e383b5daa7afc8fb621a67de

Even so, without the other classes and past experience with these types of projects, and abandoning the idea of vectors, progress was made quickly.



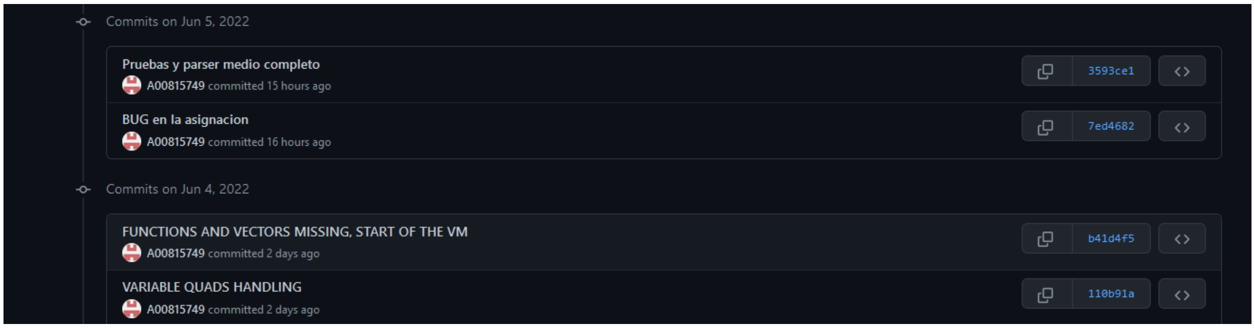
Conditionals importationg, missing for
A00815749 committed 2 days ago
2a63f5c

Skeleton of normal and special functions, initial statutes defininition
A00815749 committed 2 days ago
560e6a8

a4fff14d12136ee8e383b5daa7afc8fb621a67de

2a63f5cc488846358af7345ba4087721122d2ca7

These past days have been rough, and I should have followed the schedule of the class, even if it was the only one.



Commits on Jun 5, 2022

Pruebas y parser medio completo
A00815749 committed 15 hours ago
3593ce1

BUG en la asignacion
A00815749 committed 16 hours ago
7ed4682

Commits on Jun 4, 2022

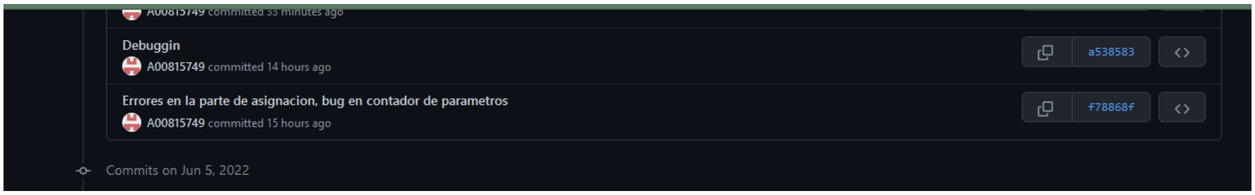
FUNCTIONS AND VECTORS MISSING, START OF THE VM
A00815749 committed 2 days ago
b41d4f5

VARIABLE QUADS HANDLING
A00815749 committed 2 days ago
110b91a

110b91a82af0f451378fd9e212785061a040550e

b41d4f5a6cecf1d5e08e7798e423cbdc0cc71a14

And one of the worries that this project considers are the rigors of presenting, as in the words of our teachers that the real companies don't pull any punches, and that we are grownups.



A00815749 committed 33 minutes ago

Debuggin
A00815749 committed 14 hours ago

Errores en la parte de asignacion, bug en contador de parametros
A00815749 committed 15 hours ago

Commits on Jun 5, 2022

f78868f40f4bb907653144c077855885e6284b5c

a538583a1faf735c8f7d8cd084422895b35403fb

In the end, I think I should follow the words of Alexander Dumas and its famous Count "Await and Hope"

Final page of Commits: https://github.com/A00815749/Compiladores2022/commits/main

Student Thoughts and Reflections

I consider this project to be again the hardest thing I had done on my academic life. One of the drawbacks I greatly consider this kind of project, is the sheer amount of time wastage following dead ends, consequence of the open nature of the work, but that is the bread and butter of coders, so I shan't say no more. Maybe an alternative would be subdividing this work as smaller open projects? Even another way would be doing isolated but gradable exercises, like developing a grammar for certain requirements, or developing the neuralgic points for a language, or editing the virtual machine so that it can process the list of Quadruples of this parser. All this can lead to learning the concepts step by step, something that can be easier for students to track, like data structure handling, grammar creating, debugging big software projects, etcetera.

As a final thought, I just want to make clear that this class is one of the most interesting classes I had taken, even if it implies a lot of work, and I express my thanks to our two teachers.

Language Description

Name of Language

The Base Language that this project was based upon was the R language, which is a statistical computing language, which is fitting, for the various special methods that were originally requested in the first proposal. If I had to seriously name my final code language, I would use the end file notation of my parser output via the quadruple list, which is .mir, which originally meant 'my r' but I think the reference to the space stations its nice, so it shall be the MIR language.

Main characteristics of the Mir language

This language has the basic functionality of a simple high level programming language, with a need for strong typing, due to an easier implementation. With the exception of the function calls and conditionals, every line must end in a semicolon, every variable must be explicitly declared with its kind, certain sections of code must remain constant when being worked upon, there are certain limits of memory that must be taken in account, only vectors of one dimension are supported, and the nifty addition of some simple statistical methods.

Error lists, in Compilation and Execution

- “FUNCION EXISTENTE REPETIDA”
- “ID DE VARIABLE Y/O PROGRAMA REPETIDA”
- "OPERACION INVALIDA, MISMATCH DE TIPOS"
- "VARIABLE DECLARADA MULTIPLES VECES"
- "MISMATCH DE TIPOS"
- "TIPO DE DATO NO ACEPTADO"
- "VARIABLE SIN TIPO"
- "VARIABLE SIN VALOR"
- “NO EXISTE LA VARIABLE QUE SE BUSCA”
- "OPERACION INVALIDA"
- “FUNCION ESPERABA NO PARAMETROS"
- "FUNCION CON NUMERO DE PARAMETROS ERRONEO"
- "VARIABLE VECTOR SIN DIMENSIONES"
- ~~~~~DIVIDER BETWEEN COMPILING AND EXECUTION~~~~~
- “NONE IN HERE” + Quadruple operation being handled
- “TYPE MISMATCH” + Quadruple operation being handled
- "NOT A CHAR" + Quadruple operation being handled
- "NO EXISTENCE FOR THIS VALUE” + variable virtual address
- "TRYING NONES IN THE SUM QUADS", and REST, TIMES, DIVIDE >, >=, <=, ==,<>,AND,OR

Compiler Description

Physical Computer Equipment, Language and Special Required Utilities

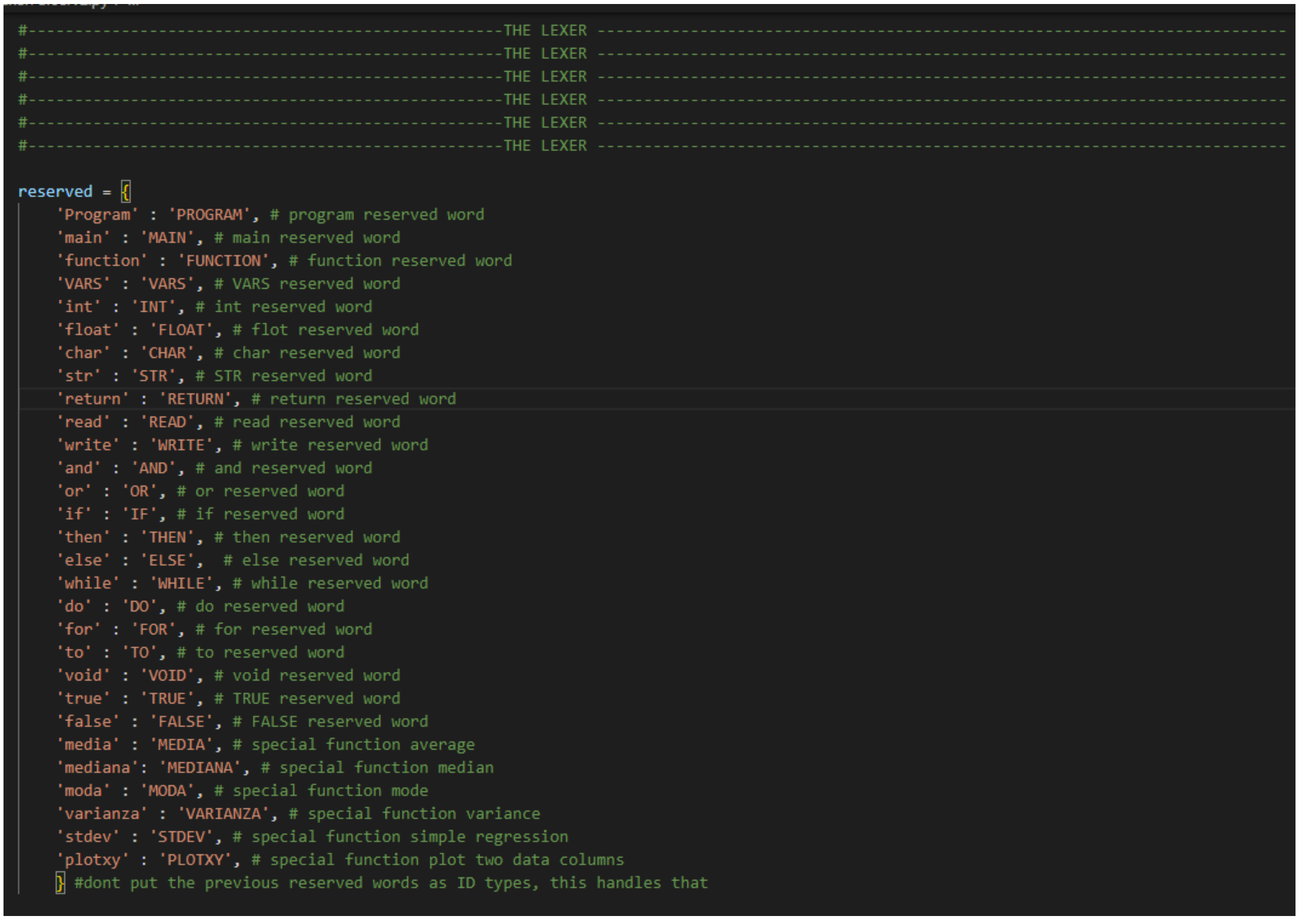
This project was mainly worked on a custom build work desktop, using the Python language as the lexical analyzer, parser, and virtual machine via the judicious use of PLY, using the next special libraries, courtesy from their creators:

- Time library
- Sys library
- Os library

- Lex and yacc form PLY library
- Statistics library
- Matplotlib.pyplot library

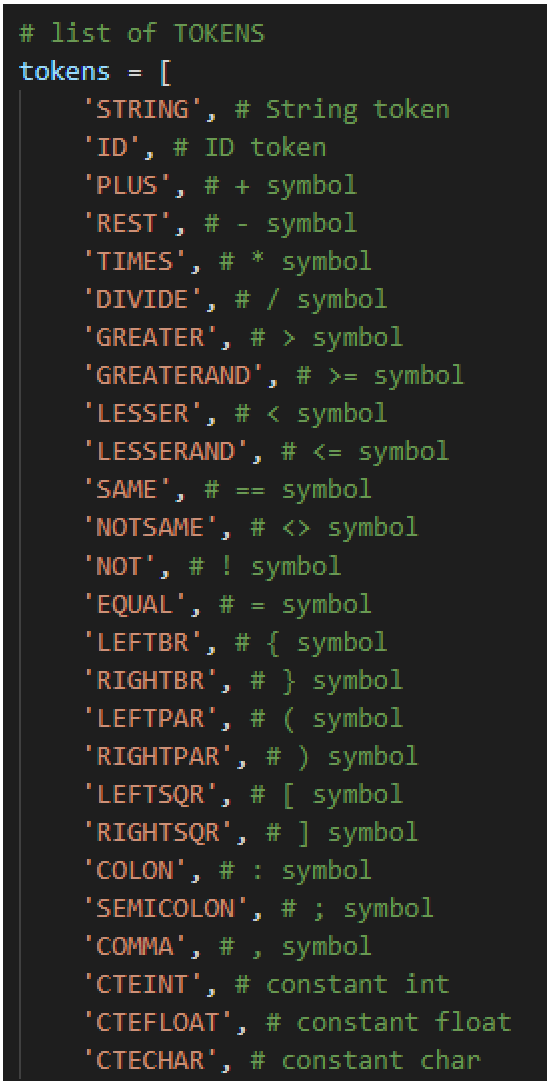
Lexical Analyzer Description

The final list of tokens in the language is as follows, starting with the reserved words:



```
#---------------------------------------------------THE LEXER --------------------------------------------------------------------------
#---------------------------------------------------THE LEXER --------------------------------------------------------------------------
#---------------------------------------------------THE LEXER --------------------------------------------------------------------------
#---------------------------------------------------THE LEXER --------------------------------------------------------------------------
#---------------------------------------------------THE LEXER --------------------------------------------------------------------------
#---------------------------------------------------THE LEXER --------------------------------------------------------------------------

reserved = {
    'Program' : 'PROGRAM', # program reserved word
    'main' : 'MAIN', # main reserved word
    'function' : 'FUNCTION', # function reserved word
    'VARS' : 'VARS', # VARS reserved word
    'int' : 'INT', # int reserved word
    'float' : 'FLOAT', # flot reserved word
    'char' : 'CHAR', # char reserved word
    'str' : 'STR', # STR reserved word
    'return' : 'RETURN', # return reserved word
    'read' : 'READ', # read reserved word
    'write' : 'WRITE', # write reserved word
    'and' : 'AND', # and reserved word
    'or' : 'OR', # or reserved word
    'if' : 'IF', # if reserved word
    'then' : 'THEN', # then reserved word
    'else' : 'ELSE',  # else reserved word
    'while' : 'WHILE', # while reserved word
    'do' : 'DO', # do reserved word
    'for' : 'FOR', # for reserved word
    'to' : 'TO', # to reserved word
    'void' : 'VOID', # void reserved word
    'true' : 'TRUE', # TRUE reserved word
    'false' : 'FALSE', # FALSE reserved word
    'media' : 'MEDIA', # special function average
    'mediana': 'MEDIANA', # special function median
    'moda' : 'MODA', # special function mode
    'varianza' : 'VARIANZA', # special function variance
    'stdev' : 'STDEV', # special function simple regression
    'plotxy' : 'PLOTXY', # special function plot two data columns
    } #dont put the previous reserved words as ID types, this handles that
```

Following that, we have the rest of the tokens:



```
# list of TOKENS
tokens = [
    'STRING', # String token
    'ID', # ID token
    'PLUS', # + symbol
    'REST', # - symbol
    'TIMES', # * symbol
    'DIVIDE', # / symbol
    'GREATER', # > symbol
    'GREATERAND', # >= symbol
    'LESSER', # < symbol
    'LESSERAND', # <= symbol
    'SAME', # == symbol
    'NOTSAME', # <> symbol
    'NOT', # ! symbol
    'EQUAL', # = symbol
    'LEFTBR', # { symbol
    'RIGHTBR', # } symbol
    'LEFTPAR', # ( symbol
    'RIGHTPAR', # ) symbol
    'LEFTSQR', # [ symbol
    'RIGHTSQR', # ] symbol
    'COLON', # : symbol
    'SEMICOLON', # ; symbol
    'COMMA', # , symbol
    'CTEINT', # constant int
    'CTEFLOAT', # constant float
    'CTECHAR', # constant char
```

Finally, we have the following construction patterns via Regex handling of the tokens:

- SEMICOLON = r'\;'
- COLON = r'\:'
- COMMA = r'\,'
- EQUAL = r'\='
- SAME = r'\=\='
- LEFTPAR = r'\('
- RIGHTPAR = r'\)'
- LEFTBR = r'\{'
- RIGHTBR = r'\}'
- LEFTSQR = r'\['
- RIGHTSQR = r'\]'
- STRING = r'\".*\"'

- PLUS = r'\+'
- REST = r'\-'
- TIMES = r'*'
- DIVIDE = r'\/'
- GREATER = r'\>'
- GREATERAND = r'\>\='
- LESSER = r'\<'
- LESSERAND = r'\<\='
- NOTSAME = r'\<\>'
- NOT = r'\!'
- CTECHAR =r"\'.\'"
- CTEFLOAT = r’-?d+\. \d+’
- CTEINT = r’-?\d+’
- ID r’[a-zA-Z_][a-zA-Z0-9]*’

Special thanks to https://regex101.com/ for providing fast and detailed checkups of the regular expressions.

Syntactical Analyzer Description

program ➔ PROGRAM varsgl modules PRINCIPAL LEFTPAR RIGHTPAR LEFTBR statutes RIGHTBR

varsgl ➔ VARS vars

| empty

vars ➔ typing COLON ID varsarr varsmul vars

| empty

varsarr ➔ LEFTSQR CTEINT RIGHTSQR

| empty

varsmul ➔ SEMICOLON

| COMMA ID varsarr varsmul

modules ➔ FUNCTION functype ID funcparam

| empty

funcparam ➔ LEFTPAR parameters RIGHTPAR SEMICOLON varsgl LEFTBR statutes RIGHTBR modules

functype ➔ VOID

| typing

