



# Tecnológico de Monterrey

**Campus Santa Fe**

*Generador de código*

**Nombre**

Ian Vázquez - A01027225


Mayo 29, 2025

## Introducción
















En este proyecto se eligió MIPS como lenguaje de destino para la generación de código, debido a su equilibrio entre simplicidad y realismo. A diferencia de lenguajes de máquina abstractos como Tiny Machine, MIPS es una arquitectura real basada en el paradigma RISC (Reduced Instruction Set Computer), lo que facilita tanto la comprensión del modelo computacional como la implementación de un generador de código eficiente. Su conjunto reducido de instrucciones, el uso exclusivo de operaciones entre registros, y una sintaxis clara y regular lo hace una plataforma accesible para estudiantes. Además, existen numerosos simuladores gratuitos como QtSpim y MARS que permiten ejecutar y depurar código MIPS sin necesidad de hardware físico y por ello MIPS se ha fue la opción preferida para la realización del proyecto

## Manual de usuario:

### 1.Descargar el zip llamado: Generador\_de\_codigo.zip

 Generador_de_codigo.zip	29/05/2025 10:46 p. m.	Carpeta comprimi...	39 KB
---	------------------------	---------------------	-------

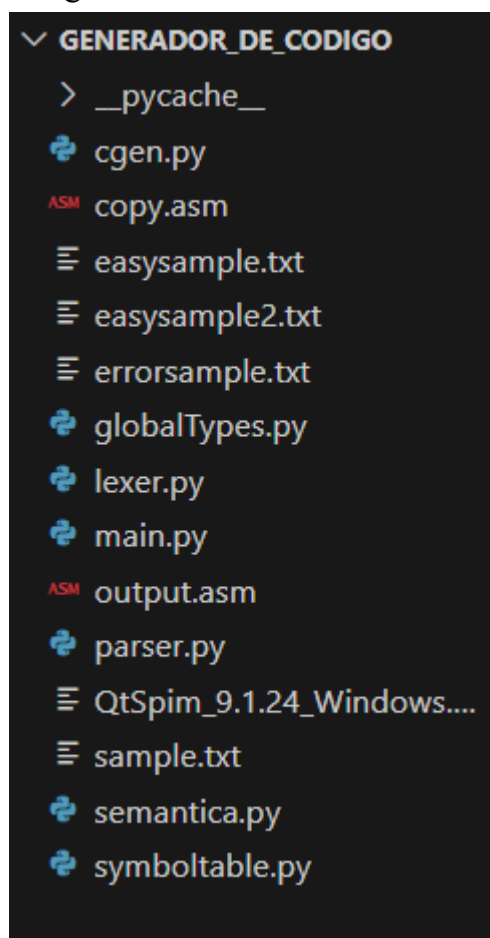
### 2.Descomprimir la carpeta zip y entrar a ella

	__pycache__	Carpeta de archivos					29/05/2025 10:42 p. m.
	cgen.py	Archivo de origen Python	4 KB	No	14 KB	78%	29/05/2025 10:42 p. m.
	copy.asm	Assembler Source	1 KB	No	2 KB	72%	28/05/2025 06:55 a. m.
	easysample.txt	Documento de texto	1 KB	No	1 KB	41%	29/05/2025 12:43 p. m.
	easysample2.txt	Documento de texto	1 KB	No	1 KB	46%	28/05/2025 02:24 p. m.
	errorsample.txt	Documento de texto	1 KB	No	1 KB	34%	20/05/2025 07:18 p. m.
	globalTypes.py	Archivo de origen Python	1 KB	No	2 KB	57%	07/05/2025 01:12 p. m.
	lexer.py	Archivo de origen Python	2 KB	No	8 KB	77%	11/05/2025 09:11 p. m.
	main.py	Archivo de origen Python	1 KB	No	1 KB	49%	29/05/2025 12:05 p. m.
	output.asm	Assembler Source	1 KB	No	2 KB	77%	29/05/2025 10:42 p. m.
	parser.py	Archivo de origen Python	4 KB	No	20 KB	81%	29/05/2025 12:12 p. m.
	QtSpim_9.1.24_Windows.msi	Paquete de Windows Insta...	20,409 KB	No	21,000 KB	3%	23/05/2025 03:10 p. m.
	sample.txt	Documento de texto	1 KB	No	1 KB	36%	29/05/2025 12:34 p. m.
	semantica.py	Archivo de origen Python	2 KB	No	5 KB	70%	25/05/2025 03:27 p. m.
	symboltable.py	Archivo de origen Python	1 KB	No	3 KB	65%	25/05/2025 03:21 p. m.

