



**Instituto Tecnológico de Estudios Superiores de Monterrey**  
**Campus Monterrey**

**Proyecto Compiladores**

**Profesora:**

Elda Guadalupe Quiroga González

Carmina López Palacios

A00830265

**Versión 3.0**  
4 de mayo, 2024

## ÍNDICE

<b>I. Gramática Formal y Expresiones Regulares.....</b>	<b>1</b>
<b>Lista de tokens.....</b>	<b>1</b>
Operadores.....	1
Separadores.....	1
Delimitadores.....	1
Palabras reservadas.....	2
<b>Expresiones regulares.....</b>	<b>2</b>
<b>Gramáticas formales.....</b>	<b>3</b>
PROGRAMA.....	3
VARS.....	3
TYPE.....	4
BODY.....	4
STATEMENT.....	4
PRINT.....	4
ASSIGN.....	4
CTE.....	5
EXPRESIÓN.....	5
EXP.....	5
TÉRMINO.....	5
FACTOR.....	6
F_CALL.....	6
CYCLE.....	6
CONDITION.....	6
FUNCS.....	6
<b>II. Léxico y Sintaxis.....</b>	<b>7</b>
Investigación herramientas.....	7
Implementación.....	8
Test-Plan.....	9
<b>III. Semántica de variables.....</b>	<b>10</b>
Cubo semántico.....	10
Directorio de Funciones.....	11
Tablas de Variables.....	12
Diccionario cte int, float y string.....	13
<b>IV. Código Intermedio.....</b>	<b>15</b>
Pilas y Fila.....	15
Puntos neurálgicos.....	16
<b>V. Direcciones de memoria.....</b>	<b>21</b>
<b>VI. Ejecución - Máquina virtual.....</b>	<b>21</b>
Mapa de memoria.....	21
Procesador (CPU).....	22

IDE.....	22
<b>VII. Tests.....</b>	<b>23</b>
<b>Liga del repositorio.....</b>	<b>35</b>
<b>Referencias.....</b>	<b>36</b>

## I. Gramática Formal y Expresiones Regulares

### Lista de tokens

#### *Operadores*

Nombre	Símbolo
Equal	=
Greater than	>
Less than	<
Not equal	!=
Plus	+
Minus	-
Multiplication	*
Division	/

#### *Separadores*

Nombre	Símbolo
Comma	,
Semicolon	;
Colon	:

#### *Delimitadores*

Nombre	Símbolo
Left brace	{
Right brace	}
Left parenthesis	(
Right parenthesis	)
Left bracket	[
Right bracket	]

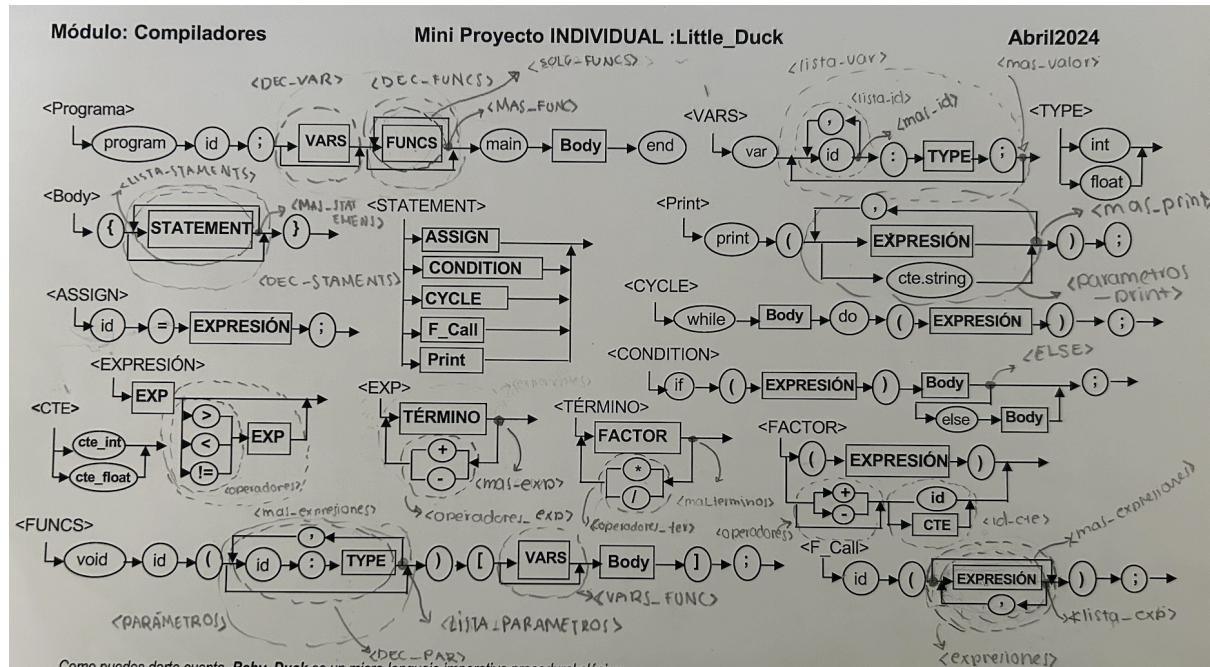
## *Palabras reservadas*

```
Unset
program
main
end
void
var
print
while
do
if
else
int
float
```

## Expresiones regulares

Elementos	REGEX
id	$^{\wedge}[a-z][a-zA-Z0-9_]*$
cte_int	$-? [0-9]^+$
cte_float	$-? [0-9]^+ \cdot [0-9]^+$
cte_string	"[^"]* "

## Gramáticas formales



## PROGRAMA

Unset

$\langle \text{PROGRAMA} \rangle \rightarrow \text{program id} ; \langle \text{DEC_VARS} \rangle \langle \text{DEC_FUNCS} \rangle \text{ main } \langle \text{BODY} \rangle$   
 end  
 $\langle \text{DEC_VARS} \rangle \rightarrow \epsilon$   
 $\langle \text{DEC_VARS} \rangle \rightarrow \langle \text{VARS} \rangle$   
 $\langle \text{SOLO_FUNCS} \rangle \rightarrow \langle \text{FUNCS} \rangle \langle \text{MAS_FUNCS} \rangle$   
 $\langle \text{MAS_FUNCS} \rangle \rightarrow \epsilon$   
 $\langle \text{MAS_FUNCS} \rangle \rightarrow \langle \text{SOLO_FUNCS} \rangle$   
 $\langle \text{DEC_FUNCS} \rangle \rightarrow \epsilon$   
 $\langle \text{DEC_FUNCS} \rangle \rightarrow \langle \text{SOLO_FUNCS} \rangle$

## VARS

Unset

$\langle \text{VARS} \rangle \rightarrow \text{var } \langle \text{LISTA_VAR} \rangle$   
 $\langle \text{LISTA_VAR} \rangle \rightarrow \langle \text{LISTA_ID} \rangle : \langle \text{TYPE} \rangle ; \langle \text{MAS_VAR} \rangle$   
 $\langle \text{MAS_VAR} \rangle \rightarrow \epsilon$   
 $\langle \text{MAS_VAR} \rangle \rightarrow \langle \text{LISTA_VAR} \rangle$   
 $\langle \text{LISTA_ID} \rangle \rightarrow \text{id } \langle \text{MAS_ID} \rangle$   
 $\langle \text{MAS_ID} \rangle \rightarrow \epsilon$   
 $\langle \text{MAS_ID} \rangle \rightarrow , \langle \text{LISTA_ID} \rangle$

## *TYPE*

Unset

$\langle \text{TYPE} \rangle \rightarrow \text{int}$   
 $\langle \text{TYPE} \rangle \rightarrow \text{float}$

## *BODY*

Unset

$\langle \text{BODY} \rangle \rightarrow \{ \langle \text{DEC\_STATEMENTS} \rangle \}$   
 $\langle \text{DEC\_STATEMENTS} \rangle \rightarrow \epsilon$   
 $\langle \text{DEC\_STATEMENTS} \rangle \rightarrow \langle \text{LISTA\_STATEMENTS} \rangle$   
 $\langle \text{LISTA\_STATEMENTS} \rangle \rightarrow \langle \text{STATEMENT} \rangle \langle \text{MAS\_STATEMENTS} \rangle$   
 $\langle \text{MAS\_STATEMENTS} \rangle \rightarrow \epsilon$   
 $\langle \text{MAS\_STATEMENTS} \rangle \rightarrow \langle \text{LISTA\_STATEMENTS} \rangle$