```
statutes ➔ assign statuteaux
            | reading statuteaux
            | writing statuteaux
            | returning statuteaux
            | ifing statuteaux
            | whiling statuteaux
            | foring statuteaux
            | exp statuteaux
            | media statuteaux
            | plotxy statuteaux
            | mediana statuteaux
            | moda statuteaux
            | variance statuteaux
            | stdev statuteaux
statuteaux ➔ statutes
                | empty
media ➔ MEDIA LEFTPAR specfuncnumbers RIGHTPAR SEMICOLON
mediana ➔ MEDIANA LEFTPAR specfuncnumbers RIGHTPAR SEMICOLON
moda ➔ MODA LEFTPAR specfuncnumbers RIGHTPAR SEMICOLON
stdev ➔ STDEV LEFTPAR specfuncnumbers RIGHTPAR SEMICOLON
variance ➔ VARIANZA LEFTPAR specfuncnumbers RIGHTPAR SEMICOLON
plotxy ➔ PLOTXY LEFTPAR specfuncnumbers RIGHTPAR SEMICOLON
specfuncnumbers ➔ CTEINT mulnumeros
                                | CTEFLOAT mulnumeros
mulnumeros ➔ COMMA specfuncnumbers
                     | empty
```

```
typing ➔ INT
        | FLOAT
        | CHAR
parameters ➔ typing COLON ID idarray mulparams
              | empty
mulparams ➔ COMMA parameters
               | empty

assign ➔ ID idarray EQUAL assignexp SEMICOLON
assignexp ➔ exp
idarray ➔ LEFTSQR exp RIGHTSQR
              | empty
returning ➔: RETURN LEFTPAR exp RIGHTPAR SEMICOLON
reading ➔ READ LEFTPAR ID idarray mulread RIGHTPAR SEMICOLON
mulread ➔ COMMA ID idarray mulread
              | empty
writing ➔ WRITE LEFTPAR neuralwrite mulwrite RIGHTPAR SEMICOLON
neuralwrite ➔ writetype
               | exp
writetype ➔ STRING
              | CTECHAR
mulwrite ➔ COMMA neuralwrite mulwrite
               | empty
ifing ➔ IF LEFTPAR exp RIGHTPAR THEN LEFTBR statutes RIGHTBR elsing
elsing ➔ ELSE LEFTBR statutes RIGHTBR
        | empty
whiling ➔ WHILE LEFTPAR exp RIGHTPAR DO LEFTBR statutes RIGHTBR
```

```
foring ➔FOR ID idarray EQUAL exp TO exp DO LEFTBR statutes RIGHTBR
exp ➔ andexp exp1
exp1 ➔ OR exp
           | empty
andexp ➔ boolexp andexp1
andexp1 : AND andexp
              | empty
boolexp : arithexp boolexp1
boolexp1 : neuralbool arithexp
               | empty
neuralbool ➔ GREATER
               | GREATERAND
               | LESSER
               | LESSERAND
               | SAME
               | NOTSAME
               | NOT
arithexp ➔ geoexp arithexp1
arithexp1 ➔ neuralarith arithexp
               | empty
neuralarith ➔ PLUS
                    | REST
geoexp ➔ finexp geoexp1
geoexp1 ➔ neuralgeo geoexp
               | empty
neuralgeo ➔ TIMES
                    | DIVIDE
```

finexp ➔ LEFTPAR exp RIGHTPAR

| cteexp

cteexp ➔ CTEINT

| CTEFLOAT

| CTECHAR

| ID paramsexp

paramsexp ➔ LEFTPAR paramsexp2 RIGHTPAR

| idarray

paramsexp2 ➔ exp mulparamsexp

| empty

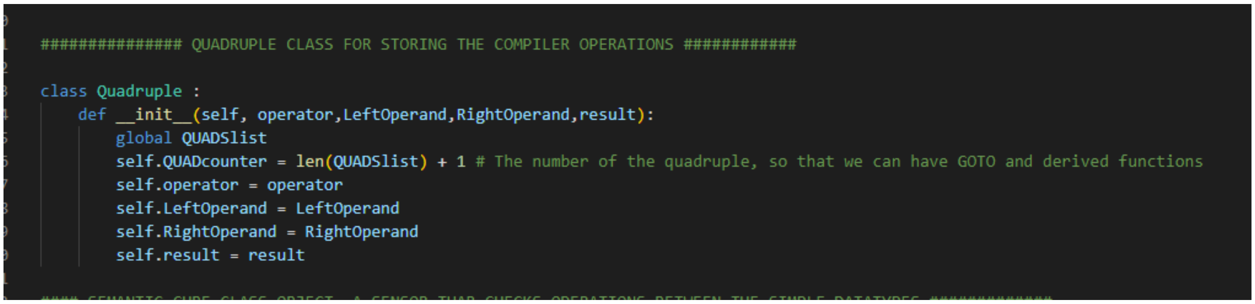
mulparamsexp ➔ COMMA exp mulparamsexp

| empty

empty ➔ ε

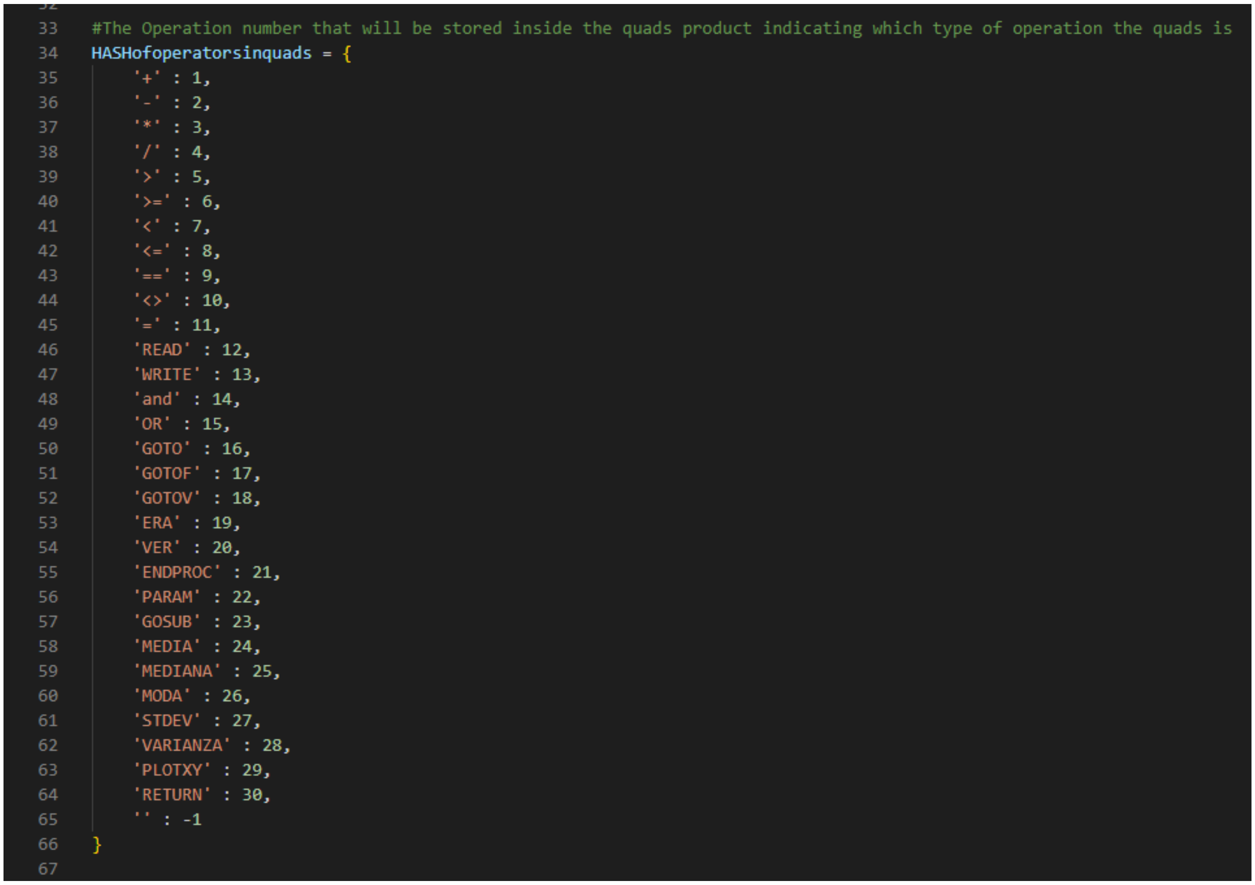
Intermediate Code Generation and Semantical Analysis

One of the core structures of the compiler it's the Quadruple class, which is an object that has the following construction:



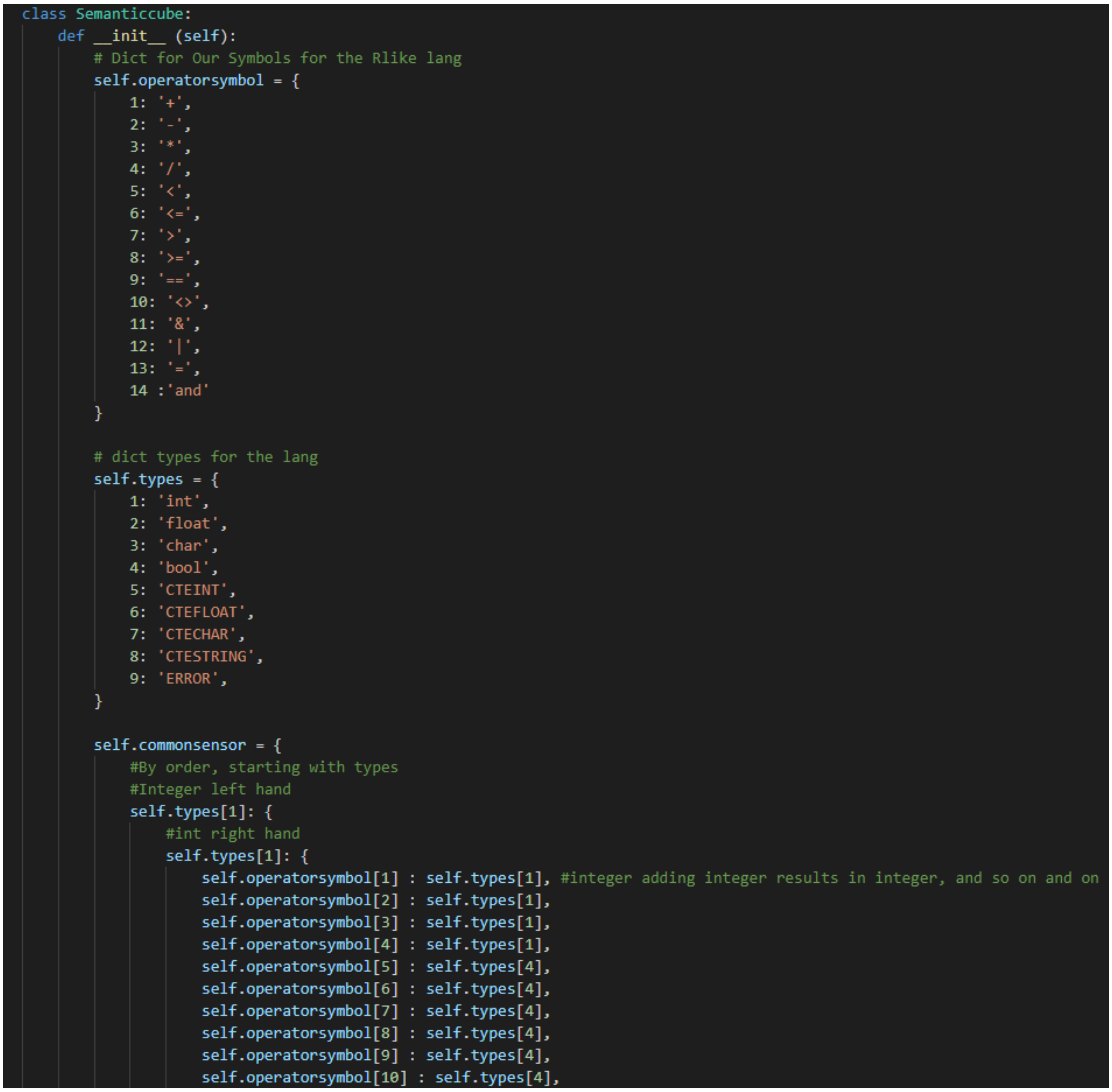
```
############### QUADRUPLE CLASS FOR STORING THE COMPILER OPERATIONS ############

class Quadruple :
    def __init__(self, operator,LeftOperand,RightOperand,result):
        global QUADSlist
        self.QUADcounter = len(QUADSlist) + 1 # The number of the quadruple, so that we can have GOTO and derived functions
        self.operator = operator
        self.LeftOperand = LeftOperand
        self.RightOperand = RightOperand
        self.result = result
```

In which we have 5 attributes, with the typical 4 of a quadruple container, and the extra quadcounter for jump purposes, and future VM use. It must be noted that the attribute operator follows a special hash map, in which depending on what token it has, it stores a numerical value, as follows:



```
#The Operation number that will be stored inside the quads product indicating which type of operation the quads is
HASHofoperatorsinquads = {
    '+' : 1,
    '-' : 2,
    '*' : 3,
    '/' : 4,
    '>' : 5,
    '>=' : 6,
    '<' : 7,
    '<=' : 8,
    '==' : 9,
    '<>' : 10,
    '=' : 11,
    'READ' : 12,
    'WRITE' : 13,
    'and' : 14,
    'OR' : 15,
    'GOTO' : 16,
    'GOTOF' : 17,
    'GOTOV' : 18,
    'ERA' : 19,
    'VER' : 20,
    'ENDPROC' : 21,
    'PARAM' : 22,
    'GOSUB' : 23,
    'MEDIA' : 24,
    'MEDIANA' : 25,
    'MODA' : 26,
    'STDEV' : 27,
    'VARIANZA' : 28,
    'PLOTXY' : 29,
    'RETURN' : 30,
    '' : -1
}

```

The semantical considerations seen on this project was primarily the responsibility of an external class, aptly named the Semantic Cube, which has an internal method to do semantic checks, accepting two operands and an operator, and returning the appropriate type for the semantic sensor. Its structure is presented on the following image:

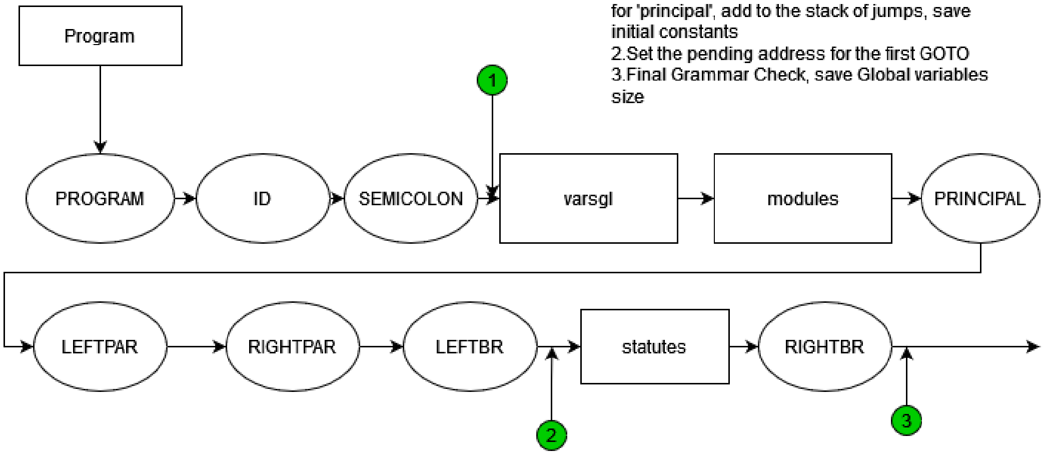


```
class Semanticcube:
    def __init__ (self):
        # Dict for Our Symbols for the Rlike lang
        self.operatorsymbol = {
            1: '+',
            2: '-',
            3: '*',
            4: '/',
            5: '<',
            6: '<=',
            7: '>',
            8: '>=',
            9: '==',
            10: '<>',
            11: '&',
            12: '|',
            13: '=',
            14 :'and'
        }