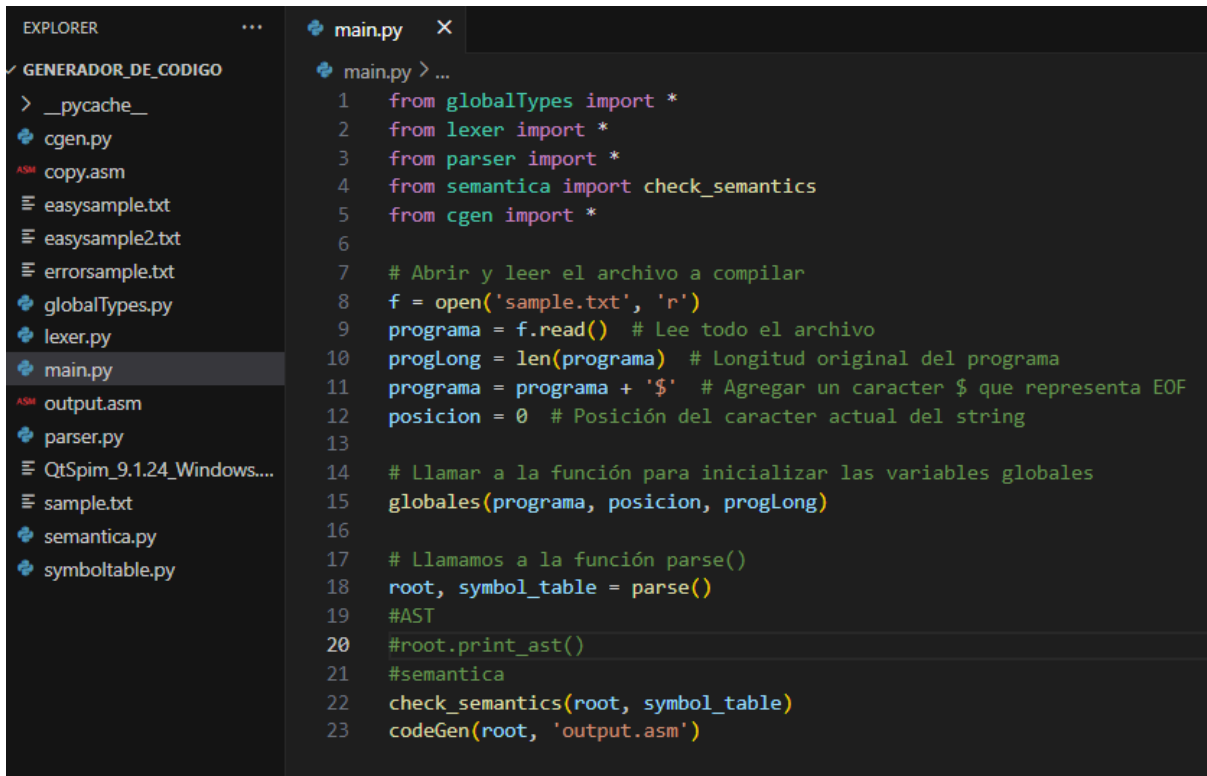
3. Una vez dentro de la carpeta Generador\_de\_codigo se debe descargar QtSpim desde el archivo AtSpim\_9.1.24\_Windows.msi (en caso de no tenerlo o no tener un simulador de mips semejante) (Si se está ejecutando en un sistema IOs buscar la version correspondiente)