## *STATEMENT*

Unset

$\langle \text{STATEMENT} \rangle \rightarrow \langle \text{ASSIGN} \rangle$   
 $\langle \text{STATEMENT} \rangle \rightarrow \langle \text{CONDITION} \rangle$   
 $\langle \text{STATEMENT} \rangle \rightarrow \langle \text{CYCLE} \rangle$   
 $\langle \text{STATEMENT} \rangle \rightarrow \langle \text{F\_CALL} \rangle$   
 $\langle \text{STATEMENT} \rangle \rightarrow \langle \text{PRINT} \rangle$

## *PRINT*

Unset

$\langle \text{PRINT\_STMT} \rangle \rightarrow \text{print}(\langle \text{PARAMETROS\_PRINT} \rangle);$   
 $\langle \text{PARAMETROS\_PRINT} \rangle \rightarrow \text{cte.string} \langle \text{MAS\_PRINT} \rangle$   
 $\langle \text{PARAMETROS\_PRINT} \rangle \rightarrow \langle \text{EXPRESIÓN} \rangle \langle \text{MAS\_PRINT} \rangle$   
 $\langle \text{MAS\_PRINT} \rangle \rightarrow \epsilon$   
 $\langle \text{MAS\_PRINT} \rangle \rightarrow , \langle \text{PARAMETROS\_PRINT} \rangle$

## *ASSIGN*

Unset

$\langle \text{ASSIGN} \rangle \rightarrow \text{id} = \langle \text{EXPRESIÓN} \rangle ;$

## *CTE*

```
Unset  
<CTE> → cte_int  
<CTE> → cte_float
```

## *EXPRESIÓN*

```
Unset  
<EXPRESIÓN> → <EXP><MAS_EXPRESIONES>  
<OPERADORES> → >  
<OPERADORES> → <  
<OPERADORES> → !=  
<MAS_EXPRESIONES> → ε  
<MAS_EXPRESIONES> → <OPERADORES><EXP>
```

## *EXP*

```
Unset  
<EXP> → <TÉRMINO><MÁS_EXP>  
<MÁS_EXP> → ε  
<MÁS_EXP> → <w><EXP>  
<OPERADORES_EXP> → +  
<OPERADORES_EXP> → -
```

## *TÉRMINO*

```
Unset  
<TÉRMINO> → <FACTOR><MAS_TÉRMINO>  
<MAS_TERMINO> → ε  
<MAS_TERMINO> → <OPERADORES_TER> <TÉRMINO>  
<OPERADORES_TER> → *  
<OPERADORES_TER> → /
```

## *FACTOR*

```

Unset
<FACTOR> → <OPERADORES_FACTOR><ID_CTE>
<FACTOR> → ( <EXPRESION> )
<ID_CTE> → id
<ID_CTE> → <CTE>
<OPERADORES_FACTOR> → +
<OPERADORES_FACTOR> → -
<OPERADORES_FACTOR> → ε

```

## *F\_CALL*

```

Unset
<F_CALL> → id ( <EXPRESIONES> );
<EXPRESIONES> → ε
<EXPRESIONES> → <EXPRESIONES_F>
<EXPRESIONES_F> → <EXPRESION><LISTA_EXP>
<LISTA_EXP> → ε
<LISTA_EXP> → , <EXPRESIONES_F>

```

## *CYCLE*

```

Unset
<CYCLE> → do <BODY> while ( <EXPRESIÓN> );

```

## *CONDITION*

```

Unset
<CONDITION> → if ( <EXPRESIÓN> ) <BODY> <ELSE>;
<ELSE_CONDITION> → ε
<ELSE_CONDITION> → else <BODY>

```

## *FUNCS*

```

Unset
<FUNCS> → void id ( <PARAMETROS> ) [ <VARS_FUNC> <BODY> ];

```

```
<PARAMETROS> → ε  
<PARAMETROS> → <DEC_PARAMETROS>  
<DEC_PARAMETROS> → id : <TYPE> <LISTA_PARAMETROS>  
<LISTA_PARAMETROS> → ε  
<LISTA_PARAMETROS> → , <DEC_PARAMETROS>  
<VARS_FUNC> → ε  
<VARS_FUNC> → <VARS>
```

## II. Léxico y Sintaxis

### Investigación herramientas

#### *Python Lex-Yacc (PLY)*

PLY es una implementación únicamente en Python, la cual cuenta con herramientas de construcción de léxicos y sintaxis. Ofrece una implementación de lex y yacc que es flexible y potente, permitiendo definir reglas de tokens y gramáticas de manera programática en Python. PLY utiliza un tipo de parser LR o también llamado Bottom-Up el cual va sustituyendo de izquierda a derecha. Asimismo este cuenta con una amplia documentación.

Como se mencionó anteriormente, PLY consta de dos módulos separados: lex.py y yacc.py, ambos se encuentran en el paquete PLY. El módulo lex.py se utiliza para dividir el texto de entrada en una colección de tokens especificados por una colección de reglas de expresiones regulares. Y el módulo yacc.py se utiliza para reconocer la sintaxis del lenguaje que ha sido especificada en forma de una gramática libre de contexto.

Documentación: <https://ply.readthedocs.io/en/latest/>

#### *ANother Tool for Language Recognition (AntLR)*

ANTLR es una herramienta de generación de analizadores sintácticos, originariamente desarrollada en Java. Sin embargo, ofrece la flexibilidad de ser utilizada en una amplia variedad de lenguajes de programación, que incluyen C++, C#, Dart, Java, JavaScript, PHP, Python3, Swift, TypeScript y Go.

Se basa en la técnica de parseo LL, también conocida como Top-Down, que implica una sustitución de derecha a izquierda. ANTLR es especialmente reconocido por su habilidad para manejar gramáticas complejas y su soporte para la generación de árboles de análisis sintáctico. Además, si se detectan ambigüedades en la gramática utilizada, ANTLR es capaz de identificar posibles conflictos y alertar al respecto.

No obstante, es importante tener en cuenta que los analizadores sintácticos generados por ANTLR pueden tener un tamaño considerable debido a la inclusión del runtime de la biblioteca, lo cual puede llegar a ocupar mucho espacio.

Documentación: <https://www antlr2.org/doc/>

### *SLY*

SLY, es una herramienta de análisis sintáctico para Python, ofrece una sintaxis directa y sencilla para definir tokens y gramáticas directamente en el código Python. Únicamente se puede usar en Python 3.6 o versiones posteriores para su funcionamiento. Cabe mencionar que, continúan realizando mejoras, por lo que, no ha sido lanzado oficialmente como una versión estable y completa.

Una de las fortalezas de SLY radica en su capacidad para proporcionar una amplia información de informes de error y diagnóstico, lo que facilita la detección y corrección de problemas durante la construcción de analizadores sintácticos. Además, SLY ofrece un soporte completo para producciones vacías, recuperación de errores, especificadores de precedencia y gramáticas moderadamente ambiguas. Al igual que PLY, SLY utiliza un tipo de parser LR debido a que usa las herramientas de yacc.

Documentación: <https://sly.readthedocs.io/en/latest/>

### *PyParsing*

PyParsing es una biblioteca versátil que brinda soluciones tanto para el análisis léxico como para el análisis sintáctico. Aunque su enfoque principal se centra en el análisis sintáctico, PyParsing ofrece una solución integrada y completa para ambos tipos de análisis.

Utilizando un enfoque de análisis sintáctico LL (left-to-right, leftmost derivation), PyParsing permite procesar la entrada de manera eficiente y precisa, garantizando una interpretación precisa de la sintaxis del lenguaje. Esta metodología permite una mayor flexibilidad en la definición de gramáticas y reglas de análisis sintáctico.

Documentación: <https://pypi.org/project/pyparsing/>

## **Implementación**

Para implementar el parser y scanner se optó por utilizar la herramienta Ply, debido a su extensa documentación y a su robustez en el análisis léxico y sintáctico. Además, la elección de Ply garantiza una mayor eficiencia en el proceso de análisis del código fuente y de igual forma, ofrece una implementación fácil de ejecutar.

## Test-Plan

Es importante destacar que se desarrolló un plan de pruebas por secciones, con el objetivo de asegurar el correcto funcionamiento de las gramáticas formales y los tokens previamente definidos.