        # dict types for the lang
        self.types = {
            1: 'int',
            2: 'float',
            3: 'char',
            4: 'bool',
            5: 'CTEINT',
            6: 'CTEFLOAT',
            7: 'CTECHAR',
            8: 'CTESTRING',
            9: 'ERROR',
        }

        self.commonsensor = {
            #By order, starting with types
            #Integer left hand
            self.types[1]: {
                #int right hand
                self.types[1]: {
                    self.operatorsymbol[1] : self.types[1], #integer adding integer results in integer, and so on and on
                    self.operatorsymbol[2] : self.types[1],
                    self.operatorsymbol[3] : self.types[1],
                    self.operatorsymbol[4] : self.types[1],
                    self.operatorsymbol[5] : self.types[4],
                    self.operatorsymbol[6] : self.types[4],
                    self.operatorsymbol[7] : self.types[4],
                    self.operatorsymbol[8] : self.types[4],
                    self.operatorsymbol[9] : self.types[4],
                    self.operatorsymbol[10] : self.types[4],
```

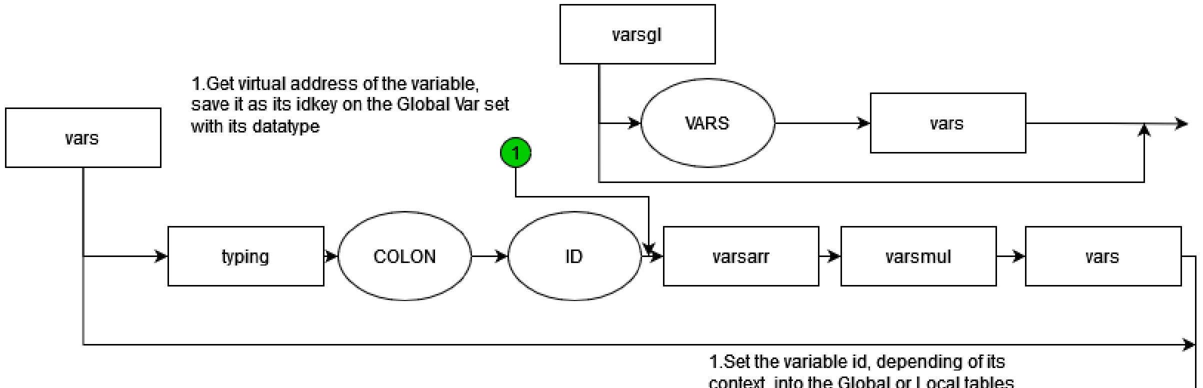
We can see that the class semantic cube has three attributes, a set of operator symbols, which stores the symbols that our language will use for operations. Then we have a second set of types which will be assigned a certain numerical key for ease of use. And finally, we have a nested set of sets, in which we have three layers, the first layers represent the left type of an operation, the second level represents the right hand of an operations, and the third layer in which we store the specific operator we are working with. Finally, the combined 3 dimensions point to a single value, which is the result of the combining of the previous two types, leading to a useful common sensor, giving you which types can work with which operators, and gives you an error message when you break the logic of the code.

Next, we have the exciting part of the syntaxis diagram with the noted neuralgic points, giving us the code actions that we need to set up the inner works of our compiler:

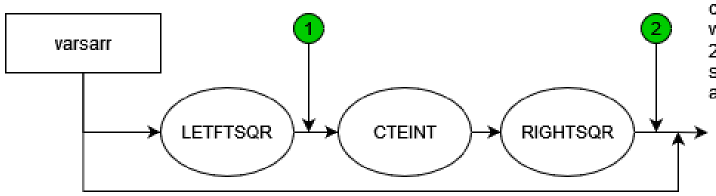
1.Insert the name of the function in the TheTableofFuncions, generate the GOTO quad for 'principal', add to the stack of jumps, save initial constants
2.Set the pending address for the first GOTO
3.Final Grammar Check, save Global variables size

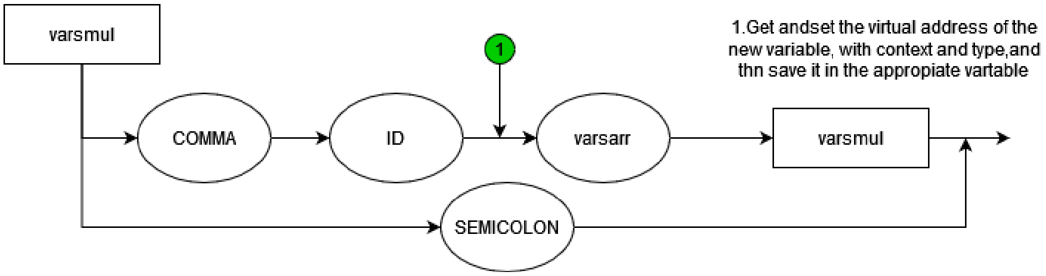


1.Get virtual address of the variable, save it as its idkey on the Global Var set with its datatype

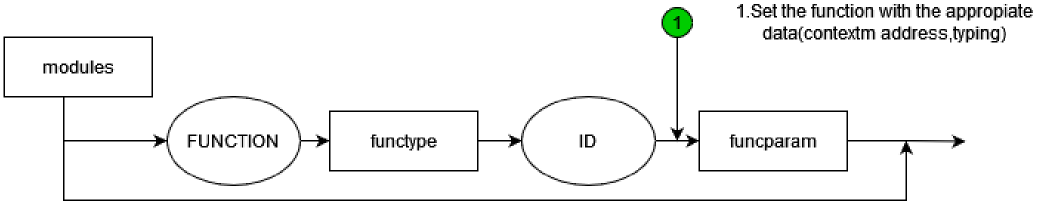


1.Set the variable id, depending of its context, into the Global or Local tables with a sensor for arrays
2.Get the resolved size of the array, store it at the constant table if not already there.

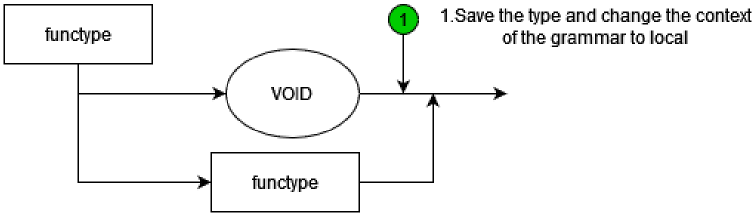




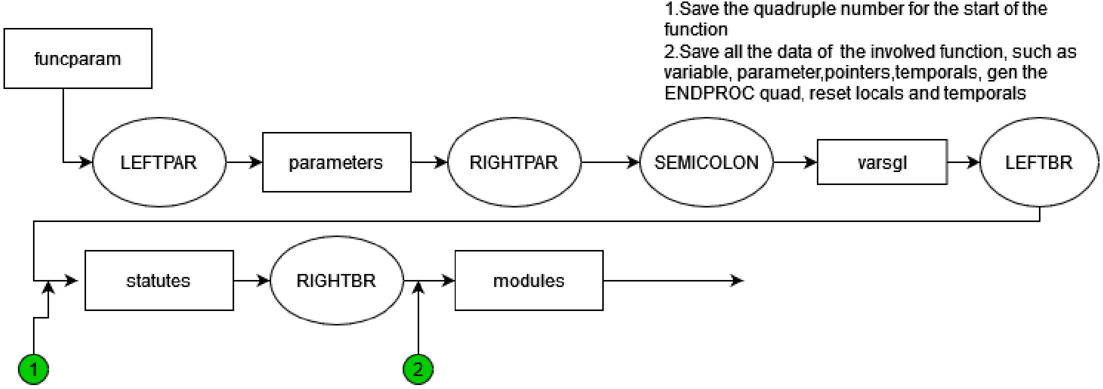
1.Get andset the virtual address of the new variable, with context and type,and thn save it in the appropiate vartable



1.Set the function with the appropiate data(contextm address,typing)

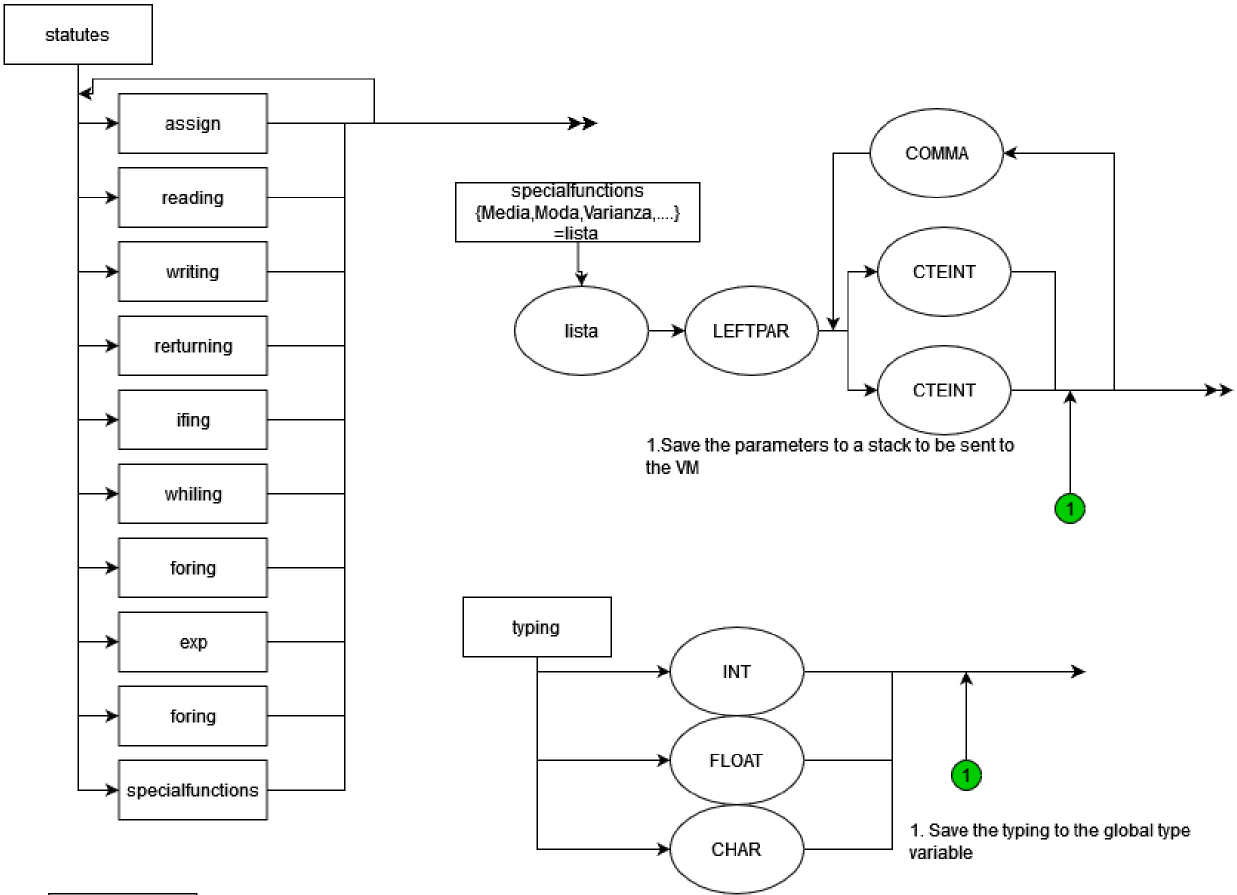


1.Save the type and change the context of the grammar to local



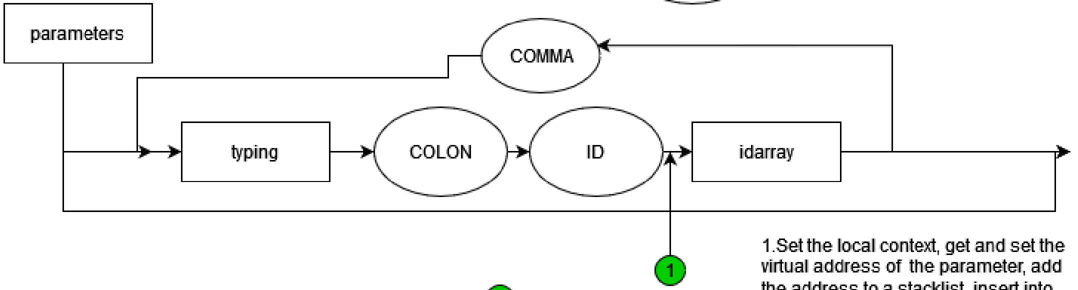
1.Save the quadruple number for the start of the function

2.Save all the data of the involved function, such as variable, parameter,pointers,temporals, gen the ENDPROC quad, reset locals and temporals

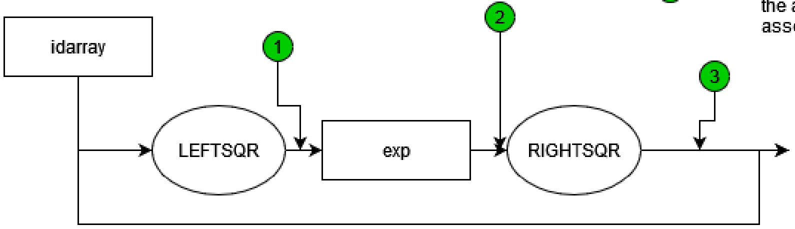


1.Save the parameters to a stack to be sent to the VM

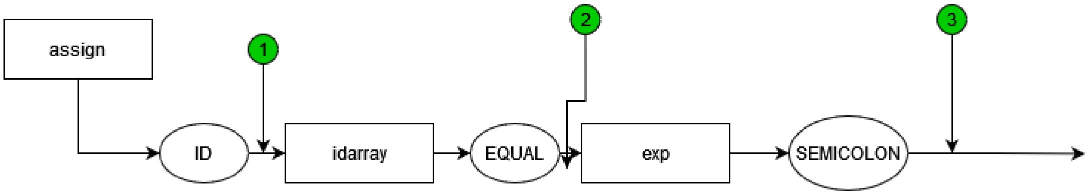
1. Save the typing to the global type variable



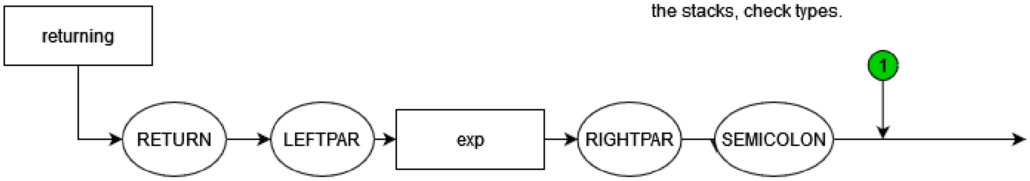
1.Set the local context, get and set the virtual address of the parameter, add the address to a stacklist, insert into associate tables.



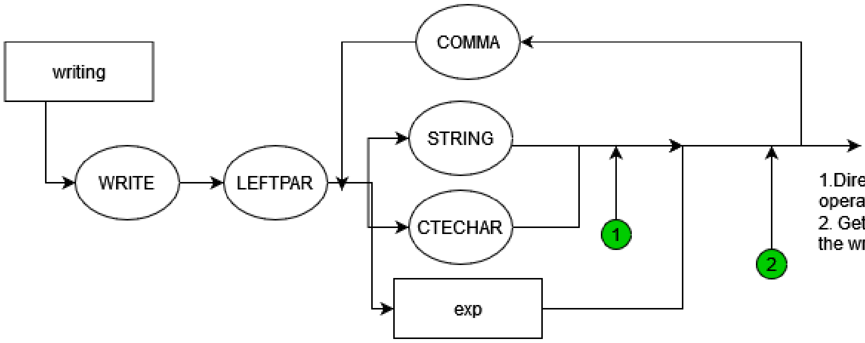