4. Una vez obtenido QtSpim (o un semejante) se debe abrir la carpeta en un editor de código como Visual Studio Code



## 5. Ingresar a main.py



The screenshot shows a code editor with two panels. The left panel, titled 'EXPLORER', displays a file tree under the folder 'GENERADOR\_DE\_CODIGO'. The files listed are: '\_\_pycache\_\_', 'cgen.py', 'copy.asm', 'easysample.txt', 'easysample2.txt', 'errorsample.txt', 'globalTypes.py', 'lexer.py', 'main.py' (which is selected and highlighted), 'output.asm', 'parser.py', 'QtSpim\_9.1.24\_Windows....', 'sample.txt', 'semantica.py', and 'symboltable.py'. The right panel shows the code for 'main.py'. The code imports modules from 'globalTypes', 'lexer', 'parser', 'semantica', and 'cgen'. It then opens 'sample.txt', reads its content, appends an EOF character '\$', and calls functions to initialize global variables, parse the code, print the AST, check semantics, and generate assembly code.

```
main.py > ...
1  from globalTypes import *
2  from lexer import *
3  from parser import *
4  from semantica import check_semantics
5  from cgen import *
6
7  # Abrir y leer el archivo a compilar
8  f = open('sample.txt', 'r')
9  programa = f.read() # Lee todo el archivo
10 progLong = len(programa) # Longitud original del programa
11 programa = programa + '$' # Agregar un caracter $ que representa EOF
12 posicion = 0 # Posición del caracter actual del string
13
14 # Llamar a la función para inicializar las variables globales
15 globales(programa, posicion, progLong)
16
17 # Llamamos a la función parse()
18 root, symbol_table = parse()
19 #AST
20 #root.print_ast()
21 #semantica
22 check_semantics(root, symbol_table)
23 codeGen(root, 'output.asm')
```

6. En este manual se trabajara con el archivo que viene por defecto al descargar el proyecto llamado sample.txt que es un supuesto código de c-, en caso de querer cambiarlo a otro archivo se necesita poner el archivo a ejecutar en la carpeta de Generador\_de\_codigo y cambiar el nombre del archivo sample.txt al nombre de su archivo

```
# Abrir y leer el archivo a compilar
f = open('sample.txt', 'r')
programa = f.read() # Lee todo el archivo
progLong = len(programa) # Longitud original del programa
programa = programa + '$' # Agregar un caracter $ que representa EOF
posicion = 0 # Posición del caracter actual del string
```

7. Una vez colocado el código a ejecutar se debe presionar el botón de correr



8. Una vez ejecutado el código es posible ver en la terminal la tabla de símbolos del archivo ejecutado, la asignación de los espacios en cada una de las funciones y todos los errores que hubo en el proceso