Núm	Tipo de prueba	Descripción	Validada
1	Sintaxis del Programa	Verificar la estructura general del programa: inicio con "program", declaración de variables y funciones, presencia de la declaración principal "main", y terminación con "end".	
2	Declaración de Variables	Validar la correcta declaración de variables con la palabra clave "var", seguida de identificadores y tipos de datos.	
3	Cuerpo del Programa	Comprobar que el cuerpo del programa esté correctamente encerrado entre llaves y contenga declaraciones de sentencias.	
4	Declaraciones de Sentencias	Verificar la correcta implementación de asignaciones, condicionales, ciclos, llamadas a funciones y sentencias de impresión.	
5	Asignaciones	Validar la estructura de las asignaciones: identificador seguido de un signo igual y una expresión válida.	
6	Impresión	Comprobar que las sentencias de impresión estén correctamente estructuradas con la palabra clave "print" y parámetros.	
7	Expresiones	Verificar que las expresiones sean evaluadas correctamente, incluyendo operadores de comparación y aritméticos.	
8	Llamadas a Funciones	Comprobar la correcta formación de las llamadas a funciones, incluyendo el nombre de la función y parámetros.	
9	Ciclos	Validar la correcta estructura de los ciclos con la palabra clave "do", una expresión entre paréntesis y un cuerpo de while.	
10	Condiciones	Verificar la correcta formación de las estructuras condicionales con la palabra clave "if", una expresión entre paréntesis y un cuerpo de condición, y la cláusula "else" opcional.	

11	Funciones	Comprobar la declaración adecuada de funciones con la palabra clave "void", el nombre de la función, paréntesis y parámetros opcionales.	
----	-----------	------------------------------------------------------------------------------------------------------------------------------------------	--

De igual forma, se desarrollaron distintos test para evaluar un conjunto de opciones de programas.

### III. Semántica de variables

#### Cubo semántico

Operación	int,int	int,float	float,int	float,float	int, bool	float, bool
=	int	float	error	float	error	error
+	int	float	float	float	error	error
-	int	float	float	float	error	error
*	int	float	float	float	error	error
/	float	float	float	float	error	error
!=	bool	bool	bool	bool	error	error
<	bool	bool	bool	bool	error	error
>	bool	bool	bool	bool	error	error

Estructura seleccionada: Hashmap

Primero se usa un hashmap en donde el <key> va a almacenar las combinaciones en una tupla que representa la combinación (tipo de op1, tipo de op2, operador). Y el valor el tipo de resultante de la operación.

```
Python
cubo_semántico = {
    <key>: <value>,
    <(op1,op2,operador)>: <type>,
    .
    .
    .
    <key>: <value>
}
```

## **Justificación:**

Se seleccionó esta estructura, ya que, en python cuando realizas una consulta de un elemento, se utiliza una función hash interna para convertir la clave (en este caso, una tupla) en un índice en la tabla hash. Por lo que, tiene una complejidad de O(1) lo que la vuelve muy eficiente.

## **Directorio de Funciones**

Una estructura de datos adecuada para esto es un diccionario, donde el nombre de la función es la clave y el valor es la tabla de variables que contiene las variables de la función.

Para la implementación se definió una clase (FuncDirectory) en el archivo FuncDirectory.py, la cual contiene las siguientes funciones:

- *\_\_init\_\_(self)*: Se inicializa el diccionario, los atributos currentFunc (para saber cual es la función actual) y globalName, asimismo, se inicializa los contadores usando iteradores (que contienen los rangos de las direcciones de memoria) para las variables globales y locales.
- *add\_function(self, name, tipo)*: Agrega una nueva función al diccionario, si esta ya fue declarada imprime “Multiple declaration of the function ‘name’”.
- *create\_variable\_table(self)*: Crea la tabla de variables ligada a la función actual.
- *add\_variable\_to\_current\_func(self, var\_name, var\_type)*: Se agrega la variable a la tabla de variables. Se valida que la variable agregada, no sea la misma a una variable declarada en la función global. Y se asigna una dirección de memoria dependiendo de su tipo.
- *get\_next\_memory\_address(self, var\_type, is\_global)*: Retorna el siguiente espacio de memoria con base a su tipo de variable (int, float).
- *add\_params\_to\_current\_func(self, param\_type)*: Agrega el tipo de parámetros al directorio de función.
- *delete\_variable\_table(self, func\_name)*: Borra la tabla de variables que ya no se requiere.
- *get\_current\_type(self, name)*: Obtiene el tipo de las variables, el cual se va agregar a la pila PType que después se utilizará en los cuádruplos.
- *get\_current\_address(self, name)*: Obtiene la dirección de memoria asignada a la variable.

## Tablas de Variables

Para este caso, de igual manera se seleccionó un diccionario, donde el nombre de la variable es la clave y el valor es la tabla de variables que contiene las variables de la función.

Para la implementación se definió una clase (VariableTable) en el archivo VariableTable.py, la cual contiene las siguientes funciones:

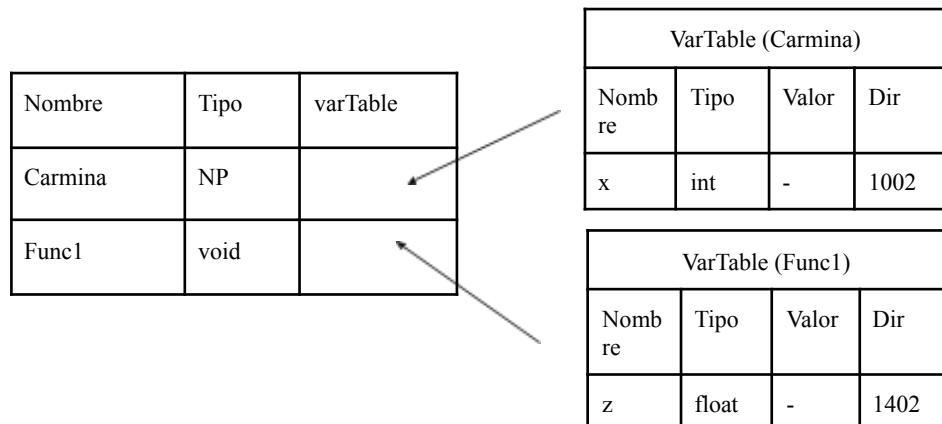
- `__init__(self)`: Inicializa el diccionario de la tabla de variables.
- `add_variable(self, name, var_type, memDirection)`: Agrega la nueva variable al diccionario. Esta se utiliza dentro del directorio de funciones, cuando se ejecuta el método `add_variable_to_current_func()`.

## Justificación

Para el directorio de funciones y la tabla de variables, al igual que el cubo semántico se seleccionó esta estructura, ya que, en python cuando realizas una consulta de un elemento del diccionario, se utiliza la función hash interna para convertir la clave en un índice en la tabla hash. Por lo que, tiene una complejidad de O(1) lo que la vuelve muy eficiente.

\*\*NOTITA\*\*: La tabla de variables de funciones (locales) no puede usar el mismo nombre de las variables globales.

Simbólicamente se visualizan de la siguiente forma:



Python

```
Directorio Funciones = {  
    <key>: {Values}  
    <nombre_función>: {
```

```

        'tipo': tipo,
        'varTable': tabla_variables
    }
}

'func1': {
    'tipo': tipo,
    'varTable': {
        'suma': {'type': 'int', 'value': None, 'dirección': dir_memoria},
        'resta': {'type': 'float', 'value': None, 'dirección':
dir_memoria}
    },
}

.

.

<key>: {Values}
}

```

### Diccionario cte int, float y string

Se seleccionó un diccionario, donde el valor de la constante es la clave y su tipo y el valor es la dirección de memoria. Se seleccionó la clave y su tipo debido a que, se puede declarar dos variables con el mismo número, pero con diferente tipo (ej.  $x = 5.0$  y  $z = 5$ )

Para la implementación se definió una clase (`ConstantDirectory`) en el archivo `cteDirectory.py`, la cual contiene las siguientes funciones:

- `__init__(self)`: Inicializa el diccionario de constantes. Asimismo, se inicializan los contadores usando iteradores (que contienen los rangos de las direcciones de memoria) para las constantes.
- `add_constant(self, constant, constan_type)`: Agrega la constante con su tipo y dirección de memoria.
- `get_constant(self, constant)`: Obtiene la constante, junto con su tipo y dirección. Este método se utiliza en la generación de cuádruplos.
- `get_all_constants(self)`: Regresa todas las constantes y su dirección de memoria, en un objeto de tipos `constantDirectory`.