1.Handle the vector by popign the stackofoperators and stackofoperands, get the name,check for valid vector start, add to stackofdims, add fakebottom.
2. Make the quadruple VER with the limits
3.With a working vector, get constant addr for the dim, save, get pointer, add extra quads for the VER process.



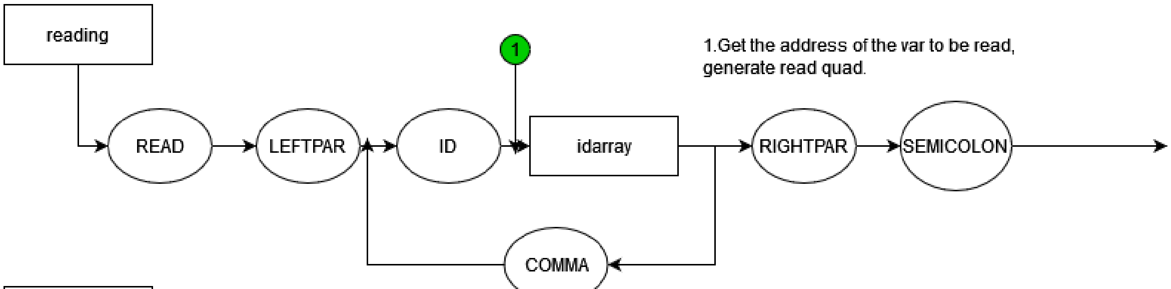
1.Get the address of the id we are looking at, add to the stackofoperands and add the type to the stackoftypes
2.Add to stack of operators the =
3.Generate the = quadruple via popping operands and types in the stacks, check types.



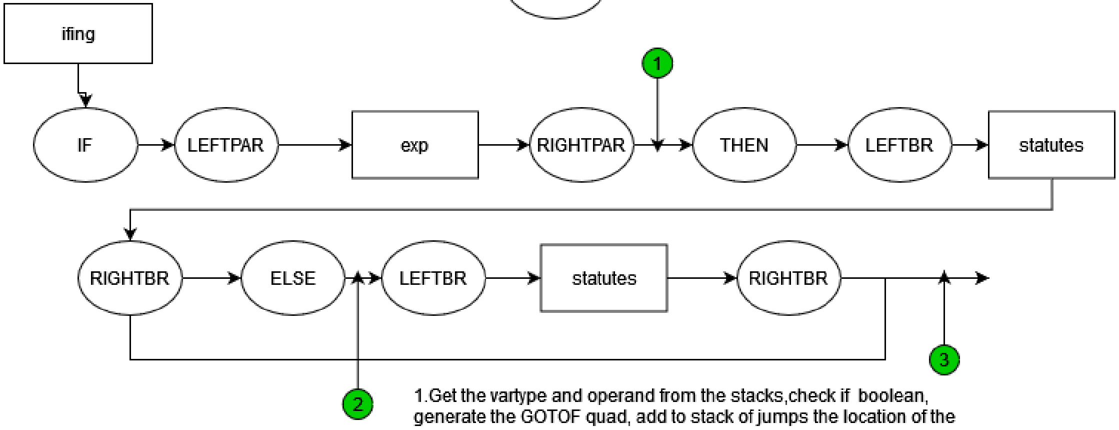
1.Generate return quad via poping the operandstack and the typestack,get the address for the global var to be assigned the return value.Add to quadrupleslist



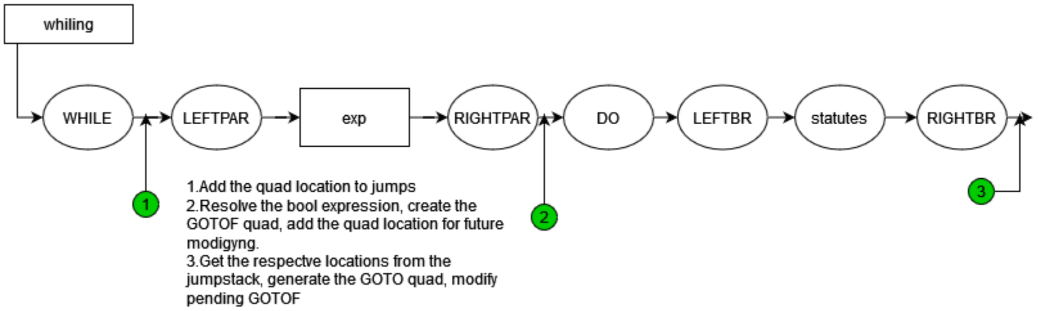
1.Directly append the token to the stack of operands.
2. Get the operand from its stack, generate the write quadruple.



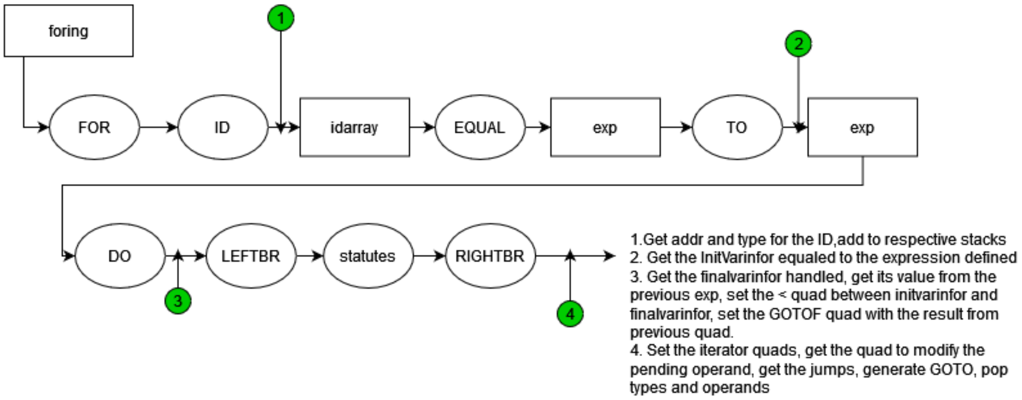
1.Get the address of the var to be read, generate read quad.



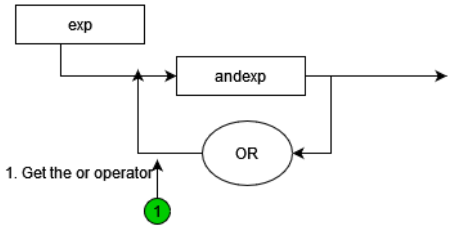
1.Get the vartype and operand from the stacks,check if boolean, generate the GOTOF quad, add to stack of jumps the location of the next quad.
2.If pending jump, add GOTO quad via poping jump, add the location of the else to the jumps, modify the pending GOTOF quad.
3. Modify the pending GOTO to get the correct quadruplecounter.



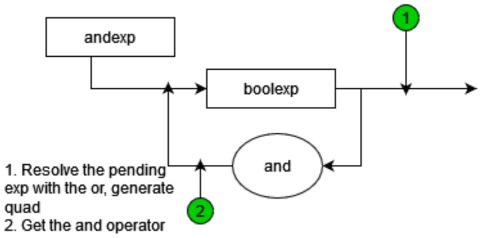
1.Add the quad location to jumps
2.Resolve the bool expression, create the GOTOF quad, add the quad location for future modigyng.
3.Get the respectve locations from the jumpstack, generate the GOTO quad, modify pending GOTOF



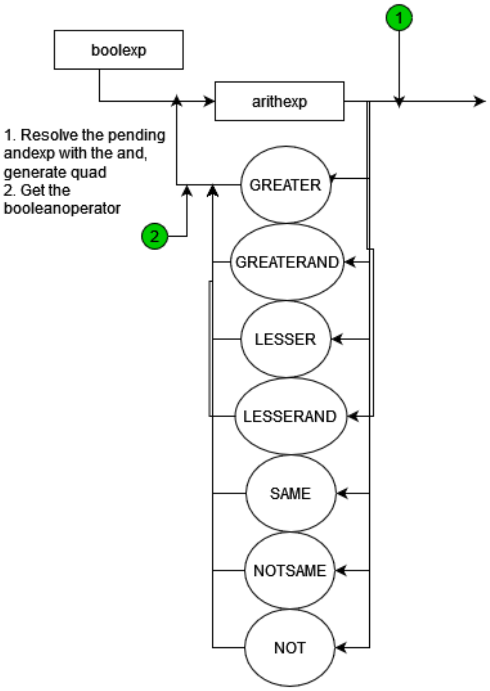
1.Get addr and type for the ID,add to respective stacks
2. Get the InitVarinfor equaled to the expression defined
3. Get the finalvarinfor handled, get its value from the previous exp, set the < quad between initvarinfor and finalvarinfor, set the GOTOF quad with the result from previous quad.
4. Set the iterator quads, get the quad to modify the pending operand, get the jumps, generate GOTO, pop types and operands



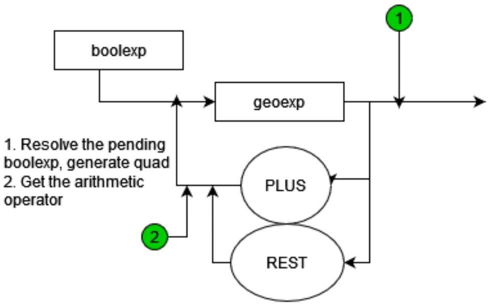
1. Get the or operator



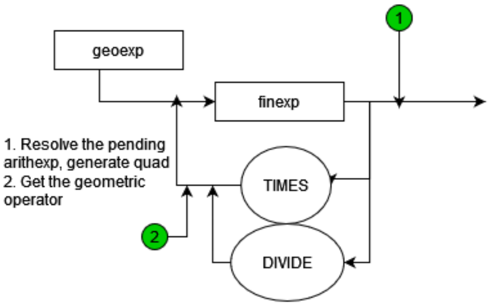
1. Resolve the pending exp with the or, generate quad
2. Get the and operator



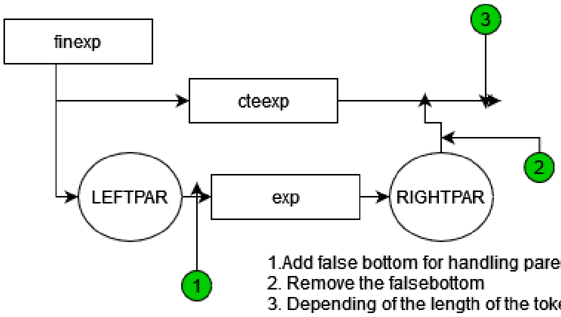
1. Resolve the pending andexp with the and, generate quad
2. Get the booleanoperator



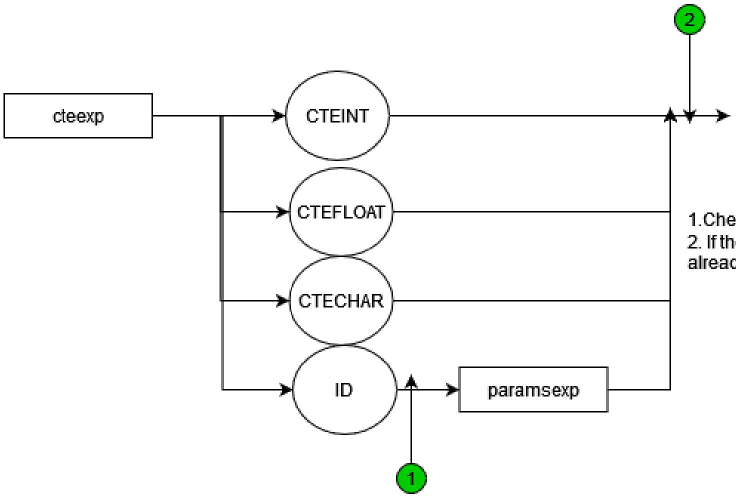
1. Resolve the pending boolexp, generate quad
2. Get the arithmetic operator



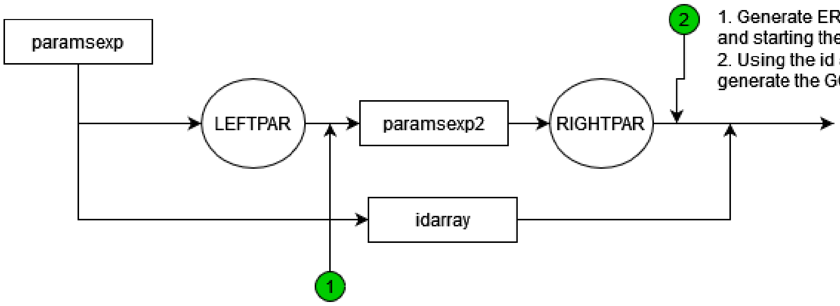
1. Resolve the pending arithexp, generate quad
2. Get the geometric operator



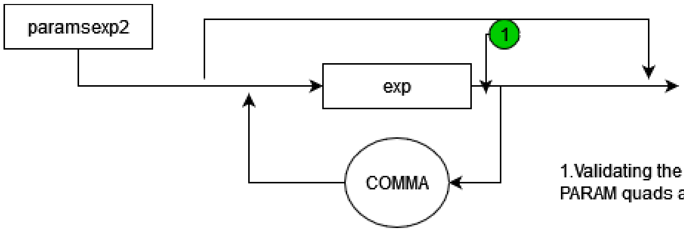
1.Add false bottom for handling parenthesis
2. Remove the falsebottom
3. Depending of the length of the token, we can see if its anther finexp, a function or a vector. Depending on what it is, you can add the function value, and deal with its typing, handle the vector or deal with the pending finexp via generating quad



1.Check if the supposed variable actually exists.
2. If the possible constant (possible parameters) isnt already in the constants table, save it.



1. Generate ERA quad, appending false bottom, and starting the contparameterslist.
2. Using the id and parameters datastructures, generate the GOSUB quad.

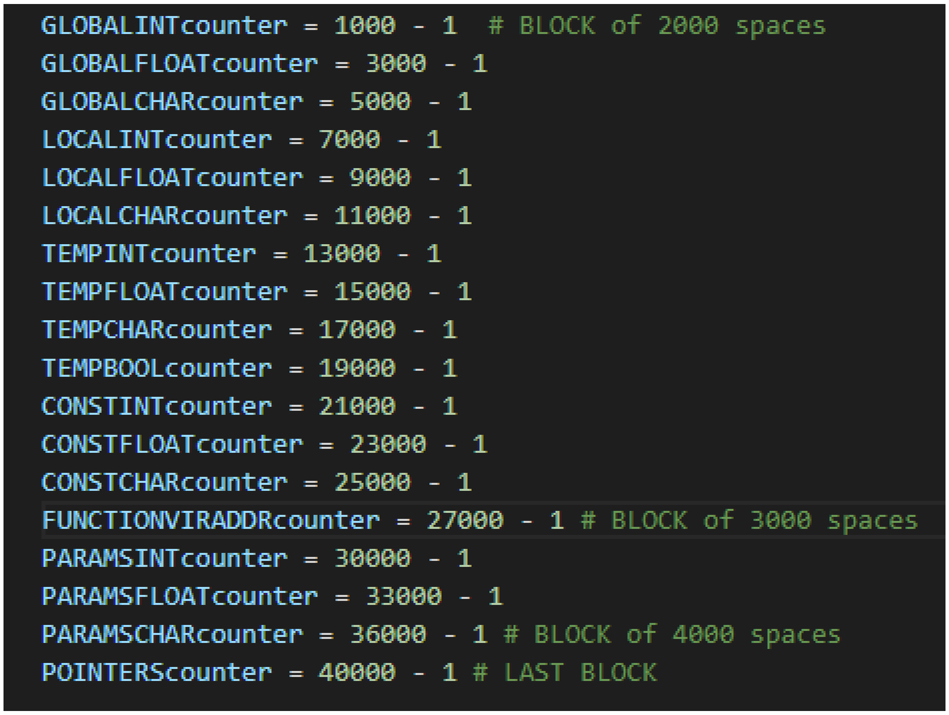


1.Validating the arguments via pops, create the PARAM quads and add it to the contparameters list

Full resolution inside the project folder.

Memory administration during the compilation process

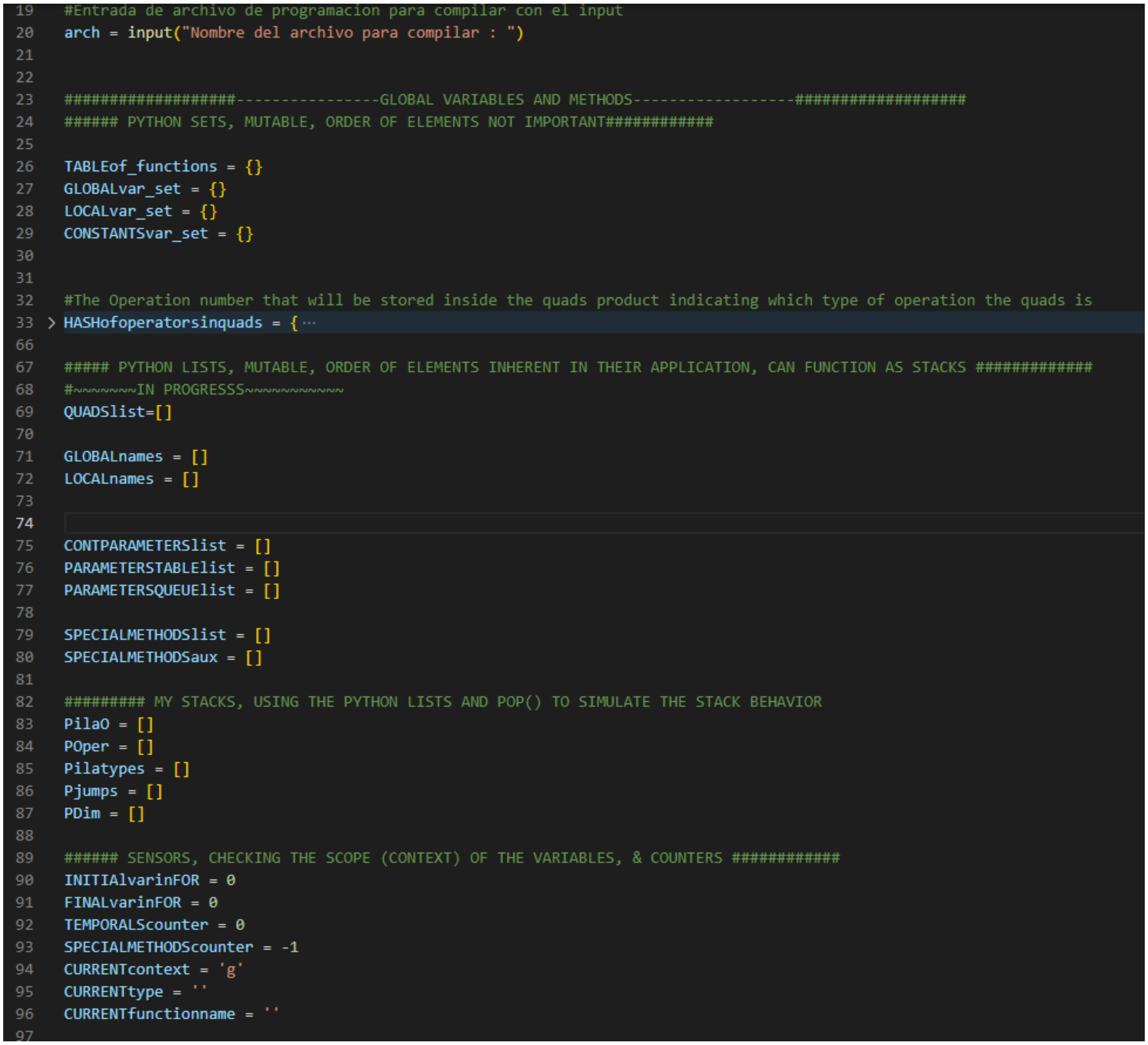
During the compilation process, we had to abstract away the storing of values, via the use of 'virtual' addresses, which stand in place of possible memory blocks. Due to this, we must organize a virtual memory map for the storing of our values. Its structure it's the following:



```
GLOBALINTcounter = 1000 - 1  # BLOCK of 2000 spaces
GLOBALFLOATcounter = 3000 - 1
GLOBALCHARcounter = 5000 - 1
LOCALINTcounter = 7000 - 1
LOCALFLOATcounter = 9000 - 1
LOCALCHARcounter = 11000 - 1
TEMPINTcounter = 13000 - 1
TEMPFLOATcounter = 15000 - 1
TEMPCHARcounter = 17000 - 1
TEMPBOOLcounter = 19000 - 1
CONSTINTcounter = 21000 - 1
CONSTFLOATcounter = 23000 - 1
CONSTCHARcounter = 25000 - 1
FUNCTIONVIRADDRcounter = 27000 - 1 # BLOCK of 3000 spaces
PARAMSINTcounter = 30000 - 1
PARAMSFLOATcounter = 33000 - 1
PARAMSCHARcounter = 36000 - 1 # BLOCK of 4000 spaces
POINTERScounter = 40000 - 1 # LAST BLOCK
```

We start with block of 2000 spaces, going through our possible type of variables we must store, function results that also need to be saved, and actual quadruple addresses in the pointers. Some blocks have more than 2000 spaces, such as the possible function values, or the storage space of char parameters.

We also handle a lot of data structures to help with the compilation process, which can be seen in the following image:



```
#Entrada de archivo de programacion para compilar con el input
arch = input("Nombre del archivo para compilar : ")


###################----------------GLOBAL VARIABLES AND METHODS------------------###################
###### PYTHON SETS, MUTABLE, ORDER OF ELEMENTS NOT IMPORTANT############

TABLEof_functions = {}
GLOBALvar_set = {}
LOCALvar_set = {}
CONSTANTSvar_set = {}


#The Operation number that will be stored inside the quads product indicating which type of operation the quads is
HASHofoperatorsinquads = {···

##### PYTHON LISTS, MUTABLE, ORDER OF ELEMENTS INHERENT IN THEIR APPLICATION, CAN FUNCTION AS STACKS #############
#~~~~~~~~IN PROGRESSS~~~~~~~~~~~~
QUADSlist=[]

GLOBALnames = []
LOCALnames = []


CONTPARAMETERSlist = []
PARAMETERSTABLElist = []
PARAMETERSQUEUElist = []

SPECIALMETHODSlist = []
SPECIALMETHODSaux = []

######### MY STACKS, USING THE PYTHON LISTS AND POP() TO SIMULATE THE STACK BEHAVIOR
PilaO = []
POper = []
Pilatypes = []
Pjumps = []
PDim = []

###### SENSORS, CHECKING THE SCOPE (CONTEXT) OF THE VARIABLES, & COUNTERS ############
INITIALvarinFOR = 0
FINALvarinFOR = 0
TEMPORALScounter = 0
SPECIALMETHODScounter = -1
CURRENTcontext = 'g'
CURRENTtype = ''
CURRENTfunctionname = ''
```

Each one was used in the compilation process, some more than others, and their descriptions is as follows:

- Tableof_functions: The central set where all the functions are stored, with their ids, size, initial address, and variables. Due to not needing to heed a certain order, this is a set, which speeds up the search process.
- GLOBALvar_set: Like the above set, but storing the information of globals, with their id, virtual address, and type and related information. Also, a set with the same benefits.
- LOCALvar_set: Same as GLOBALvar_set, but with locals. Capable of being flushed to make space for new local variables.
- CONSTANTSvar_set: Same as the the GLOBALvar_set, but with constants.

- GLOBALnames: Structure to store the actual written names of the global variables, to prevent duplication. On second look, this should be a set, or be folded into GLOBALvar_set.
- LOCALnames: Same use as the above structure, but with local variables. Also shares the same caveats as GLOBALnames and should be deprecated.
- QUADSlist: MOST important data structure, which stores the compiler output as a list of quadruple objects, which we have already described earlier. Due to the need to be outputted to a Virtual Machine, its vastly preferable that the structure maintains its order, so that the reading of its data can be done directly, instead of sorting it in the importation process.
- CONTPARAMETERSlist: A list used as a stack, used in the verification process of the call of a function in the neuralgic points. Due to the need of being popped in the process, is a list.
- PARAMETERSTABLElist: A list storing every data type for a function call. Due to being used as a verifier of the CONTPARAMETERSlist, which follows an ordered nature, the PARAMETERSTABLElist must also be ordered, hence a list.
- PARAMETERQUEUElist: A list that stores the virtual address of the parameters, due to being mated to the CONTPARAMETERSlist in the process of generating PARAM quads, it must be ordered, so it must be a list.
- SPECIALMETHODlist: The datastructure that the compiler uses to store constant values for the special method calls, directly accessed by the VM. Due to the nature of certain statistical methods, it must preserve its order, so it's a list.
- SPECIALMETHODaux: A nested structure of the special methods parameters, storing constants. Could be refactored.
- PilaO: It's the central stack to get the necessary quadruple structure of operands, works via virtual addresses so its values can be stored in the outputted quadruples. As seen in class, a stack.
- POper: The central stack for operator symbols, its destination it's to be hashed via the HASHofoperatorsinquads, so that the output can be put directly into the quadruples. AS seen in class, a stack.
- Pilatypes: The mirror stack of operators, but handling types, is used by global methods and the semantic cube so that the resultant quadruples are working as intended. Due to being a mirror of a stack, it also is a stack.
- Pjumps: A stack that stores the quadruple counters of the QUADSlist and is used to handle jumps. Due to the need of it to be to be a stack due to nesting for appropriate jump calls, is a list.
- PDim: Leftover of attempts of arrays of more than one dimension, currently used a backup storage of the operand in a vector.
- The rest are simple counters, simple sensors, placeholder values in the for iterators, and simple containers for storing function names and types.

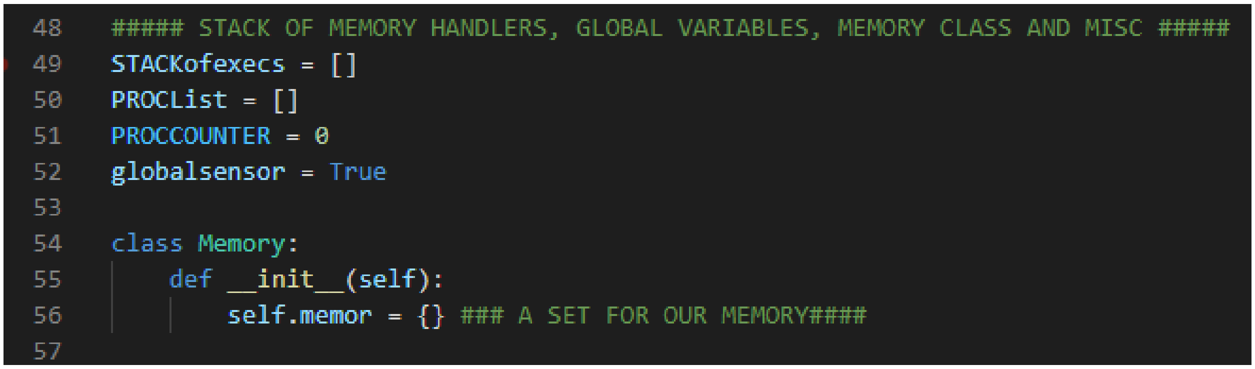
Virtual Machine Description

Physical Computer Equipment, Language and Special Required Utilities

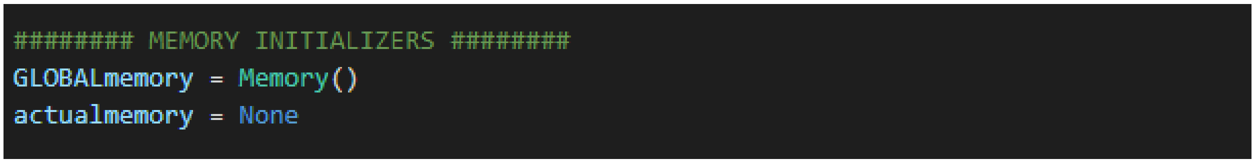
The same characteristics as used in the compiler description.

Memory administration during code execution

As described in class, compilers output intermediate machine code, and if available with an interpreter, this machine code can be managed to output actual operations. Due to this, most interpreters must be able to read the quadruples (if working with actual quadruples) and its elements and be able to output the specific operations associated with those quadruples. This can be done with certain generic elements, which can be seen here declared in our Virtual machine here:



```
##### STACK OF MEMORY HANDLERS, GLOBAL VARIABLES, MEMORY CLASS AND MISC #####
STACKofexecs = []
PROCList = []
PROCCOUNTER = 0
globalsensor = True

class Memory:
    def __init__(self):
        self.memor = {} ### A SET FOR OUR MEMORY####

```



```
######## MEMORY INITIALIZERS ########
GLOBALmemory = Memory()
actualmemory = None
```

This are all recognized parts of a VM machine, and follows much of the same logic, but due to the nature of our quadruples, and its hashed operators, and stored quadruple counters, much of the interpreting work can be sped up and be processed via simpler methods, which is centered on a while loop looking at each quadruple by quadruple.

- STACKofexecs: Used as a stack of executions, primarily focused with function quadruples, meaning ERA's and ENDPROC's. Works with verifying interactions between values in an out of functions and accessing local memory for the address values. Via working of nested functions, must be a stack, which permits us to let memory enter a pseudo-dormant state.
- PROCList: A stack that works with the function quadruples, works via implementing the of functions executions, and allows to store the jumps.
- PROCounter: Simple counter that works with the quadruple counters to run the intermediate machine code. Normally follows an iterative process.
- Globalsensor: Simple sensor if we are not working inside functions.
- Memory class: A simple object, that stores an attribute set, which speeds up massively code lookups, in which we load the actual data of our variables, dividing

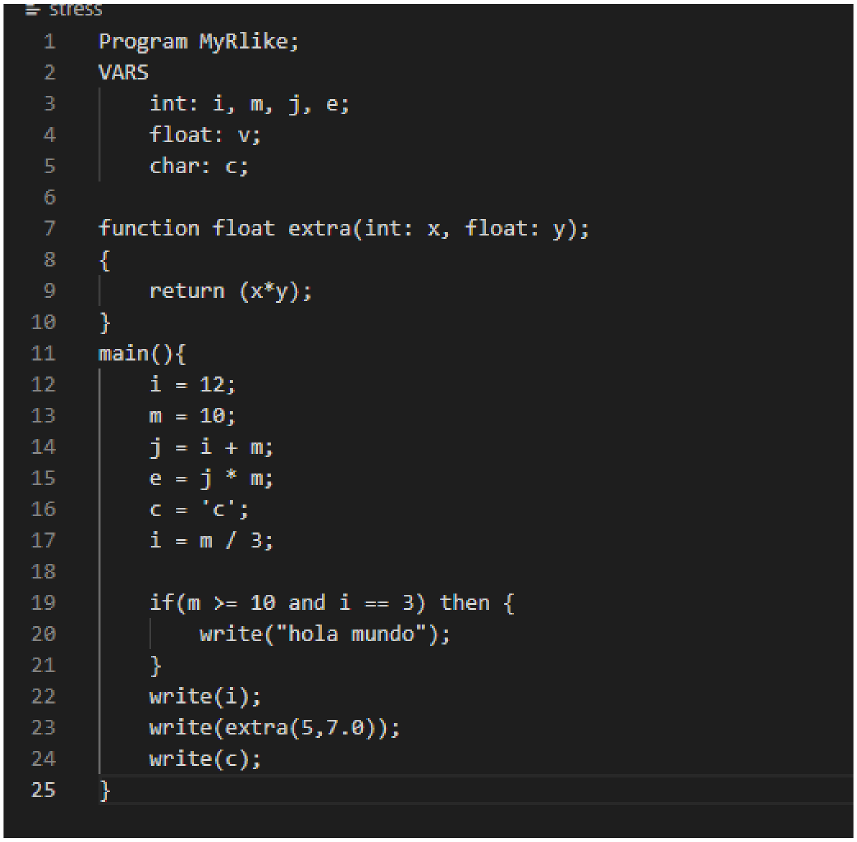
it in two objects, a GLOBALmemory object that is always active and available to be called, and an actualmemory, which is the local memory of the functions, capable of dormancy and only being flushed when entering an ENDPROC quadruple or a return quadruple.

With this data structures, and global methods and error handlings with some external handling of the constants table that was imported directly from the parser, we can have a simple Virtual Machine, which enters a while cycle, which reads directly the Quads object, which stored in the intermediate machine code, divides it into 5 variables, which are, index, operat, leftoperd, rightoperd, result, following the normal structure of a quadruple, and depending of the operat value which we know was hashed, we enter a list of elifs, and using the appropriate memory via the use of sensors and error checking, directly access the stored values of the virtual address, and via Python directly apply the operation. With a direct link between the virtual addresses used to output in the compiler, and the loading of a memory, that abstracts away the ability of a computer to read stored bytes as actual values, we get in the end a working compilator.

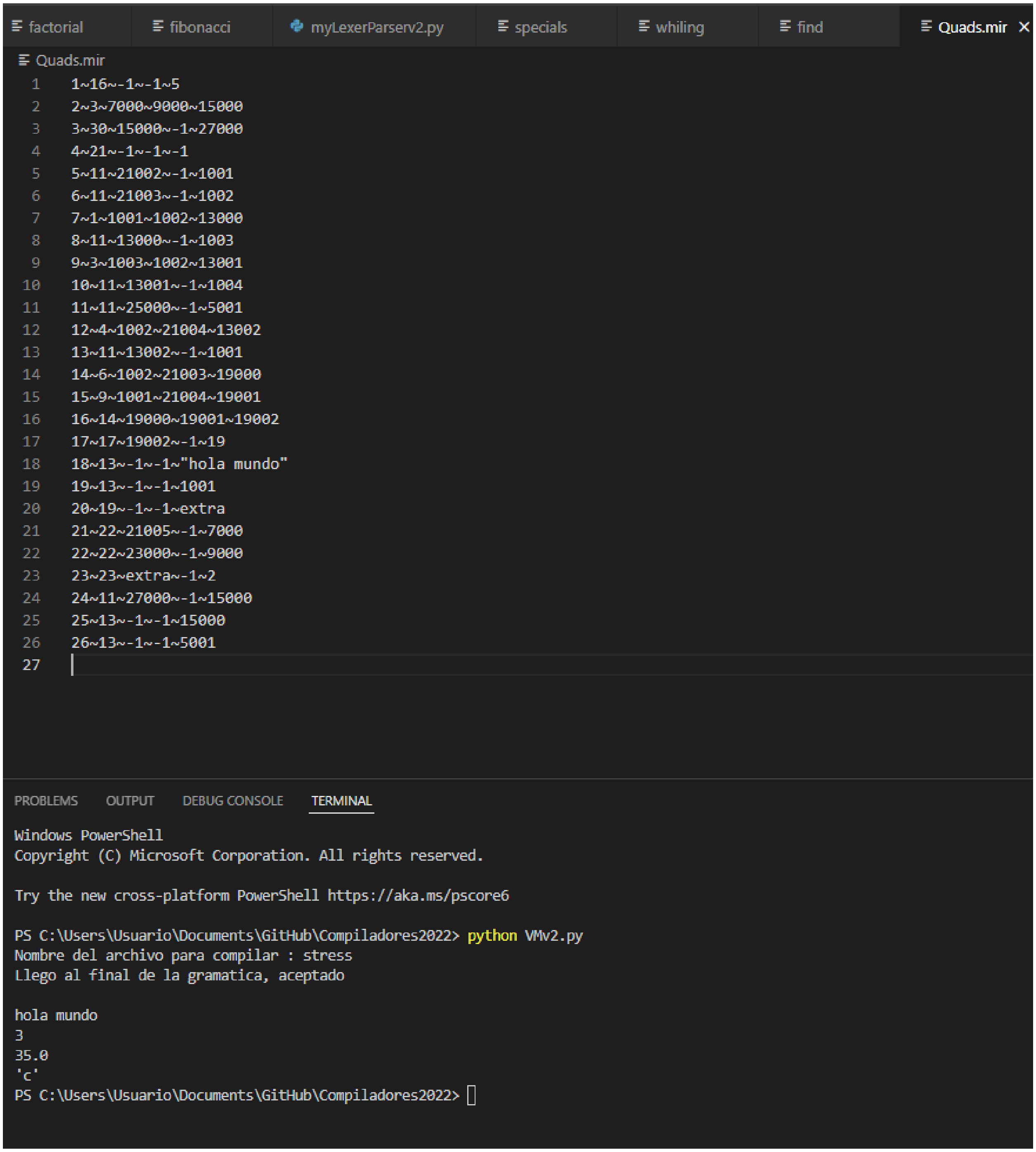
Working tests of the language

In here, we are going to show the three-step process of the original mir code, the intermediate machine code, and the execution process.

Stress testing



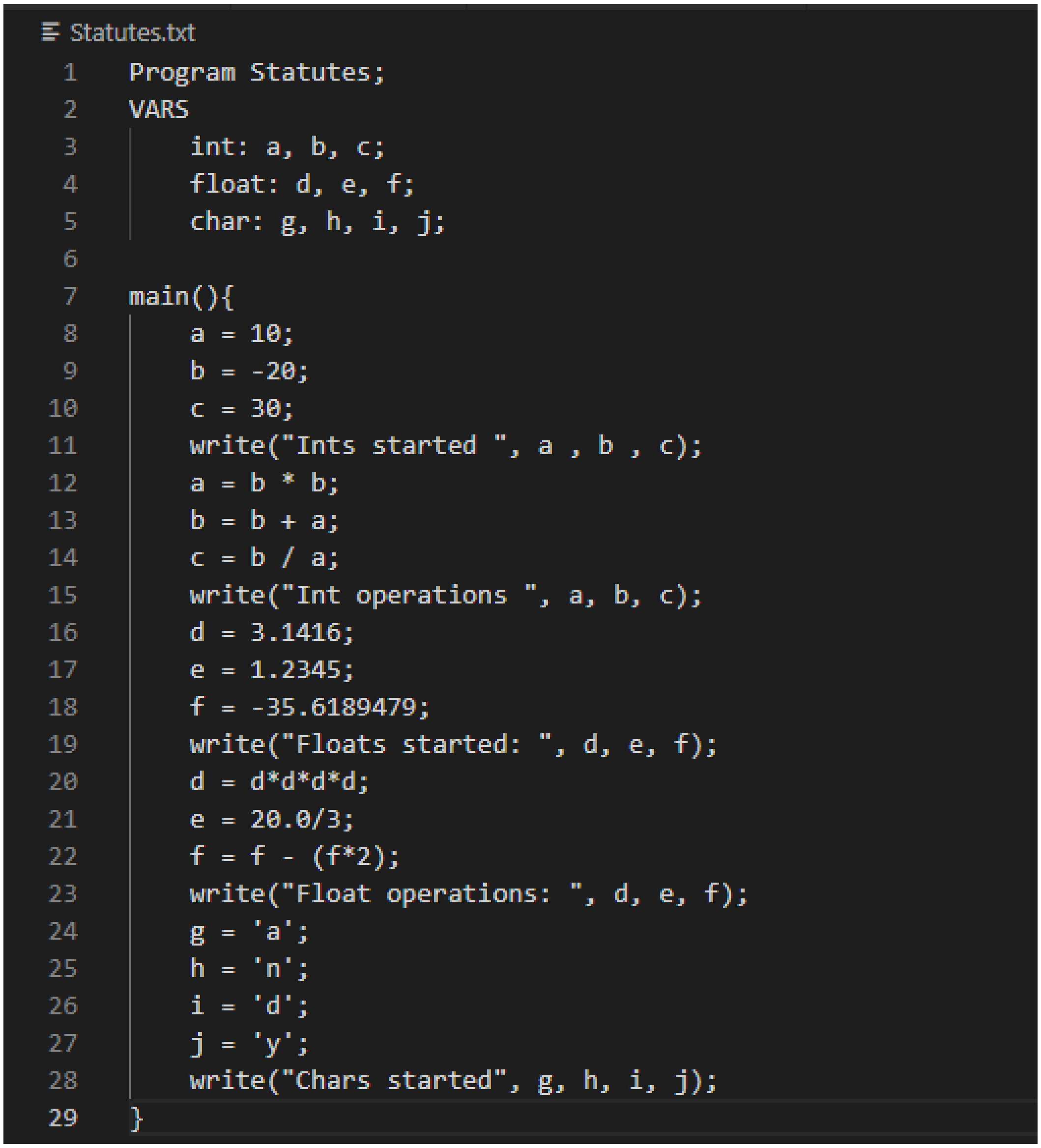
```
Program MyRlike;
VARS
    int: i, m, j, e;
    float: v;
    char: c;