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\hp\Documents\progra\Regular\Generador_de_codigo> ^C
PS C:\Users\hp\Documents\progra\Regular\Generador_de_codigo> & C:/Us
Tabla de símbolos: global
input (FUNCTION, INT) en línea 1
output (FUNCTION, VOID) en línea 1
gcd (GLOBAL_VAR, VOID) en línea 3
main (GLOBAL_VAR, VOID) en línea 10
Tabla de símbolos: gcd
u (LOCAL_VAR, INT) en línea 3
v (LOCAL_VAR, INT) en línea 3
Tabla de símbolos: main
x (LOCAL_VAR, INT) en línea 12
y (LOCAL_VAR, INT) en línea 12
Buscando tabla hija para función 'gcd' en ámbito 'global'
Buscando tabla hija para función 'main' en ámbito 'global'
No se encontraron errores semánticos.
symbol_table en gcd -> {'u': 8, 'v': 12}
symbol_table en main -> {'x': -4, 'y': -8, 'output': -12}
PS C:\Users\hp\Documents\progra\Regular\Generador_de_codigo>
```

9. Al ejecutar el código se creará una capeta con un código .asm equivalente al c-  
seleccionado en el paso 6 parecido al que se muestra a continuación:

EXPLORER \*\*\*

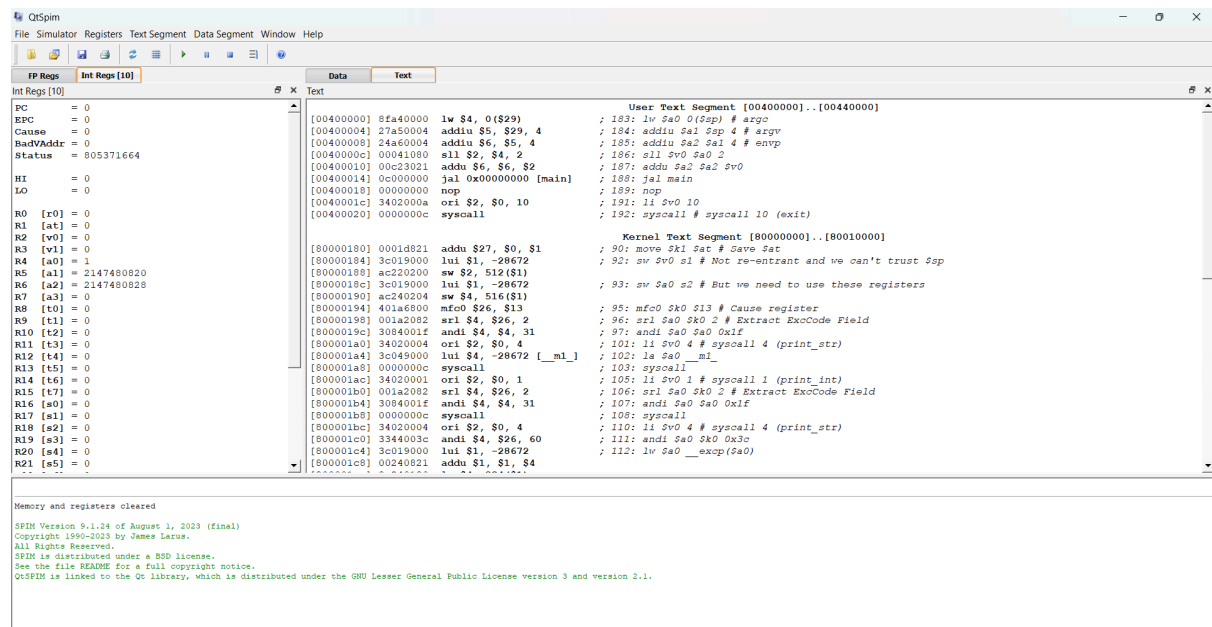
main.py output.asm X

GENERADOR\_DE\_CODIGO

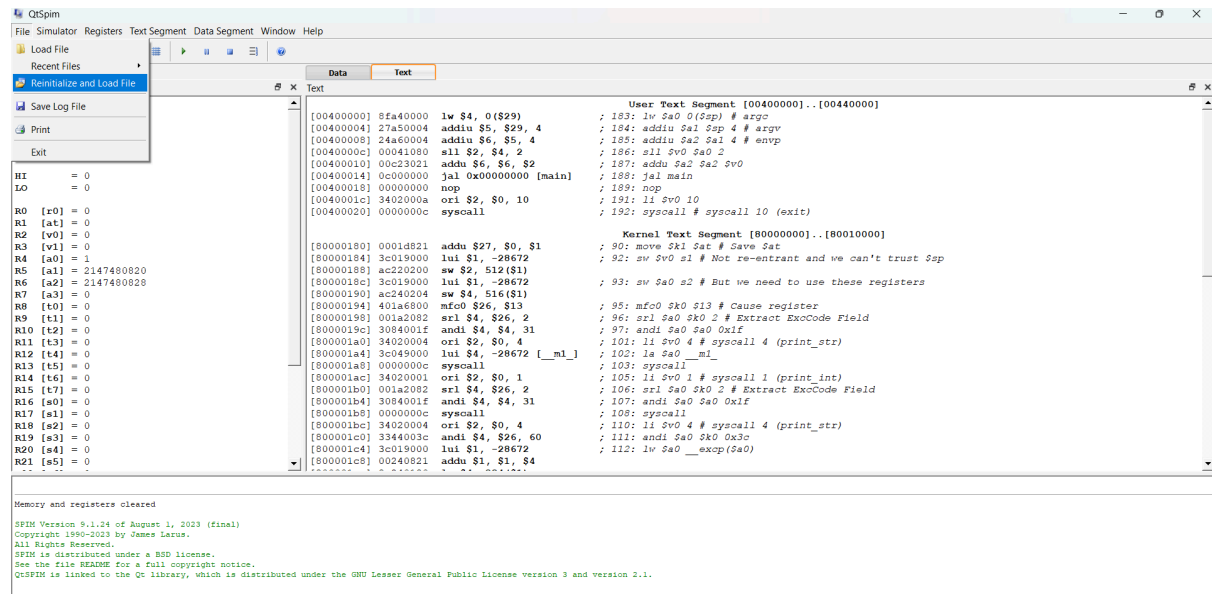
> \_\_pycache\_\_  
 cgen.py  
 easysample.txt  