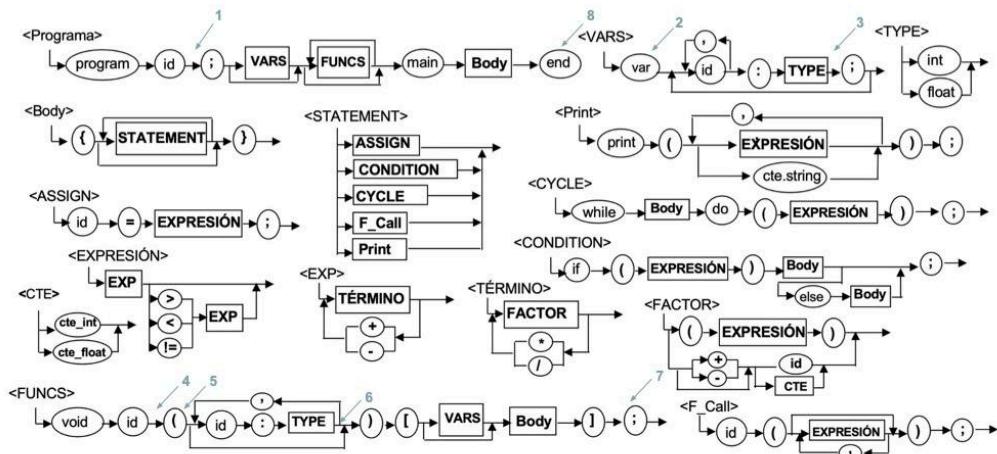
Python

```
constantDirectory = {
    <key>: <value>,
    <valor, tipo>: {
        'dirección': dir_memoria,
    }
}

.
.
.

<key>: <value>
}
```

## Puntos neurálgicos Directorio y Tabla



1. Add(id,type) to dirFunc and set current function(id).
2. Create variable table and link it to current func
  - a. Validate if it doesn't have one
3. Add the variables to the variable table
  - a. Validate if it already exists.
    - i. Error: Multiple declaration
  - b. Add(id, type)
4. Add(id,type) to dirFunc and set current function(id).
5. Create variable table.
  - a. Validate if it doesn't have one
6. Add variable to variable table.
  - a. Validate if it already exists.
    - i. Error: Multiple declaration
  - b. Add(id, type)
7. Delete variable of function (id).
8. Delete FuncDir

## IV. Código Intermedio

### Pilas y Fila

Se crearon 3 pilas, las cuales son:

PilaO = Pila Operandos

En donde como su nombre lo indica, se van agregando los operandos y su respectivo tipo en una pila con tuplas.

Pila Operandos

Tipo (int|float)

POper = Pila Operadores

En este se agregan únicamente operadores y se van sacando dependiendo de la jerarquía.

POper

PJumps = Pila Saltos

En esta se van agregando los índices de los cuádruplos pendientes a completar (goto, gotoF)

PilaS

Y la fila donde se almacenan los cuádruplos

Quad = cuádruplos

En esta fila se almacena un objeto de la clase quadruple, donde tiene como parámetro primero el índice del operador, después la dirección del operando izquierdo, la del derecho y la dirección del resultado el cual se define el tipo por medio del cubo semántico.

(operator, left\_operand, right\_operand, result)

\*Notita: los resultados se encuentran en variables temporales (AVAIL)

Los índices de los operadores y palabras de los cuádruplos se definieron en un diccionario de la siguiente forma:

Python

```
operator_index = {
    '=': 0,
    '+': 1,
    '-': 2,
```

```

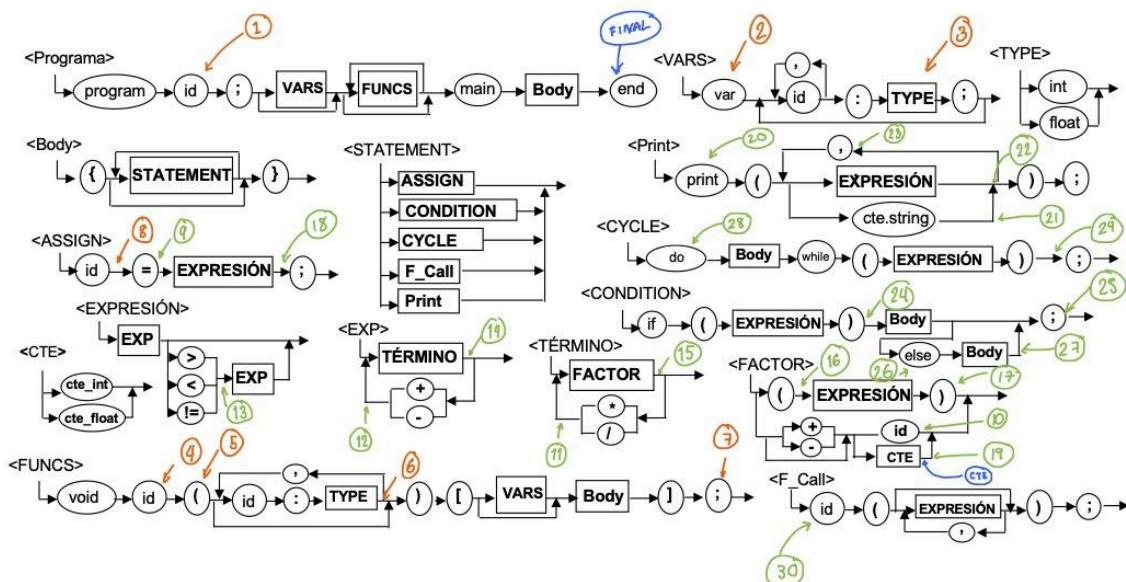
'*': 3,
'/': 4,
'!=': 5,
'<': 6,
'>': 7,
'Goto': 8,
'GotoF': 9,
'GotoV': 10,
'print': 11
}

```

### Justificación:

Se eligió un diccionario, para poder llevar un orden y poder documentar sobre qué representa cada número. De igual manera, para que cuando se requiera agregar algo más, sea de manera sencilla.

### Puntos neurálgicos



1. Add ( $\text{id}$ ,  $\text{type}$ ) to  $\text{dirFunc}$  and set current function ( $\text{id}$ )
  2. Create variable table and link it to current func.
    - a) Validate if it doesn't have one.
  3. Add the variables to the variable table.
    - a) Validate if it already exists.
      - i) Error: Multiple declaration.
    - b) Add ( $\text{id}$ ,  $\text{type}$ )
  4. Add ( $\text{id}$ ,  $\text{type}$ ) to  $\text{dirFunc}$  and set current Function ( $\text{id}$ ).
  5. Create variable table
    - a) Validate if it doesn't have one.
  6. Add variable to variable table.
    - a) Validate if it already exists.
      - i) Error: Multiple declaration
    - b) Add ( $\text{id}$ ,  $\text{type}$ )
  7. Delete variable of function ( $\text{id}$ )
  8. Delete FuncDir.
- CTE. Determine constant-type (float or int)  
Add\_constant ( $\text{id}$ , constan-type) to ConstDir

9. pOper.append( = )  
 10. pilaO.append( id.name, id.type )  
 11. pOper.append( \* or / )  
 12. pOper.append( + or - )  
 13. Paper.append( > or < or != )  
 13. 1. see if there's a > or < or != at Paper  
 14. If paper.top() == '+' or '-' then  
     a) right\_operand, right\_type = PilaO.pop()  
         left\_operand, left\_type = PilaO.pop()  
         operator = POper.pop()  
         Evaluate the result type = SemanticCube(left\_type, right\_type, operator)  
  
     b) If result\_type != VALUE ERROR  
         result = take next temporary  
         Quadruple(operator, left\_op, right\_op, result)  
         cont++  
         pilaO.append(result, result\_type)  
         return to AVAIL-temporal  
  
     else  
         ERROR ('Type mismatch')
15. If pOper.top() == '\*' or '/' then  
 semanticOperations()

16. Save 'migajita de Pan' POper.append()

17. Adiós migajita POper.pop()

18. if pOper.top() == '='