function float extra(int: x, float: y);
{
    return (x*y);
}
main(){
    i = 12;
    m = 10;
    j = i + m;
    e = j * m;
    c = 'c';
    i = m / 3;

    if(m >= 10 and i == 3) then {
        write("hola mundo");
    }
    write(i);
    write(extra(5,7.0));
    write(c);
}
```



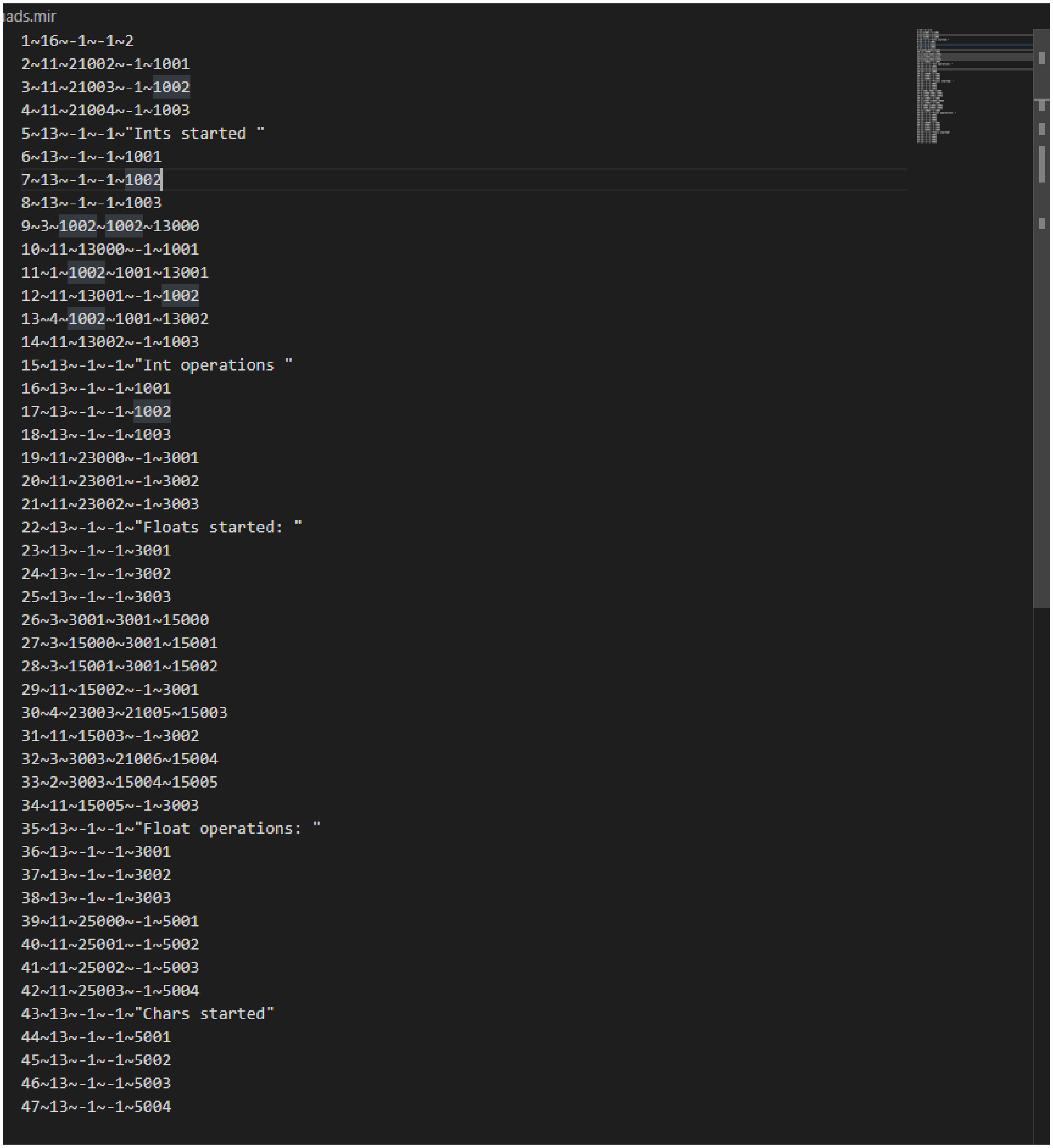
```
1~16~-1~-1~5
2~3~7000~9000~15000
3~30~15000~-1~27000
4~21~-1~-1~-1
5~11~21002~-1~1001
6~11~21003~-1~1002
7~1~1001~1002~13000
8~11~13000~-1~1003
9~3~1003~1002~13001
10~11~13001~-1~1004
11~11~25000~-1~5001
12~4~1002~21004~13002
13~11~13002~-1~1001
14~6~1002~21003~19000
15~9~1001~21004~19001
16~14~19000~19001~19002
17~17~19002~-1~19
18~13~-1~-1~"hola mundo"
19~13~-1~-1~1001
20~19~-1~-1~extra
21~22~21005~-1~7000
22~22~23000~-1~9000
23~23~extra~-1~2
24~11~27000~-1~15000
25~13~-1~-1~15000
26~13~-1~-1~5001
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> python VMv2.py
Nombre del archivo para compilar : stress
Llego al final de la gramatica, aceptado

hola mundo
3
35.0
'c'
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022>
```

Statutes testing

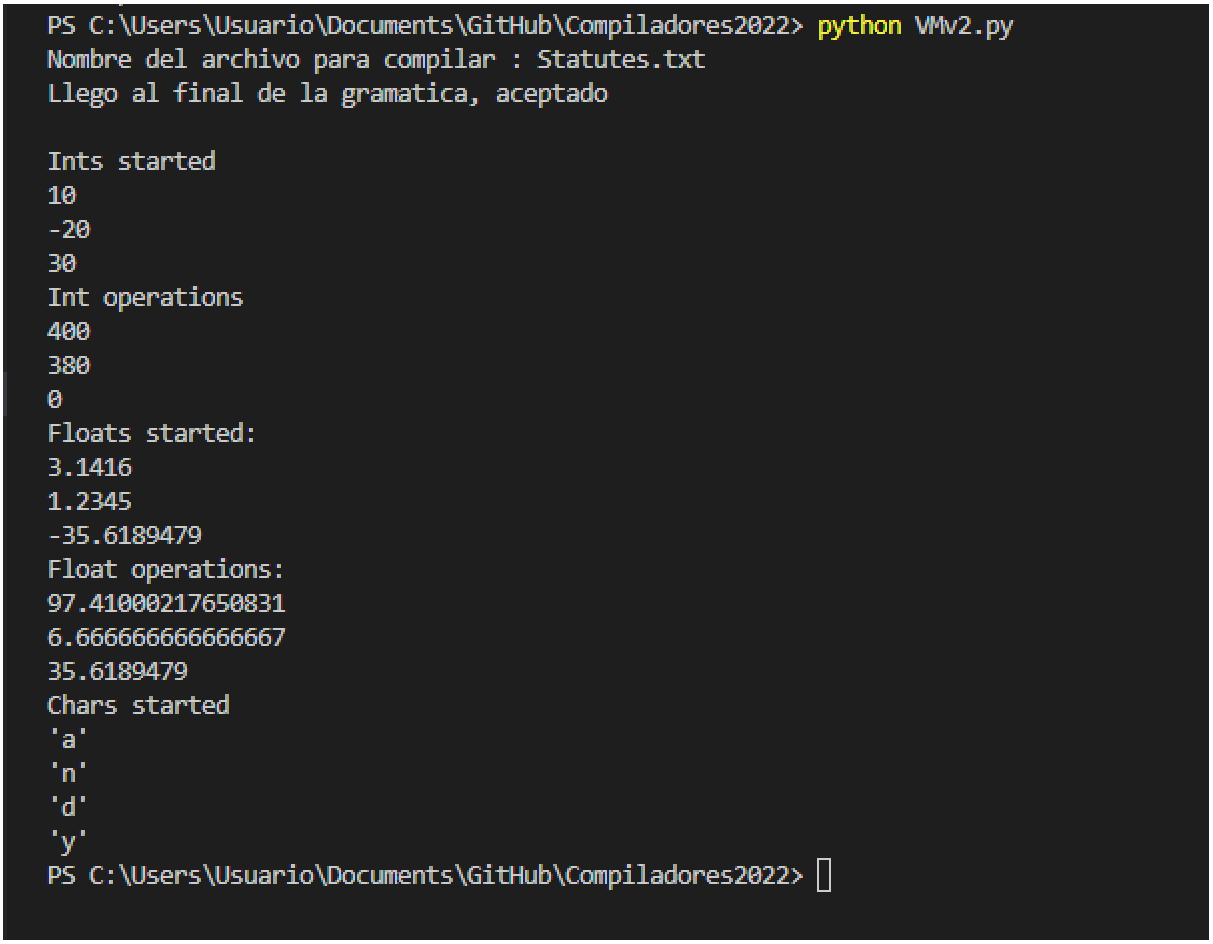


```
Program Statutes;
VARS
    int: a, b, c;
    float: d, e, f;
    char: g, h, i, j;

main(){
    a = 10;
    b = -20;
    c = 30;
    write("Ints started ", a , b , c);
    a = b * b;
    b = b + a;
    c = b / a;
    write("Int operations ", a, b, c);
    d = 3.1416;
    e = 1.2345;
    f = -35.6189479;
    write("Floats started: ", d, e, f);
    d = d*d*d*d;
    e = 20.0/3;
    f = f - (f*2);
    write("Float operations: ", d, e, f);
    g = 'a';
    h = 'n';
    i = 'd';
    j = 'y';
    write("Chars started", g, h, i, j);
}
```



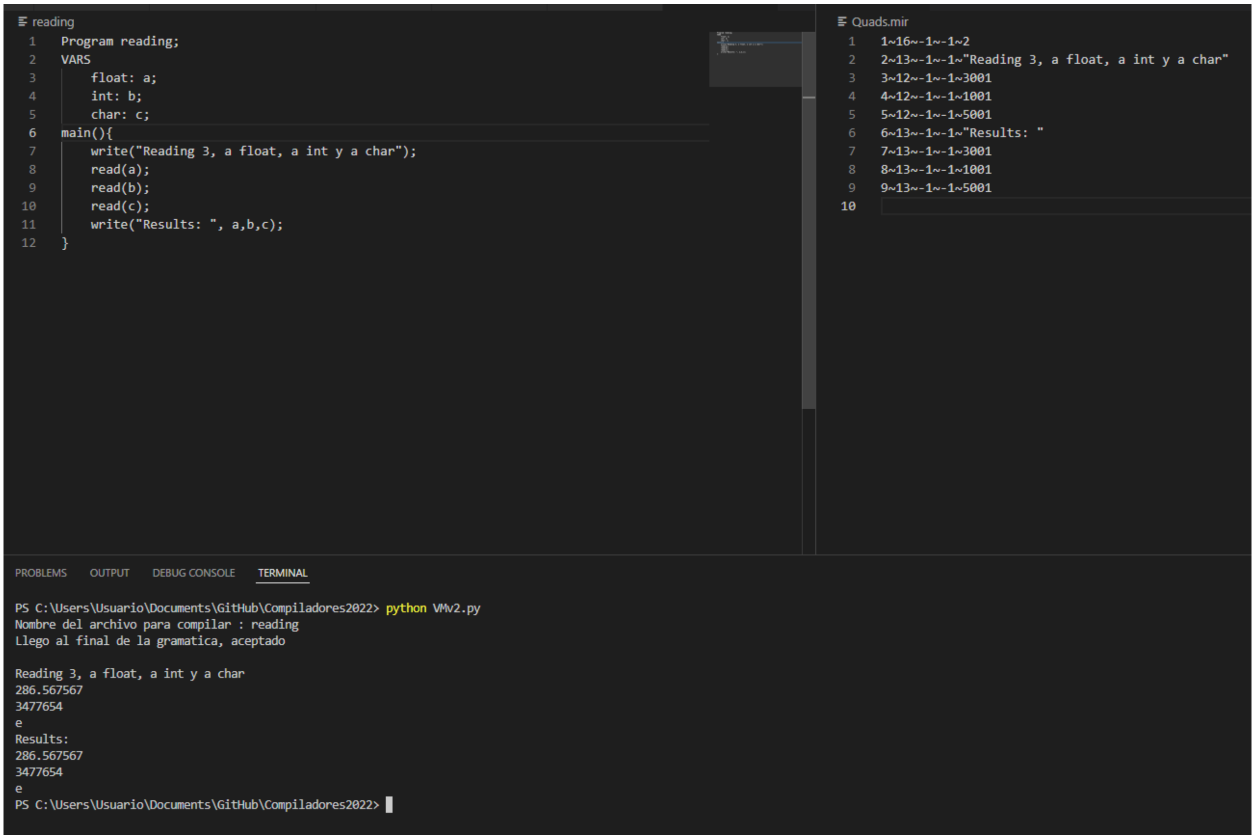
ads.mir