 easysample2.txt  
 errorsample.txt  
 globalTypes.py  
 lexer.py  
 main.py  
 output.asm  
 parser.py  
 QtSpim\_9.1.24\_Windows....  
 sample.txt  
 semantica.py  
 symboltable.py

ASM output.asm  
 1 .globl gcd  
 2 .globl main  
 3 .data  
 4 .text  
 5 gcd:  
 6     addiu \$sp, \$sp, -8  
 7     sw \$ra, 4(\$sp)  
 8     sw \$fp, 0(\$sp)  
 9     addiu \$fp, \$sp, 0  
 10    lw \$t0, 12(\$fp)  
 11    addiu \$sp, \$sp, -4  
 12    sw \$t0, 0(\$sp)  
 13    li \$t0, 0  
 14    lw \$t1, 0(\$sp)  
 15    addiu \$sp, \$sp, 4  
 16    xor \$t0, \$t0, \$t1  
 17    sltu \$t0, \$t0, 1  
 18    beqz \$t0, else\_2683856649392  
 19    lw \$t0, 8(\$fp)  
 20    move \$v0, \$t0  
 21    j gcd\_epilogue  
 22    j endif\_2683856649392  
 23 else\_2683856649392:  
 24    lw \$t0, 8(\$fp)  
 25    addiu \$sp, \$sp, -4  
 26    sw \$t0, 0(\$sp)  
 27    lw \$t0, 8(\$fp)  
 28    addiu \$sp, \$sp, -4

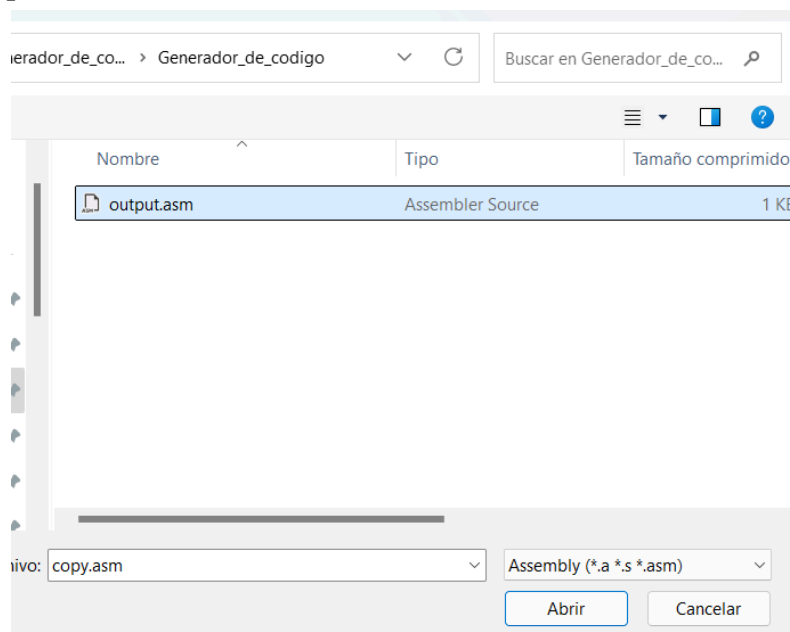
10. Se abre QtSpim o equivalente:



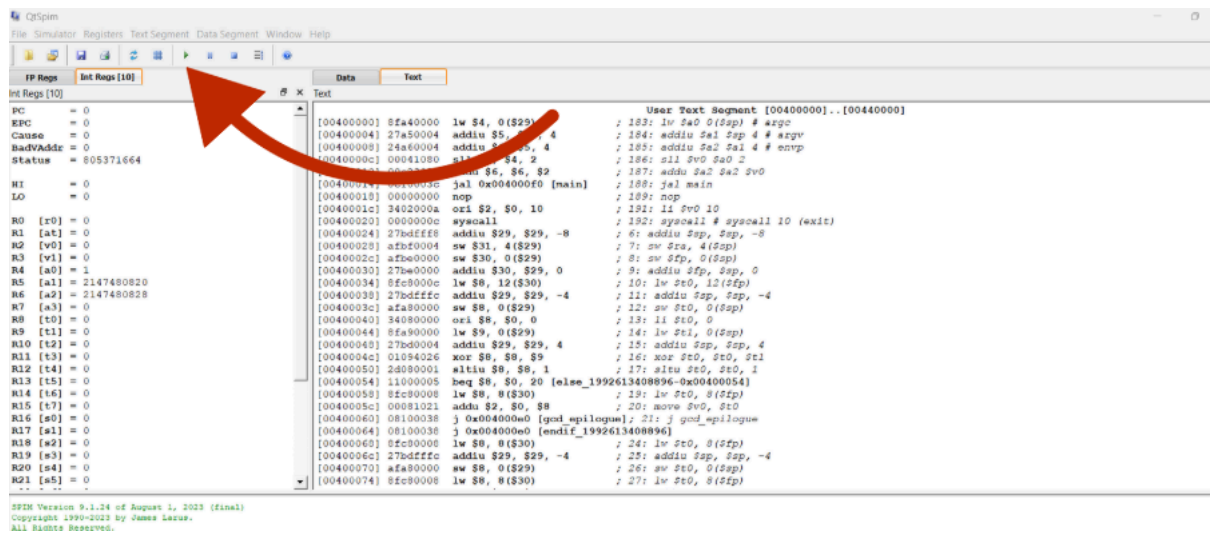
11. Siempre para ejecutar cualquier código se debe ir a la sección de Reinitialize and load file