Semantic Assign()

a) right\_operand, right\_type = PilaO.pop()

left\_operand, left\_type = PilaO.pop()

operator = POper.pop()

result type = SemanticCube(left\_type, right\_type, operator)

b) if result\_type != VALUE ERROR

Quadruple(operator, right\_op, None, left\_op)

Cont ++

19. Save cte

cte\_address = constDirac['address']

cte\_type = constDirac['type']

20. POper.append('Print')

21. PilaO.append(cte\_string, 'string')

22. if pOper.top() == 'print'

SemanticPrint()

cte\_string, cte\_type = PilaO.pop()

operator = POper.pop()

Quadruple(operator, None, None, cte\_string)

Cont ++

23. POper.append('Print')

24. exp, exp\_type = PilaO.pop()

icConditionIf()

if (exp\_type != bool)

VALUE ERROR

else

Quadruple('GotoF', result, None, None)

semantic

```

    } · PJumps.append (cont)
        cont ++
    
```

25. Fill quadruples Goto F

```

fillGotoF() {
    if 'only if'
        if PJumps not empty
            end = PJumps.pop()
            Quad[end].result = cont
}
    
```

26. Go to Else

```

semantic(CondElse) {
    Quadruple ('Go to', None, None, None)
    cont ++
    fillGotoF()
    PJumps.append(cont - 1)
}
    
```

27. FillGoto()

```

if PJumps not empty
    false = PJumps.pop()
    Quad [false].result = cont
}
    
```

28. Add 'migajita pan' of cycle

```
PJumps.append (cont);
```

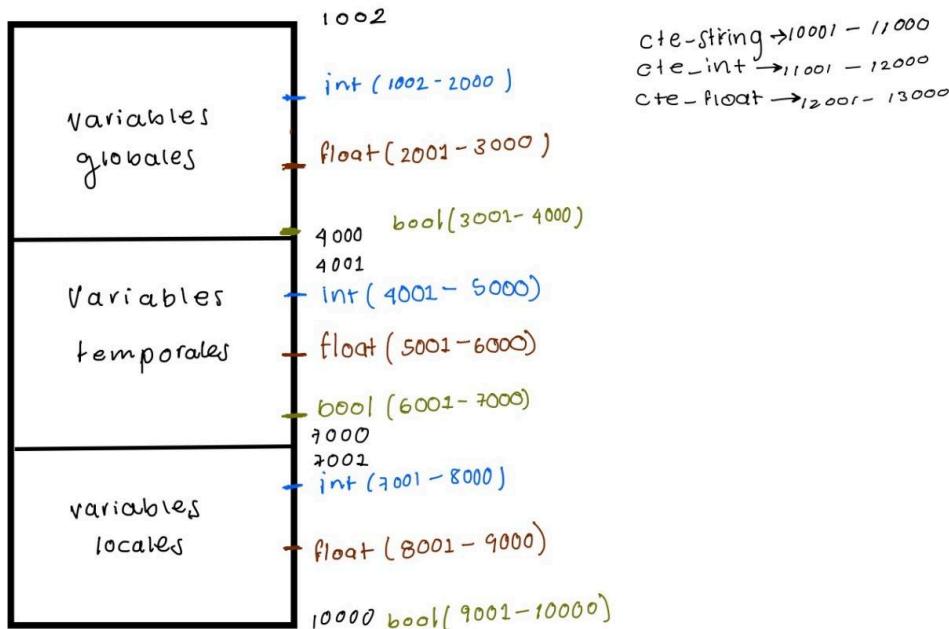
29. Go to V of cycle

```

condition, condition-type = PilaO.pop()
semantic(Cycle) {
    if condition-type != bool
        ERROR
    else
        returnTo = PJumps.pop()
        Quadruple ('GotoV', result, None, returnTo)
        cont ++
}
    
```

## V. Direcciones de memoria

Para las direcciones de memoria se determinaron los siguientes rubros.



## VI. Ejecución - Máquina virtual

### Mapa de memoria

Se creó una clase llamada Memory en el archivo Memory.py, con la cual se maneja una memoria para cada tipo de variable. En donde, se utiliza la estructura de datos de un arreglo, el cual tiene un tamaño que depende de las direcciones de memoria utilizadas.

Esta contiene los siguientes métodos:

- `__init__(self)`: Se inicializan los arreglos para cada tipo y sección.
- `get_memory_segment(self, address)`: Determina el segmento de memoria a la que se va a agregar el valor. Y retorna el segmento, así como la dirección base.
- `set_value(self, address, value)`: Se almacena el valor de la memoria en la dirección especificada. Para calcular en qué index del arreglo se va a agregar, se calcula la `dirección_de_memoria - dirección_base` (ej. para una variable global de tipo int con la dirección 1005, el index es igual a  $1005 - 1002 = 3$ ).

- *get\_value(self, address)*: Se obtiene el valor de la dirección de memoria especificada. Por lo que se accede al arreglo de la siguiente manera, segmento[dirección\_de\_memoria - dirección\_base].
- *initialize\_constants(self, constant\_directory)*: Se inicializa en memoria las constantes leídas del directorio de constantes en su arreglo correspondiente (constant\_string, constant\_int, constant\_float).

## Procesador (CPU)

Para la lecturas de los cuádruplos se creó un archivo *virtualMachine.py*, el cual contiene una función llamada *execute\_quadruples()*. En donde, se implementa la lógica de ejecutar la lista de objetos de tipo quadruples y actualiza la memoria conforme se va leyendo estos.

En donde se recorre todos los cuádruplos y se accede a cada elemento de la siguiente manera:

Python

```
quad = quadruples[current_ip]
operator, left_operand, right_operand, result = quad.operator, quad.left_operand,
quad.right_operand, quad.result
```

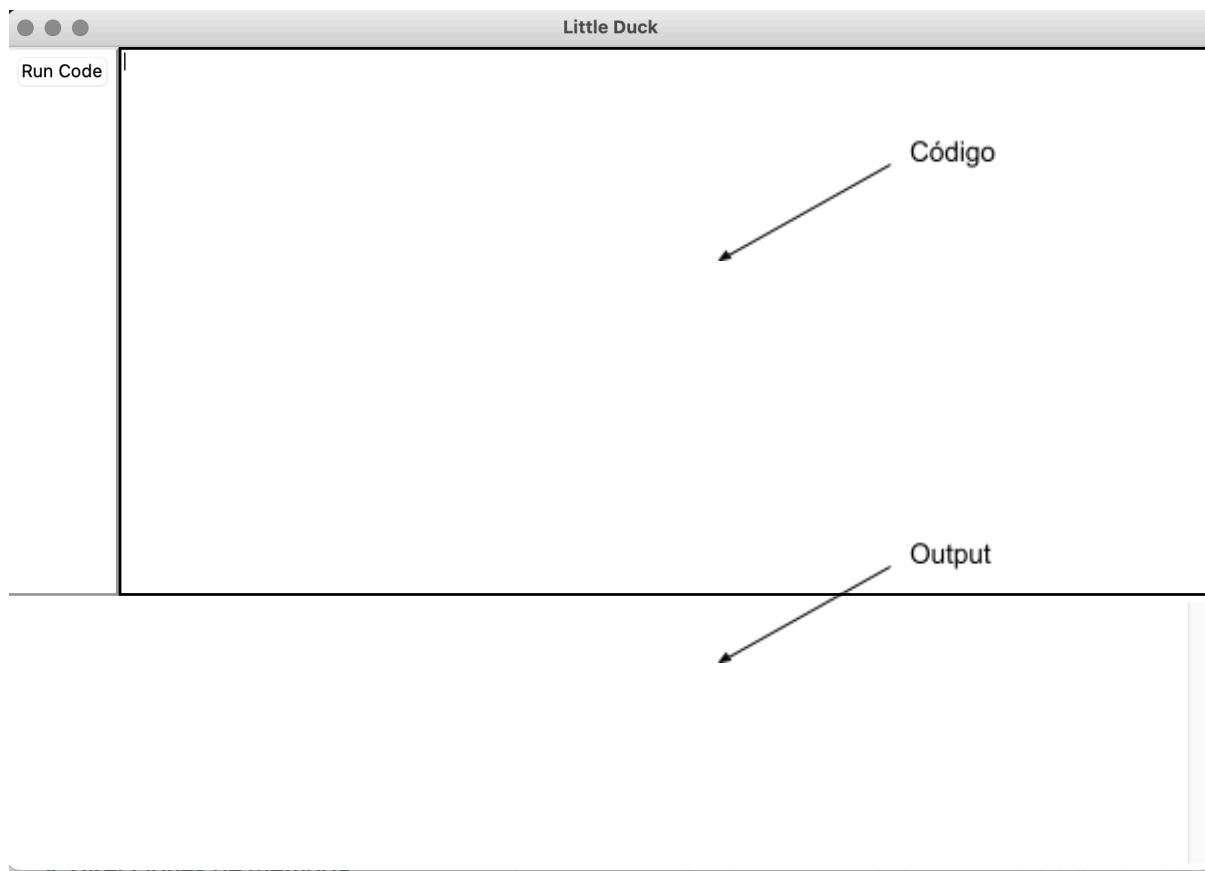
Luego por medio de un ciclo se va recorriendo los cuádruplos usando un índice llamado *current\_ip*, y después con condicionales se determina qué tipo de operación se va a realizar, esto dependiendo del valor que tiene el cuádruplo en *operator*. Y retorna un arreglo que contiene los valores de print (*output*).