```
1~16~-1~-1~2
2~11~21002~-1~1001
3~11~21003~-1~1002
4~11~21004~-1~1003
5~13~-1~-1~"Ints started "
6~13~-1~-1~1001
7~13~-1~-1~1002
8~13~-1~-1~1003
9~3~1002~1002~13000
10~11~13000~-1~1001
11~1~1002~1001~13001
12~11~13001~-1~1002
13~4~1002~1001~13002
14~11~13002~-1~1003
15~13~-1~-1~"Int operations "
16~13~-1~-1~1001
17~13~-1~-1~1002
18~13~-1~-1~1003
19~11~23000~-1~3001
20~11~23001~-1~3002
21~11~23002~-1~3003
22~13~-1~-1~"Floats started: "
23~13~-1~-1~3001
24~13~-1~-1~3002
25~13~-1~-1~3003
26~3~3001~3001~15000
27~3~15000~3001~15001
28~3~15001~3001~15002
29~11~15002~-1~3001
30~4~23003~21005~15003
31~11~15003~-1~3002
32~3~3003~21006~15004
33~2~3003~15004~15005
34~11~15005~-1~3003
35~13~-1~-1~"Float operations: "
36~13~-1~-1~3001
37~13~-1~-1~3002
38~13~-1~-1~3003
39~11~25000~-1~5001
40~11~25001~-1~5002
41~11~25002~-1~5003
42~11~25003~-1~5004
43~13~-1~-1~"Chars started"
44~13~-1~-1~5001
45~13~-1~-1~5002
46~13~-1~-1~5003
47~13~-1~-1~5004
```



```
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> python VMv2.py
Nombre del archivo para compilar : Statutes.txt
Llego al final de la gramatica, aceptado

Ints started
10
-20
30
Int operations
400
380
0
Floats started:
3.1416
1.2345
-35.6189479
Float operations:
97.41000217650831
6.666666666666667
35.6189479
Chars started
'a'
'n'
'd'
'y'
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> 
```

Reading

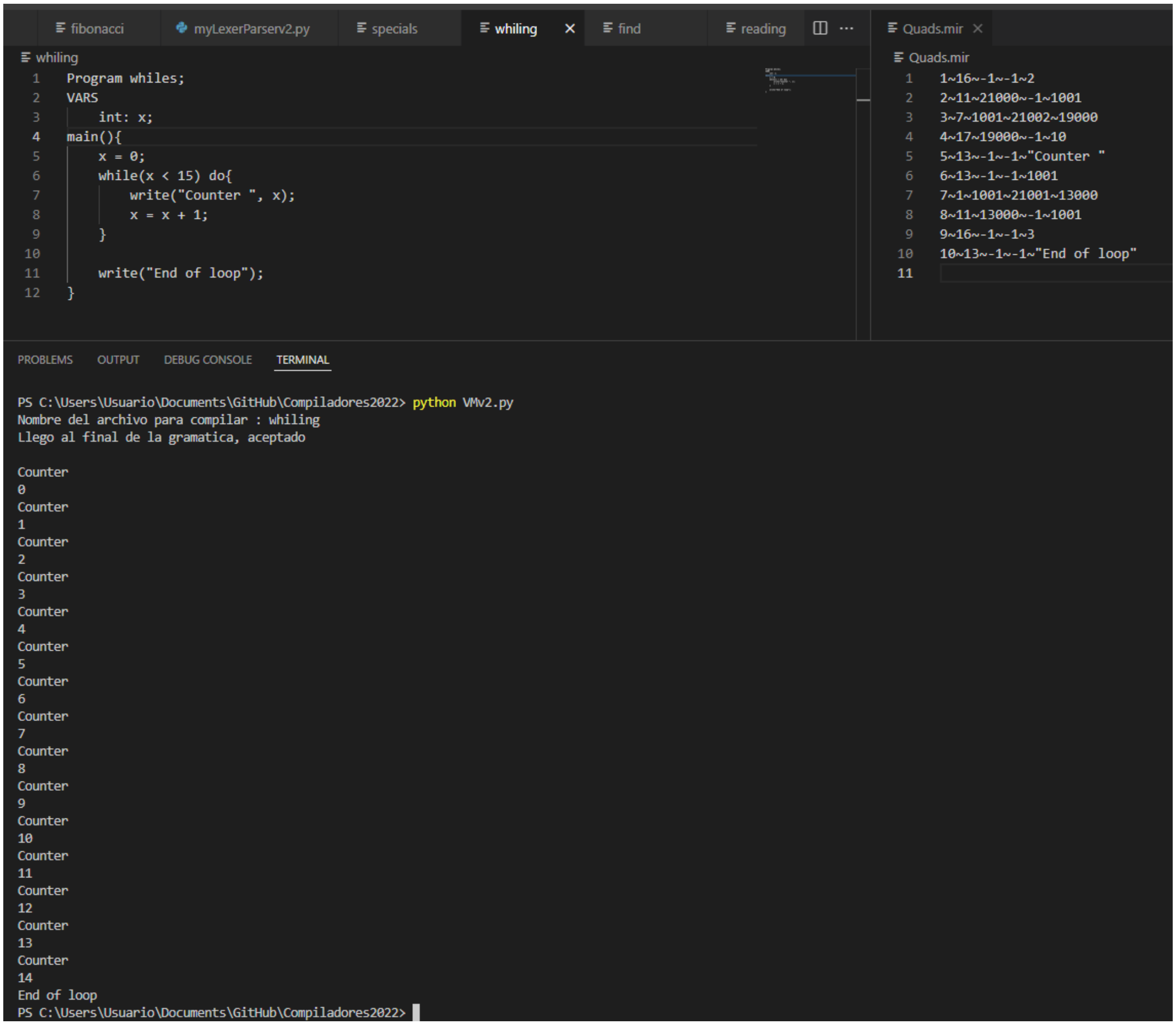


```
Program reading;
VARS
    float: a;
    int: b;
    char: c;
main(){
    write("Reading 3, a float, a int y a char");
    read(a);
    read(b);
    read(c);
    write("Results: ", a,b,c);
}
```

```
1~16~-1~-1~2
2~13~-1~-1~"Reading 3, a float, a int y a char"
3~12~-1~-1~3001
4~12~-1~-1~1001
5~12~-1~-1~5001
6~13~-1~-1~"Results: "
7~13~-1~-1~3001
8~13~-1~-1~1001
9~13~-1~-1~5001

```

```
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> python VMv2.py
Nombre del archivo para compilar : reading
Llego al final de la gramatica, aceptado

Reading 3, a float, a int y a char
286.567567
3477654
e
Results:
286.567567
3477654
e
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> 
```

Whiles:

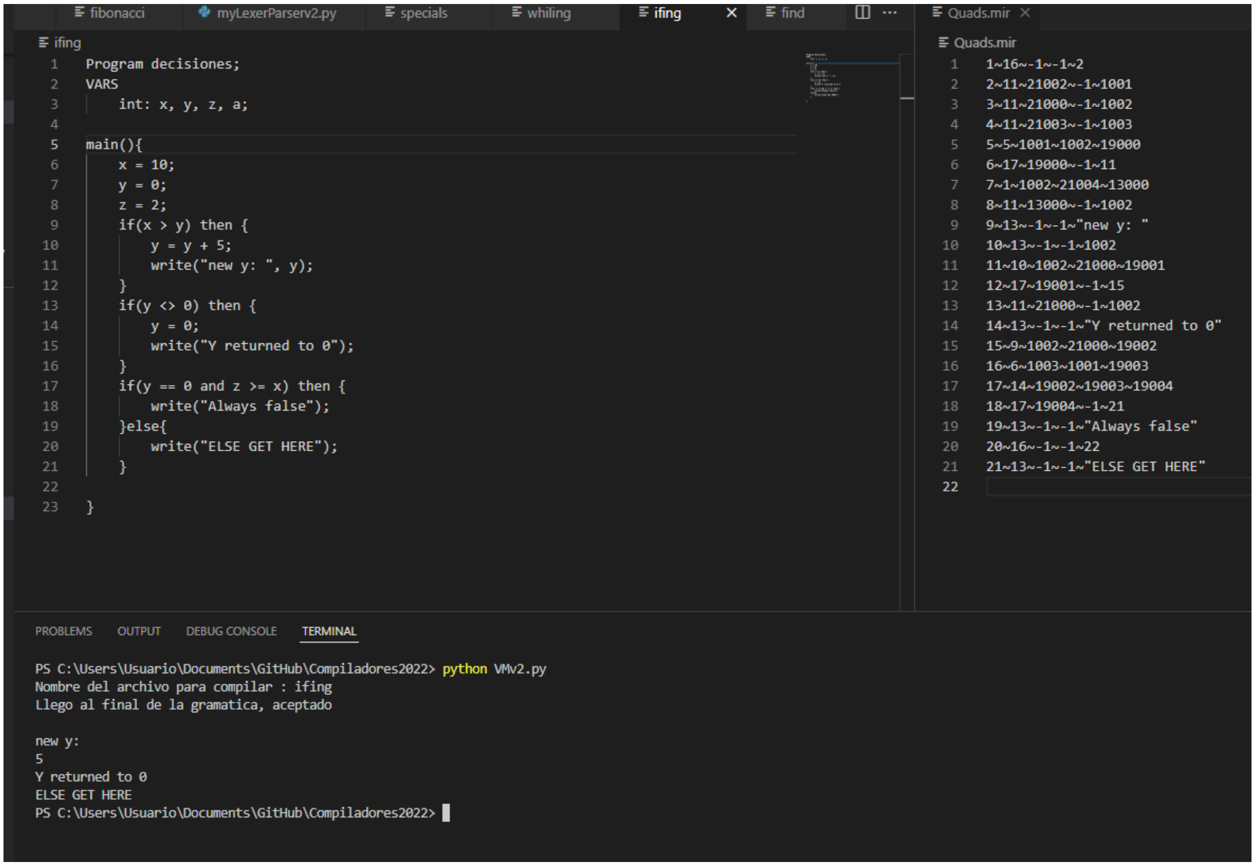


```
Program whiles;
VARS
    int: x;
main(){
    x = 0;
    while(x < 15) do{
        write("Counter ", x);
        x = x + 1;
    }

    write("End of loop");
}
```

```
1~16~-1~-1~2
2~11~21000~-1~1001
3~7~1001~21002~19000
4~17~19000~-1~10
5~13~-1~-1~"Counter "
6~13~-1~-1~1001
7~1~1001~21001~13000
8~11~13000~-1~1001
9~16~-1~-1~3
10~13~-1~-1~"End of loop"
```

```
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> python VMv2.py
Nombre del archivo para compilar : whiling
Llego al final de la gramatica, aceptado

Counter
0
Counter
1
Counter
2
Counter
3
Counter
4
Counter
5
Counter
6
Counter
7
Counter
8
Counter
9
Counter
10
Counter
11
Counter
12
Counter
13
Counter
14
End of loop
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022>
```

If:

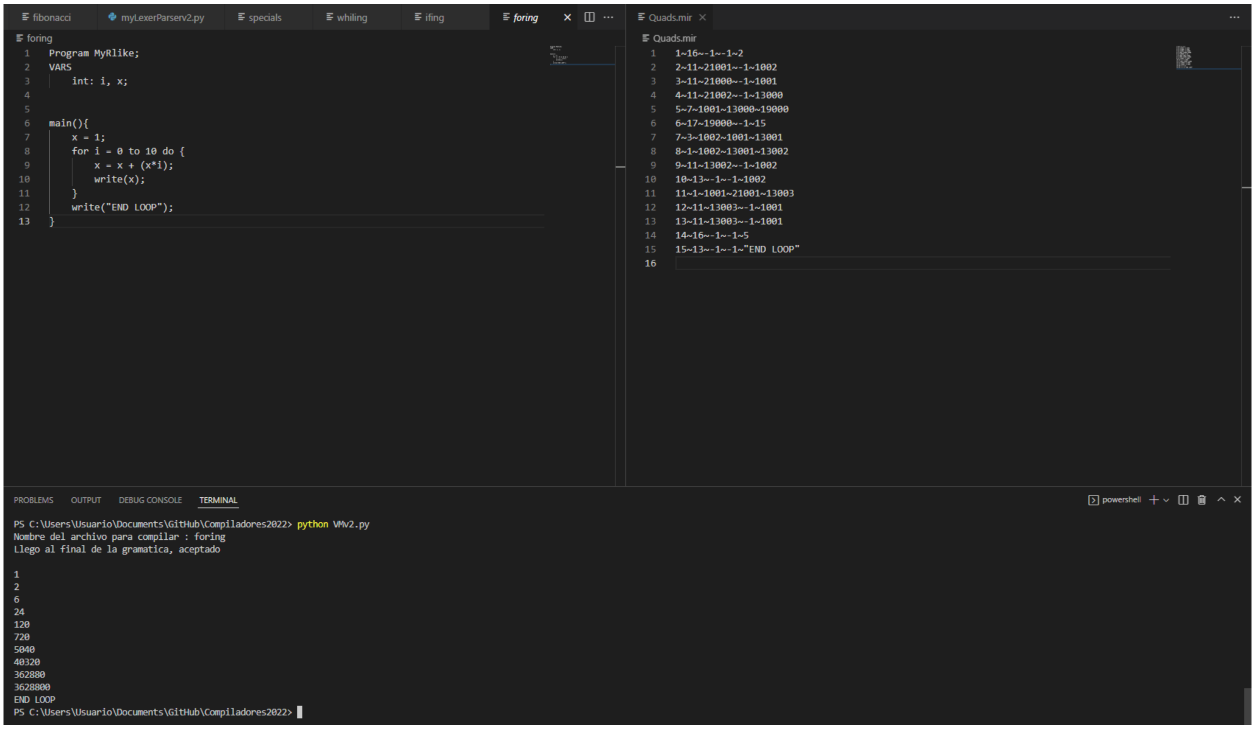


```
Program decisiones;
VARS
    int: x, y, z, a;

main(){
    x = 10;
    y = 0;
    z = 2;
    if(x > y) then {
        y = y + 5;
        write("new y: ", y);
    }
    if(y <> 0) then {
        y = 0;
        write("Y returned to 0");
    }
    if(y == 0 and z >= x) then {
        write("Always false");
    }else{
        write("ELSE GET HERE");
    }

}
```

```
1~16~-1~-1~2
2~11~21002~-1~1001
3~11~21000~-1~1002
4~11~21003~-1~1003
5~5~1001~1002~19000
6~17~19000~-1~11
7~1~1002~21004~13000
8~11~13000~-1~1002
9~13~-1~-1~"new y: "
10~13~-1~-1~1002
11~10~1002~21000~19001
12~17~19001~-1~15
13~11~21000~-1~1002
14~13~-1~-1~"Y returned to 0"
15~9~1002~21000~19002
16~6~1003~1001~19003
17~14~19002~19003~19004
18~17~19004~-1~21
19~13~-1~-1~"Always false"
20~16~-1~-1~22
21~13~-1~-1~"ELSE GET HERE"
```

```
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> python VMv2.py
Nombre del archivo para compilar : ifing
Llego al final de la gramatica, aceptado

new y:
5
Y returned to 0
ELSE GET HERE
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022>
```

Fors:

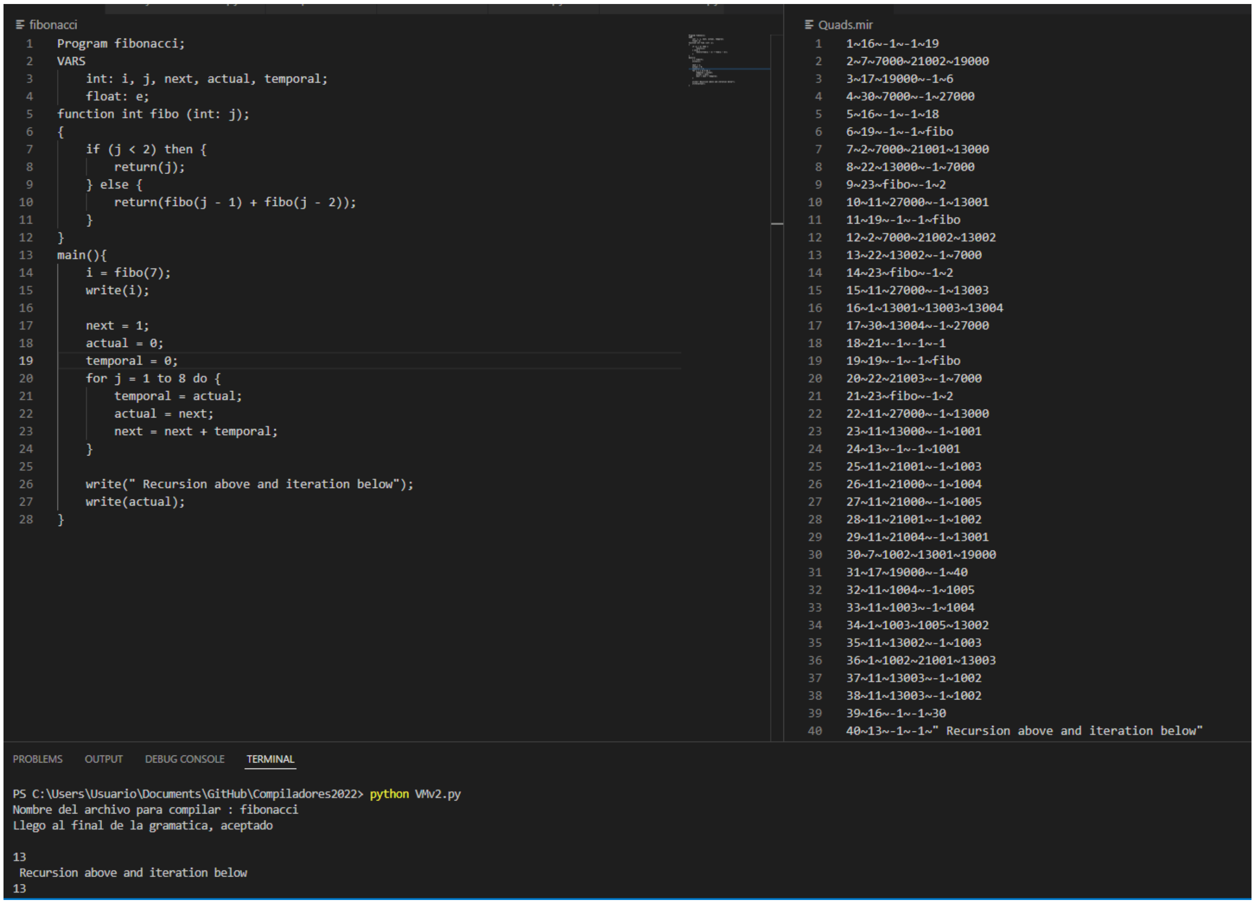


```
Program MyRlike;
VARS
    int: i, x;


main(){
    x = 1;
    for i = 0 to 10 do {
        x = x + (x*i);
        write(x);
    }
    write("END LOOP");
}
```

```
1~16~-1~-1~2
2~11~21001~-1~1002
3~11~21000~-1~1001
4~11~21002~-1~13000
5~7~1001~13000~19000
6~17~19000~-1~15
7~3~1002~1001~13001
8~1~1002~13001~13002
9~11~13002~-1~1002
10~13~-1~-1~1002
11~1~1001~21001~13003
12~11~13003~-1~1001
13~11~13003~-1~1001
14~16~-1~-1~5
15~13~-1~-1~"END LOOP"
```

```
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> python VMv2.py
Nombre del archivo para compilar : foring
Llego al final de la gramatica, aceptado

1
2
6
24
120
720
5040
40320
362880
3628800
END LOOP
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022>
```

Fibonacci:

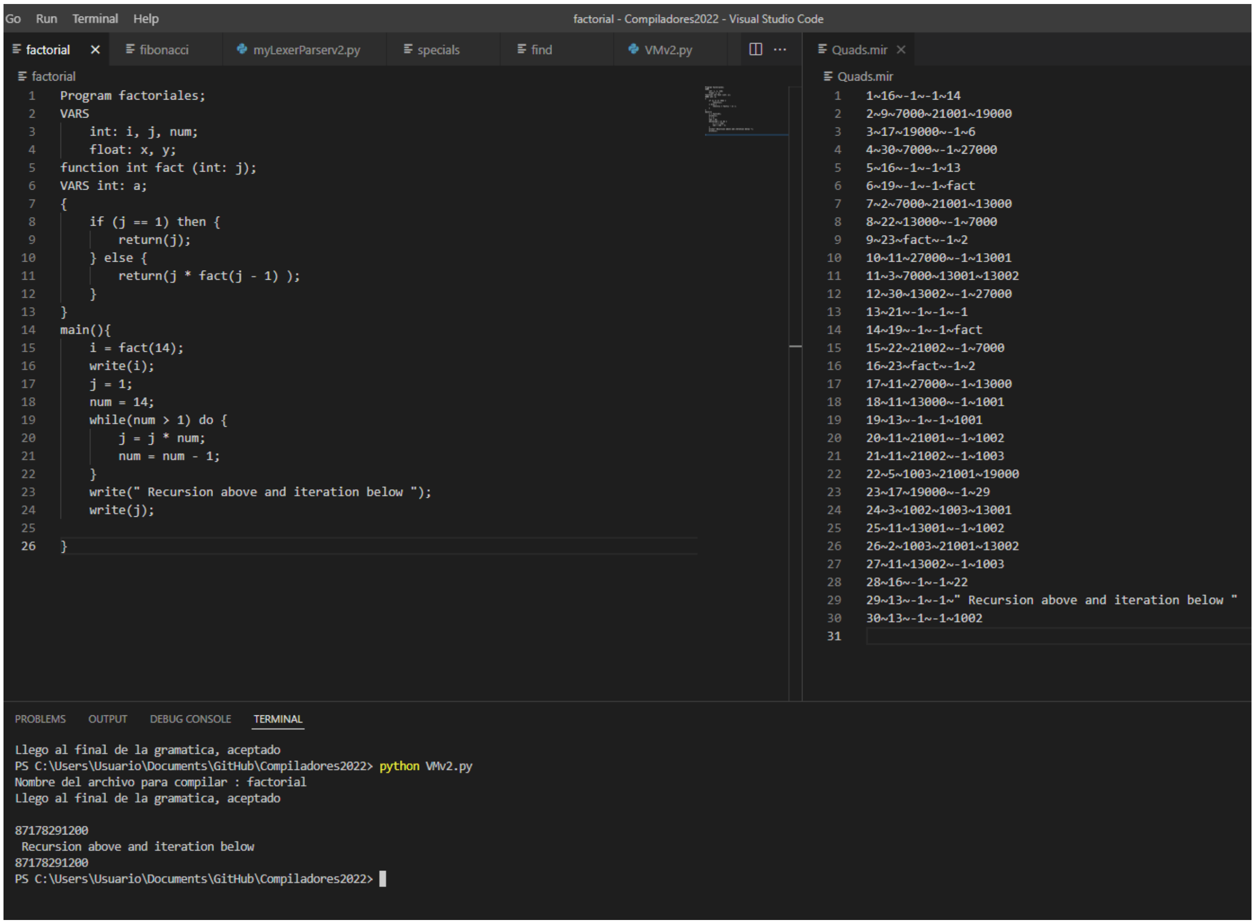


```
Program fibonacci;
VARS
    int: i, j, next, actual, temporal;
    float: e;
function int fibo (int: j);
{
    if (j < 2) then {
        return(j);
    } else {
        return(fibo(j - 1) + fibo(j - 2));
    }
}
main(){
    i = fibo(7);
    write(i);

    next = 1;
    actual = 0;
    temporal = 0;
    for j = 1 to 8 do {
        temporal = actual;
        actual = next;
        next = next + temporal;
    }

    write(" Recursion above and iteration below");
    write(actual);
}
```

```
1~16~-1~-1~19
2~7~7000~21002~19000
3~17~19000~-1~6
4~30~7000~-1~27000
5~16~-1~-1~18
6~19~-1~-1~fibo
7~2~7000~21001~13000
8~22~13000~-1~7000
9~23~fibo~-1~2
10~11~27000~-1~13001
11~19~-1~-1~fibo
12~2~7000~21002~13002
13~22~13002~-1~7000
14~23~fibo~-1~2
15~11~27000~-1~13003
16~1~13001~13003~13004
17~30~13004~-1~27000
18~21~-1~-1~-1
19~19~-1~-1~fibo
20~22~21003~-1~7000
21~23~fibo~-1~2
22~11~27000~-1~13000
23~11~13000~-1~1001
24~13~-1~-1~1001
25~11~21001~-1~1003
26~11~21000~-1~1004
27~11~21000~-1~1005
28~11~21001~-1~1002
29~11~21004~-1~13001
30~7~1002~13001~19000
31~17~19000~-1~40
32~11~1004~-1~1005
33~11~1003~-1~1004
34~1~1003~1005~13002
35~11~13002~-1~1003
36~1~1002~21001~13003
37~11~13003~-1~1002
38~11~13003~-1~1002
39~16~-1~-1~30
40~13~-1~-1~" Recursion above and iteration below"
```

```
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> python VMv2.py
Nombre del archivo para compilar : fibonacci
Llego al final de la gramatica, aceptado

13
 Recursion above and iteration below
13
```

Factorial:

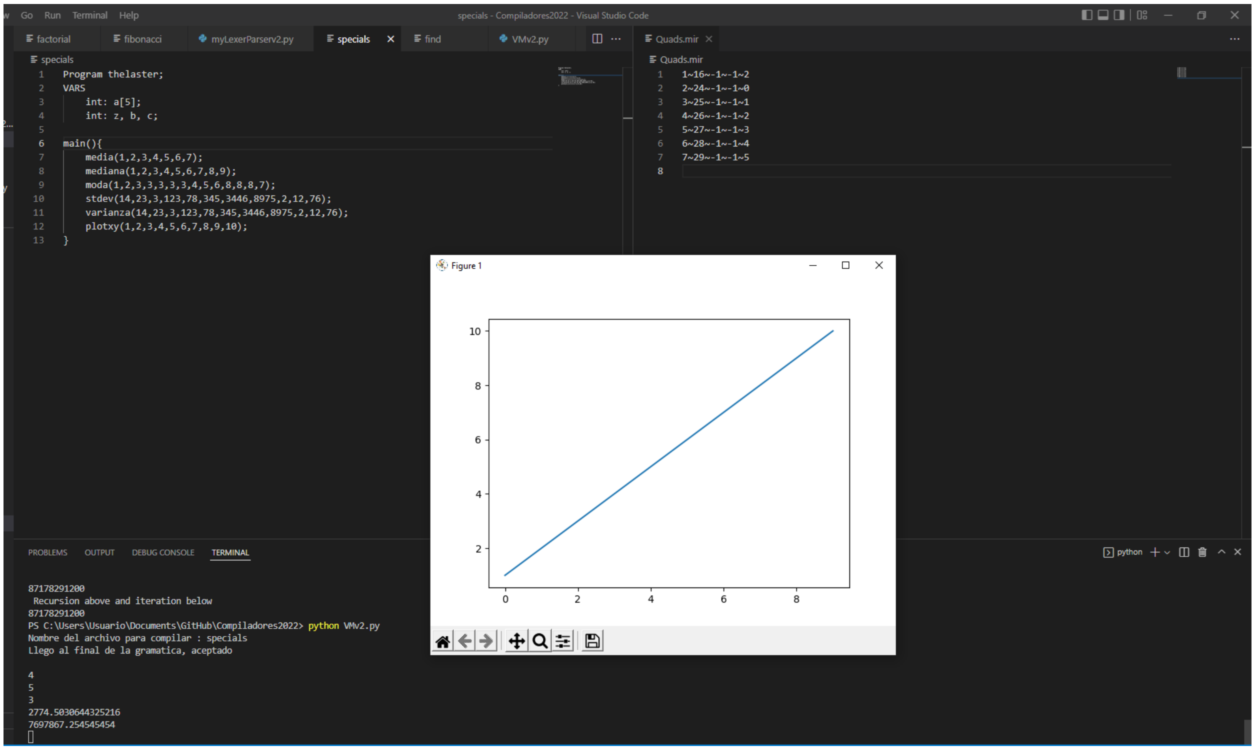


```
Program factoriales;
VARS
    int: i, j, num;
    float: x, y;
function int fact (int: j);
VARS int: a;
{
    if (j == 1) then {
        return(j);
    } else {
        return(j * fact(j - 1) );
    }
}
main(){
    i = fact(14);
    write(i);
    j = 1;
    num = 14;
    while(num > 1) do {
        j = j * num;
        num = num - 1;
    }
    write(" Recursion above and iteration below ");
    write(j);

}
```

```
1~16~-1~-1~14
2~9~7000~21001~19000
3~17~19000~-1~6
4~30~7000~-1~27000
5~16~-1~-1~13
6~19~-1~-1~fact
7~2~7000~21001~13000
8~22~13000~-1~7000
9~23~fact~-1~2
10~11~27000~-1~13001
11~3~7000~13001~13002
12~30~13002~-1~27000
13~21~-1~-1~-1
14~19~-1~-1~fact
15~22~21002~-1~7000
16~23~fact~-1~2
17~11~27000~-1~13000
18~11~13000~-1~1001
19~13~-1~-1~1001
20~11~21001~-1~1002
21~11~21002~-1~1003
22~5~1003~21001~19000
23~17~19000~-1~29
24~3~1002~1003~13001
25~11~13001~-1~1002
26~2~1003~21001~13002
27~11~13002~-1~1003
28~16~-1~-1~22
29~13~-1~-1~" Recursion above and iteration below "
30~13~-1~-1~1002
```

```
Llego al final de la gramatica, aceptado
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> python VMv2.py
Nombre del archivo para compilar : factorial
Llego al final de la gramatica, aceptado

87178291200
 Recursion above and iteration below
87178291200
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022>
```

Special methods:



```
Program thelaster;
VARS
    int: a[5];
    int: z, b, c;

main(){
    media(1,2,3,4,5,6,7);
    mediana(1,2,3,4,5,6,7,8,9);
    moda(1,2,3,3,3,3,3,4,5,6,8,8,8,7);
    stdev(14,23,3,123,78,345,3446,8975,2,12,76);
    varianza(14,23,3,123,78,345,3446,8975,2,12,76);
    plotxy(1,2,3,4,5,6,7,8,9,10);
}
```

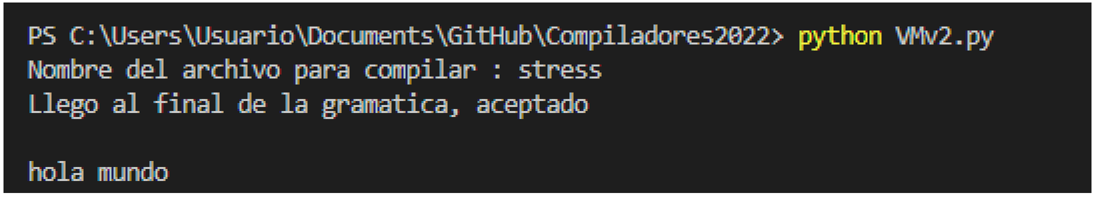
```
1~16~-1~-1~2
2~24~-1~-1~0
3~25~-1~-1~1
4~26~-1~-1~2
5~27~-1~-1~3
6~28~-1~-1~4
7~29~-1~-1~5
```

```
87178291200
 Recursion above and iteration below
87178291200
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> python VMv2.py
Nombre del archivo para compilar : specials
Llego al final de la gramatica, aceptado

4
5
3
2774.5030644325216
7697867.254545454
```

USER MANUAL

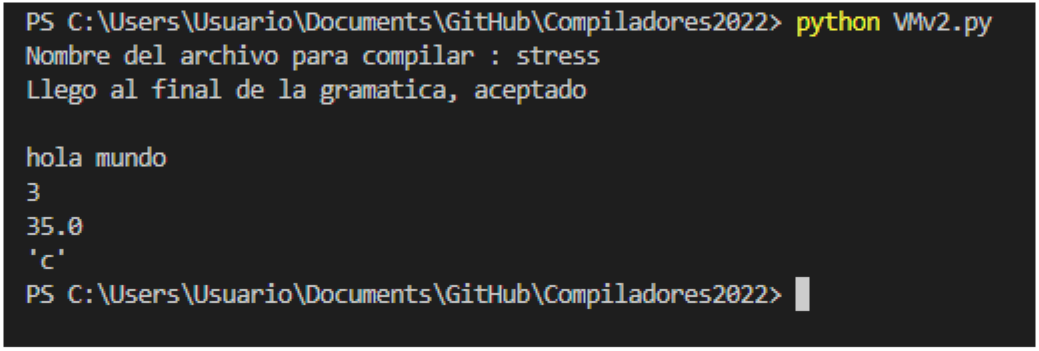
If you are looking to use this language as a learning tool or just to explore the efforts of a young IT engineer, you must follow the following steps:

1. Clone the project at the indicated repository: https://github.com/A00815749/Compiladores2022
2. Have your environment ready, which includes the requirement of python 3. (This software was mainly developed on Python 3.9.0)
3. Install the libraries that were used in the technical manual, in the Special Utilities section.
4. Navigate to the folder with your programming environment of choice. Make sure you have all the required files
5. On terminal execute in python the VMv2.py file



```
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> python VMv2.py
Nombre del archivo para compilar : stress
Llego al final de la gramatica, aceptado

hola mundo
```

6. When prompted with the following message, enter the name of the file you want to compile (make sure that the file is inside the same folder as the compiler)



```
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022> python VMv2.py
Nombre del archivo para compilar : stress
Llego al final de la gramatica, aceptado

hola mundo
3
35.0
'c'
PS C:\Users\Usuario\Documents\GitHub\Compiladores2022>
```

7. The file is going to be processed, and if working as intended, shall create the intermediate code machine, and display working results at terminal. If not, there are various error codes to see where the code went wrong if you are so interested.
8. Final note, you can directly see the source code of both the compiler and virtual machine, and if you are interested, at least 95% of the code is commented, so if you are curious, give it a read.

Video Demo file or link shall be directly appended to the project folder.