12. Buscar la carpeta `Generador_de_codigo` en el buscador de archivos de Qtspim e ingresar a ella, una vez dentro de ella seleccionar el archivo llamado `output.asm` y presionar abrir



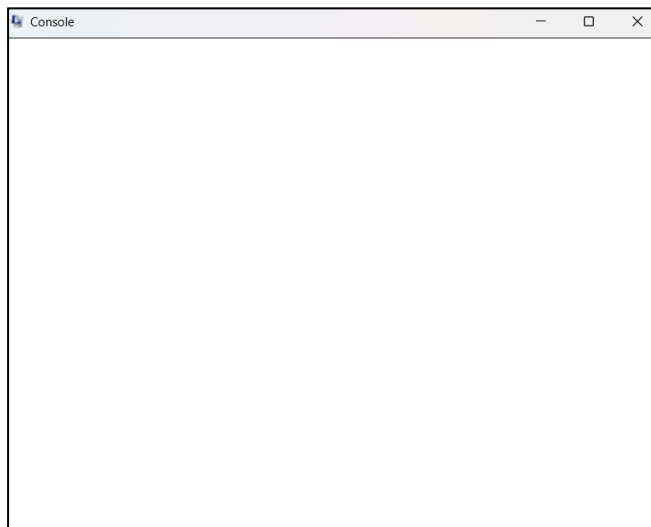
13. Presionar el botón de run



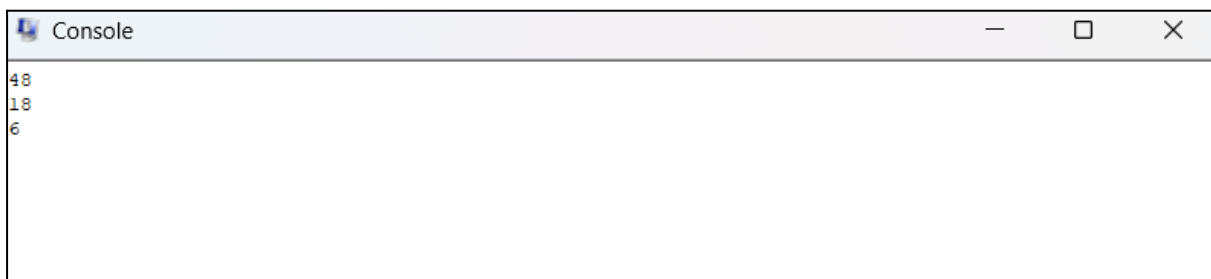
14. Se ejecutará el código querido.

## Notas:

1. En caso de que dentro pasado a main tenga una función input (Se espera que el usuario ingrese un valor), se mostrará de la siguiente manera:



No está mal, únicamente está esperando un valor lo cual se puede ingresar con el teclado y después presionar enter:





2. Se generara el código de ensamblador siempre que se pueda aunque existan errores léxicos o semánticos, por lo cual es recomendable seguir las reglas del lenguaje c- y también revisar la respuesta de la consola al ejecutar main para ver si existen errores en el código proporcionado

## Apéndice

### Analizador Semántico

#### 1. Reglas Lógicas de Inferencia de Tipos

##### 1.1 Uso de variables no declaradas

Regla lógica:

Si un identificador aparece como variable en una expresión (factor), debe haber sido previamente declarado en el ámbito actual o en alguno de sus ancestros.

Código aplicado:

```
if node.kind == "factor":
    if node.name is not None and not node.name.isdigit():
        symbol = lookup(current_table, node.name)
        if symbol is None:
            errors.append(...)
```

##### 1.2 Cada función tiene su propia tabla de símbolos

Regla lógica:

Cuando se declara una función, se debe crear una tabla de símbolos hija con su nombre, donde se almacenan sus parámetros y variables locales.

Código aplicado:

```
elif node.kind == "fun_declaration":
    child_table = current_table.get_child_by_name(node.name)
    if child_table is None:
        errors.append(...)
```

##### 1.3 Llamadas solo a funciones declaradas

Regla lógica:

No se puede invocar una función que no ha sido declarada previamente.

Código aplicado:

```
function_entry = lookup(current_table, function_name)
if function_entry is None:
    errors.append(...)
```

## 1.4 Coincidencia de número de argumentos

Regla lógica:

Una función debe ser llamada con el mismo número de argumentos que el número de parámetros que se declararon.

Código aplicado:

```
if num_declared != num_passed:
    errors.append(...)
```

## 2. Estructura de la Tabla de Símbolos

El analizador utiliza una estructura de árbol para representar los distintos ámbitos del programa. Cada función tiene su propia tabla de símbolos hija, y existe una tabla principal para el ámbito global.

La clase SymbolTable tiene los siguientes atributos:

- scope\_name: nombre del ámbito.
- symbols: diccionario de símbolos del ámbito actual.
- children: lista de tablas hijas (funciones).
- parent: referencia al ámbito padre.

La búsqueda de símbolos se hace de forma jerárquica hacia arriba (lookup):