## IDE

Con el objetivo de hacer la conexión entre el compilador y la máquina virtual, se realizó un IDE. En donde se tomó como referencia para UI la documentación y el ejemplo del siguiente link: <https://realpython.com/python-gui-tkinter/>

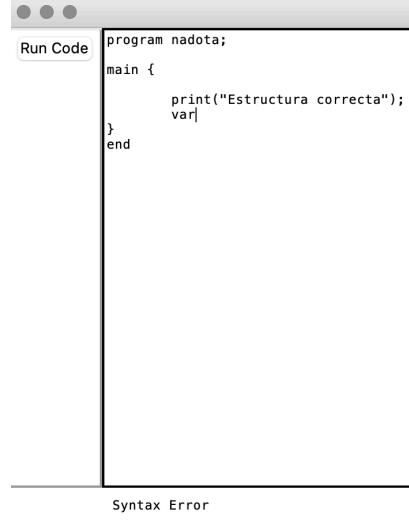
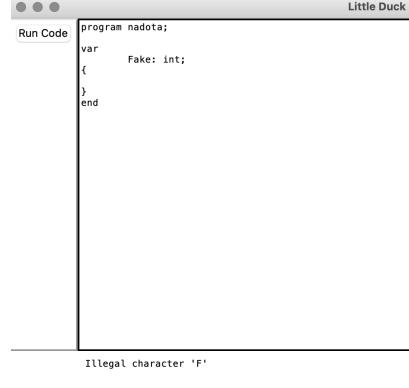
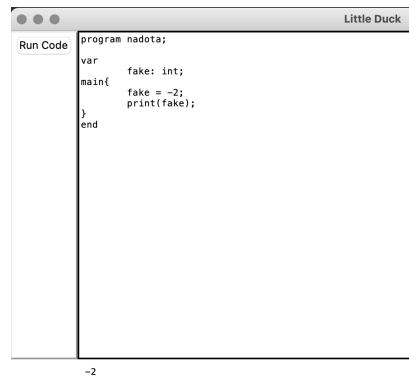
En el archivo IDE.py se encuentra el método *compile\_and\_run()*, en donde primero se almacena el código que se ingresó en el IDE, para posteriormente mandar el código a compilar (método *compile\_code()* en parser.py). Después de que se compila el código, este retorna un objeto de tipo *cte\_directory* y una lista con objetos cuádruples.

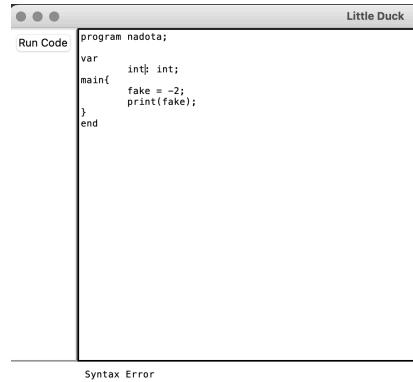
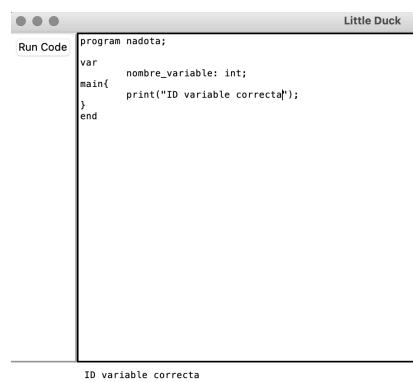
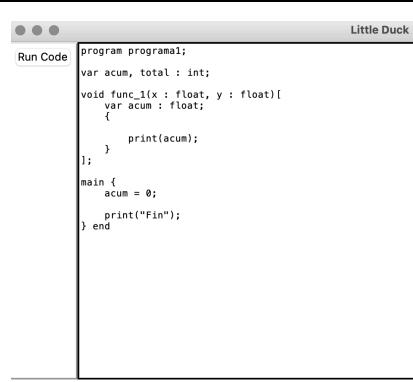
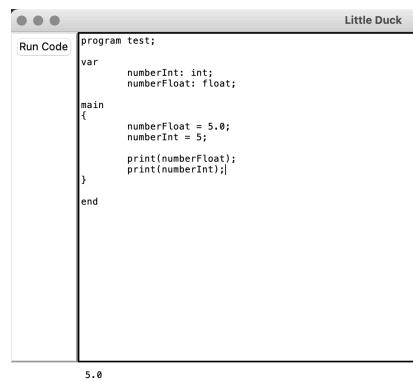
Por lo que, a partir de esto se inicializa la memoria con el directorio de constantes y se ejecutan los cuádruplos. Para posteriormente, visualizar el output del programa.

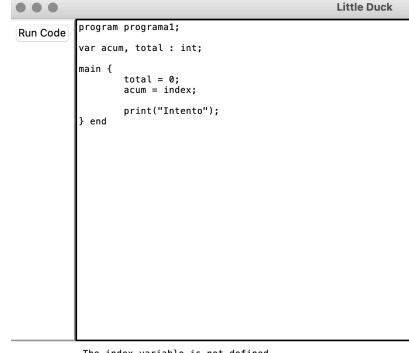
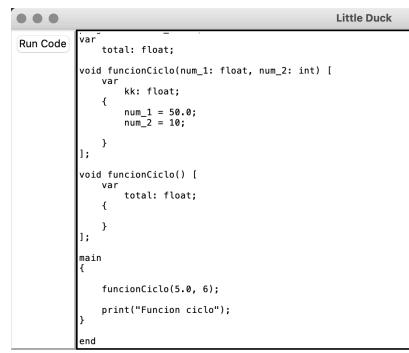
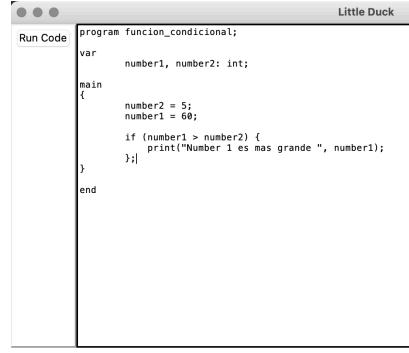
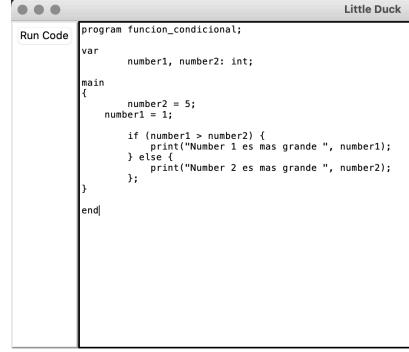


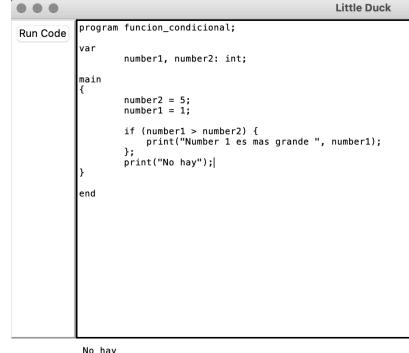
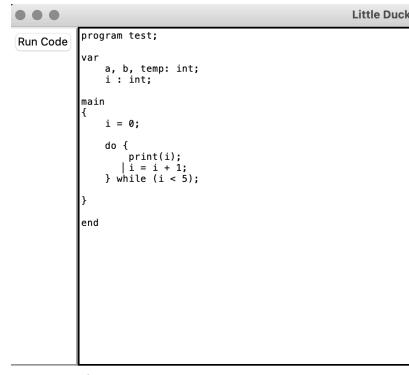
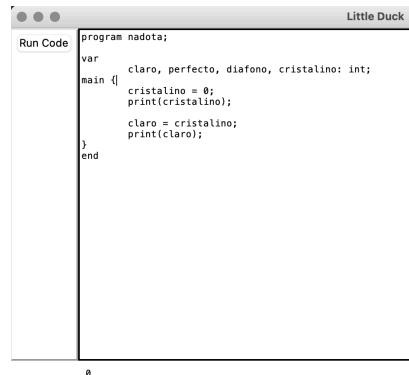
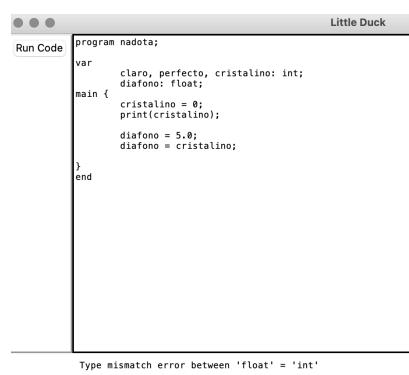
## VII. Tests

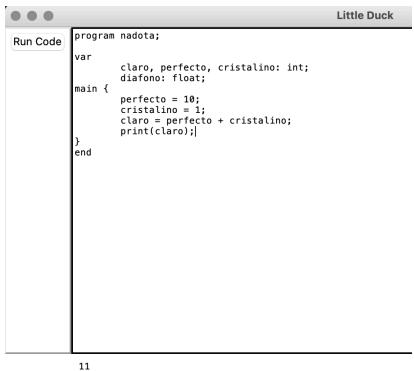
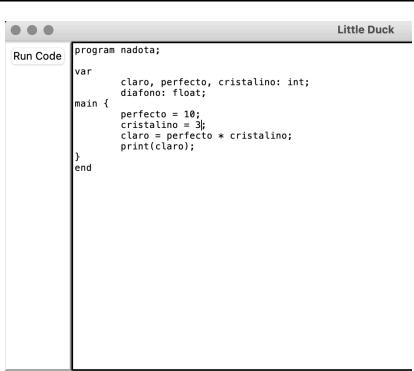
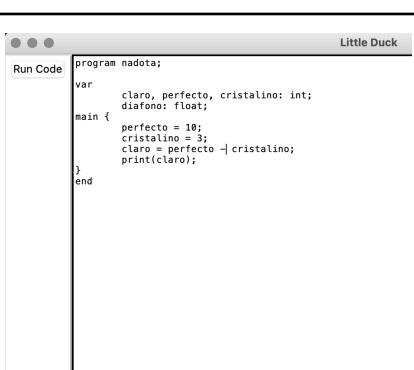
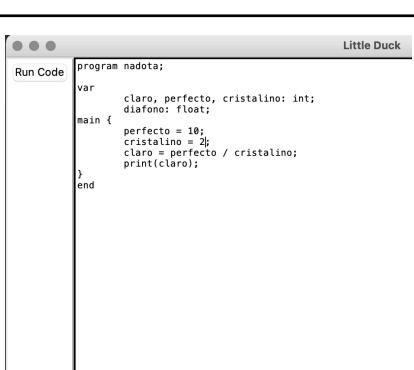
Núm	Tipo de prueba	Descripción	Evidencia
1	Sintaxis del Programa	Verificar la estructura general del programa: inicio con "program", "main", y terminación con "end".	A screenshot of the Little Duck IDE showing a successful program run. The code editor contains: <pre>program nadota; main {     print("Estructura correcta"); } end</pre> The output window below it displays the text "Estructura correcta", indicating the program ran successfully. <p>Estructura correcta</p>

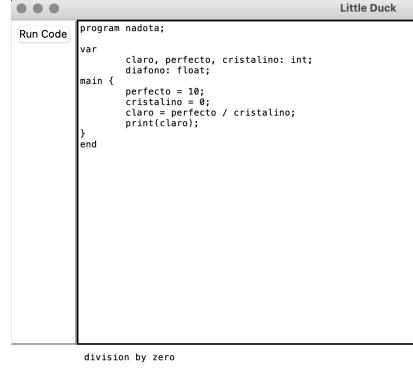
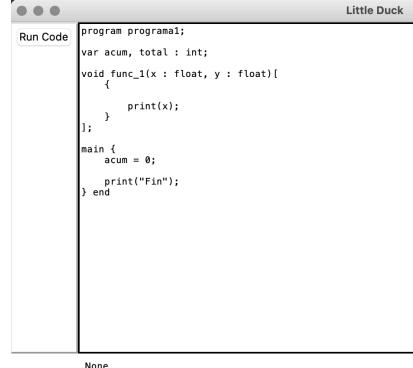
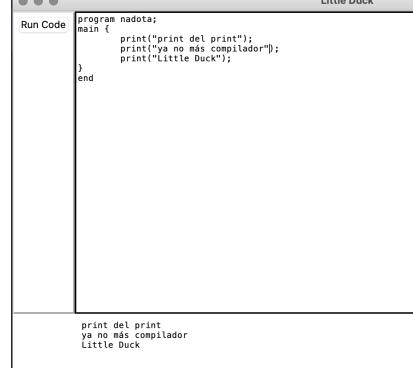
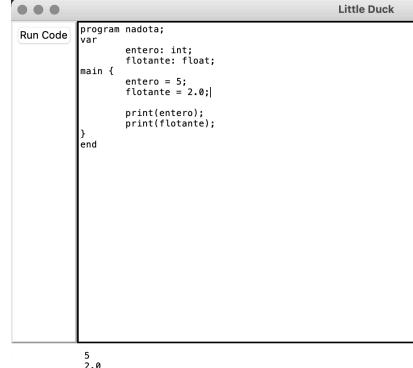
2	Sintaxis del programa	Desplegar error ‘Syntax error’. Al realizar un programa con estructura incorrecta.	 <p>Syntax Error</p>
3	Léxico de las variables	Desplegar un error ‘illegal Char’ porque se inicializa los nombres de las variables con letras mayúsculas.	 <p>Illegal character 'F'</p>
4	Léxico de variables	Aceptar variables negativas.	 <p>-2</p>

5	Léxico de palabras reservadas	Desplegar un error al poner el nombre de una variable con una palabra reservada.	 <pre>Run Code program nadota; var   int; // Error: 'int' is a reserved word main{   fake = -2;   print(fake); }end</pre> <p>Syntax Error</p>
6	Léxico de id variables	Validar que la estructura del id de La variable inicia con una letra minúscula y puede seguir con letras minúsculas y/o números o ‘_’.	 <pre>Run Code program nadota; var   nombre_variable: int; main{   print("ID variable correcta"); }end</pre> <p>ID variable correcta</p>
7	Declaración de variables	Visualizar el error, cuando hay variables repetidas con las globales.	 <pre>Run Code program programal; var acum, total : int; void func_1(x : float, y : float){   var acum : float;   {     print(acum);   }; } main {   acum = 0;   print("Fin"); }end</pre> <p>Variable 'acum' ya definida en el ámbito global</p>
8	Declaración de constantes	Visualizar la declaración de constante a una variable float y una variable constante con el mismo número (solo que diferente tipo) y ver cómo se comporta.	 <pre>Run Code program test; var   numberInt: int;   numberFloat: float; main {   numberFloat = 5.0;   numberInt = 5;   print(numberFloat);   print(numberInt); }end</pre> <p>5.0 5</p>

9	Sin declaración de variables	Desplegar error cuando se usa una variable que no está previamente definida.	 <pre>Run Code program program1; var acum, total : int; main {     total = 0;     acum = index;     print("Intento"); } end</pre> <p>The index variable is not defined</p>
10	Declaración de funciones	Error al visualizar múltiple declaración de funciones.	 <pre>Run Code var total: float; void funcionCiclo(num_1: float, num_2: int) {     var kk: float;     {         num_1 = 50.0;         num_2 = 10;     } } void funcionCiclo() {     var total: float;     {     } } main {     funcionCiclo(5.0, 6);     print("Funcion ciclo"); } end</pre> <p>Multiple declaration of the function funcionCiclo</p>
11	Condicional	Evaluar que cuando entre al 'if', funcione correctamente.	 <pre>Run Code program funcion_condicional; var number1, number2: int; main {     number2 = 5;     number1 = 60;     if (number1 &gt; number2) {         print("Number 1 es mas grande ", number1);     } } end</pre> <p>Number 1 es mas grande 60</p>
12	Condicional	Evaluar que el 'else', funcione correctamente.	 <pre>Run Code program funcion_condicional; var number1, number2: int; main {     number2 = 5;     number1 = 1;     if (number1 &gt; number2) {         print("Number 1 es mas grande ", number1);     } else {         print("Number 2 es mas grande ", number2);     } } end</pre> <p>Number 2 es mas grande 5</p>

13	Condicional	Validar que si el condicional no se cumple y no tenga un 'else', se salte a la siguiente línea.	 <pre>program funcion_condicional; var number1, number2: int; main() {     number2 = 5;     number1 = 1;      if (number1 &gt; number2) {         print("Number 1 es mas grande ", number1);     };     print("No hay"); } end</pre> <p>No hay</p>
14	Ciclo	Evaluar que el condicional del while funcione correctamente.	 <pre>program test; var a, b, temp: int; i : int;  main() {     i = 0;     do {         print(i);           i = i + 1;     } while (i &lt; 5); } end</pre> <p>0 1 2 3 4</p>
15	Asignaciones de múltiples variables.	Evaluar el correcto funcionamiento de las múltiples variables.	 <pre>program nadota; var claro, perfecto, diafono, cristalino: int; main () {     cristalino = 0;     print(cristalino);     claro = cristalino; } end</pre> <p>0 0</p>
16	Asignaciones de tipo float = int.	Desplegar error 'Type mismatch error'	 <pre>program nadota; var claro, perfecto, cristalino: int; diafono: float; main {     cristalino = 0;     print(cristalino);      diafono = 5.0;     diafono = cristalino; } end</pre> <p>Type mismatch error between 'float' = 'int'</p>

17	Suma de variables	Evaluar el correcto funcionamiento de las sumas.	 Little Duck Run Code program nadota; var     claro, perfecto, cristalino: int; diaphone: float; main { perfecto = 10; cristalino = 1; claro = perfecto + cristalino; } end  11
18	Multiplicación de variables	Evaluar el correcto funcionamiento de las multiplicaciones.	 Little Duck Run Code program nadota; var     claro, perfecto, cristalino: int; diaphone: float; main { perfecto = 10; cristalino = 3; claro = perfecto * cristalino; } end  30
19	Resta de variables	Evaluar el correcto funcionamiento de las restas.	 Little Duck Run Code program nadota; var     claro, perfecto, cristalino: int; diaphone: float; main { perfecto = 10; cristalino = 3; claro = perfecto - cristalino; } end  7
20	División de variables	Evaluar el correcto funcionamiento de las divisiones.	 Little Duck Run Code program nadota; var     claro, perfecto, cristalino: int; diaphone: float; main { perfecto = 10; cristalino = 2; claro = perfecto / cristalino; } end  5.0

21	División entre 0	Marcar error cuando se realiza una división entre cero.	 <pre>Run Code program nadota; var     claro, perfecto, cristalino: int;     diafono: float; main {     perfecto = 10;     cristalino = 0;     claro = perfecto / cristalino; } end</pre> <p>division by zero</p>
22	Print de variable sin valor	Desplegar que la variable no tiene un valor.	 <pre>Run Code program programal; var acum, total : int; void_func_1(x : float, y : float){     print(x); } main {     acum = 0;     print("Fin"); } end</pre> <p>None Fin</p>
23	Print	Evaluar el correcto funcionamiento para imprimir los resultados que se desplieguen.	 <pre>Run Code program nadota; main {     print("print del print");     print("ya no más compilador");     print("Little Duck"); } end</pre> <p>print del print ya no más compilador Little Duck</p>
24	Print de variables con su tipo	Evaluar que se imprima las variables de forma correcta (si es un int imprimir un int)	 <pre>Run Code program nadota; var     entero: int;     flotante: float; main {     entero = 5;     flotante = 2.0;     print(intento);     print(flotante); } end</pre> <p>5 2.0</p>

## Test 1

```
Unset
program test;

var i, n, square : int;

main {
    n = 10;

    i = 0;

    do {
        square = i * i;
        print(square, " ");
        i = i + 1;
    } while (i < n);
}
end
```

The screenshot shows a window titled "Little Duck". On the left, there is a code editor with the following pseudocode:

```
Run Code
program test;
var i, n, square : int;
main {
    n = 10;
    i = 0;
    do {
        square = i * i;
        print(square, " ");
        i = i + 1;
    } while (i < n);
}
end
```

On the right, the output window displays the following sequence of numbers:

```
0
1
4
9
16
25
36
49
64
81
```

## Test 2

```
Unset
program test2;

var
```

```

a,b,c,d: int;
e: float;

main
{
    a = 1;
    b = 2;
    c = 5;
    d = 0;

    if (a + b < c){
        a = b + c;
        print("if");
        print(a);
        do {
            a = a - 1;
            print("Do While");
            print(a);
        } while (a > b+c);
    }
    else {
        print("else");
        a = b + c * d;
        b = a - d;
    };
}

```

end

The screenshot shows a software interface with a title bar 'Little Duck'. The main area contains a code editor with the following content:

```

Run Code program test2;
var
    a,b,c,d: int;
    e: float;

main
{
    a = 1;
    b = 2;
    c = 5;
    d = 0;

    if (a + b < c){
        a = b + c;
        print("if");
        print(a);
        do {
            a = a - 1;
            print("Do While");
            print(a);
        } while (a > b+c);
    }
    else {
        print("else");
        a = b + c * d;
        b = a - d;
    };
}

```

Below the code editor is a terminal window displaying the output of the program:

```

if
7
Do While
6

```

### Test 3

```
Unset
program test;

var i, j, n, square, input : int;

main {
    n = 10;
    print("Number of squares to calculate: ");
    print(n);
    if (n < 0) {
        print("It's not a positive integer greater than zero.");
    } else {
        i = 0;
        do {
            if (i / 2 > 0) {
                square = i * i;
                print("Square of ", i, " is: ", square);
            } else {
                print(i, " is odd, calculating squares for next ", n, "
numbers.");
                j = i;
                do {
                    square = j * j;
                    if (square / 2 > 0) {
                        print("Square of ", j, " is: ", square, " and it
is even.");
                    } else {
                        print("Square of ", j, " is: ", square, " and it
is odd.");
                    }
                    j = j + 1;
                } while (j < i + n);
            };
            i = i + 1;
        } while (i < n);
    };
}
end
```

output:

Number of squares to calculate:

10

0

is odd, calculating squares for next

10

numbers.

Square of  
0  
is:  
0

and it is odd.

Square of  
1  
is:  
1

and it is even.

Square of  
2  
is:  
4

and it is even.

Square of  
3  
is:  
9

and it is even.

Square of  
4  
is:  
16

and it is even.

Square of  
5  
is:  
25

and it is even.

Square of  
6  
is:  
36

and it is even.

Square of  
7  
is:  
49

and it is even.

Square of  
8  
is:

64

and it is even.

Square of

9

is:

81

and it is even.

Square of

1

is:

1

Square of

2

is:

4

Square of

3

is:

9

Square of

4

is:

16

Square of

5

is:

25

Square of

6

is:

36

Square of

7

is:

49

Square of

8

is:

64

Square of

9

is:

81

## **Liga del repositorio**

<https://github.com/carminalp/Compilador>

## Referencias

Python GUI Programming With Tkinter. (s.f.). Real Python. Recuperado el 26 de mayo de 2024, desde: (<https://realpython.com/python-gui-tkinter/>)

PLY (Python Lex-Yacc). (s.f.). Read the docs. Recuperado el 26 de mayo de 2024, desde: <https://ply.readthedocs.io/en/latest/>

ANTLR Reference Manual. (s.f.). Antlr2. Recuperado el 26 de mayo de 2024, desde: <https://www.antlr2.org/doc/>

SLY (SLY Lex Yacc). (s.f.). Read the docs. Recuperado el 26 de mayo de 2024, desde: <https://sly.readthedocs.io/en/latest/>

Pyparsing PyPI. (s.f.). pypi. Recuperado el 26 de mayo de 2024, desde <https://pypi.org/project/pyparsing/>