```
def lookup(table, name):
    while table is not None:
        if name in table.symbols:
            return table.symbols[name]
        table = table.parent
    return None
```

Esto permite visibilidad de variables entre ámbitos según las reglas del lenguaje.

## Gramática:

program → declaration\_list  
declaration\_list → declaration declaration\_list | ε  
declaration → var\_declaration | fun\_declaration  
var\_declaration → type\_specifier ID var\_declaration'  
var\_declaration' → ';' | '[' NUM ']' ';' ;  
type\_specifier → 'int' | 'void'  
fun\_declaration → type\_specifier ID '(' params ')' compound\_stmt  
params → param\_list | 'void'  
param\_list → param param\_list'  
param\_list' → ',' param param\_list' | ε  
param → type\_specifier ID param'  
param' → '[' ']' | ε  
compound\_stmt → '{' local\_declarations statement\_list '}'  
local\_declarations → local\_declarations var\_declaration | ε  
statement\_list → statement\_list statement | ε  
statement → expression\_stmt  
              | compound\_stmt  
              | selection\_stmt  
              | iteration\_stmt  
              | return\_stmt  
expression\_stmt → expression ';' | ';' ;  
selection\_stmt → 'if' '(' expression ')' statement ('else' statement)?  
iteration\_stmt → 'while' '(' expression ')' statement  
return\_stmt → 'return' return\_stmt'  
return\_stmt' → ';' | expression ';' ;  
expression → var '=' expression | simple\_expression  
var → ID var'  
var' → '[' expression ']' | ε  
simple\_expression → additive\_expression relop additive\_expression | additive\_expression  
relop → '<=' | '<' | '>' | '>=' | '==' | '!=' ;  
additive\_expression → term additive\_expression'  
additive\_expression' → ('+' | '-') term additive\_expression' | ε  
term → factor term'  
term' → ('\*' | '/') factor term' | ε  
factor → '(' expression ')' | var | call | NUM  
call → ID '(' args ')' ;  
args → arg\_list | ε  
arg\_list → expression arg\_list'  
arg\_list' → ',' expression arg\_list' | ε

## Expresiones regulares:

1. Identificadores (ID)

[a|b|...|z|A|B|...|Z]<sup>+</sup>

2. Números (NUM)

[0|1|...|9]<sup>+</sup>

3. IF

^if\$

4. WHILE

^while\$

5. ELSE

^else\$

6. INT

^int\$

7. RETURN

^return\$

8. VOID

^void\$

9. PLUS

'+'

10. MINUS

-

11. TIMES

'\*'

12. DIVIDE

/

13. LT

<

14. LTEQ

<=

15. GT

>

16. GTEQ

>=

17. ASSIGN

=

18. EQ

==

19. NEQ

!=

20. SEMI

;

21. COMMA

,

22. LPAREN

'('

23. RPAREN

)'

- 24. LBRANCE  
  {
- 25. RBRANCE  
  }
- 26. LBRACKET  
  '[
- 27. RBRACKET  
  ']
- 28. COMMENT  
  /'\*['^']\*'\*'+(?:['^/\*']['^']\*'\*'+)\*/
- 29. SPACE  
  ['\t']+
- 30. ENDLINE  
  \n
- 31. ERRORES ID  
  [a|b|...|z|A|B|...|Z|]+[0|1|...|9]
- 32. ERRORES NUM  
  [0|1|...|9|]+[a|b|...|z|A|B|...|Z]

## DFA

Por el tamaño resultante del dfa se prefirió dar el enlace para que se pueda analizar cómodamente, pudiendo navegar en cualquier parte del dfa y si se requiere tener más detalle poder hacer zoom

[https://www.canva.com/design/DAGkM1Auf6w/FbQcqKNtQ7KNan3giXiwjw/edit?utm\\_content=DAGkM1Auf6w&utm\\_campaign=designshare&utm\\_medium=link2&utm\\_source=sharebutton](https://www.canva.com/design/DAGkM1Auf6w/FbQcqKNtQ7KNan3giXiwjw/edit?utm_content=DAGkM1Auf6w&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton)