

Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Organización de Lenguajes y Compiladores 1
Segundo Semestre 2024



USAC
TRICENTENARIA
Universidad de San Carlos de Guatemala

Catedráticos:

Ing. Manuel Castillo

Ing. Kevin Lajpop

Ing. Mario Bautista

Tutores académicos:

Fabian Reyna

Carlos Acabal

Xhunik Miguel

ComplInterpreter

Proyecto 2

Tabla de Contenido

1. Objetivo General	5
2. Objetivos específicos	5
3. Descripción General	5
4. Entorno de Trabajo	6
4.1 Editor	6
4.2 Funcionalidades	6
4.4 Herramientas	6
4.5 Reportes	6
4.6 Área de Consola	7
5. Descripción del Lenguaje	7
5.1 Case Insensitive	7
5.2 Comentarios	7
5.3 Tipos de Dato	8
5.4 Secuencias de escape	9
5.5 Operadores Aritméticos	9
5.5.1 Suma	9
5.5.2 Resta	10
5.5.3 Multiplicación	10
5.5.4 División	11
5.5.5 Potencia	11
5.5.6 Raíz	12
5.5.6 Módulo	12
5.5.7 Negación Unaria	12
5.6 Operadores Relacionales	13
5.7 Operador ternario	14
5.8 Operadores Lógicos	15
5.9 Signos de Agrupación	15
5.10 Precedencia de Operaciones	15
5.11 Caracteres de finalización y encapsulamiento de sentencias	16
5.12 Declaración de Variables	16
5.12.1 Variables mutables	17
5.12.2 Variables inmutables	17
5.13 Casteos	18
5.14 Incremento y Decremento	18
5.15 Vectores	19
5.15.1 Declaración de Vectores	19
5.15.2 Acceso a vectores	19
5.15.3 Modificación de Vectores	20
5.16 Sentencias de control	20
5.16.1 Sentencia IF	21

5.16.2 Switch case	21
5.16.2.1 Switch	21
5.16.2.2 Case	22
5.16.2.2 Default	22
5.17 Sentencias Cíclicas	22
5.17.1 While	23
5.17.2 For	23
5.17.3 Do-Until	23
5.17.4 Loop	24
5.18 Sentencias de Transferencia	24
5.18.1 Break	24
5.18.2 Continue	24
5.18.3 Return	25
5.19 Funciones	25
5.20 Métodos	26
5.21 Llamadas	27
5.21 Función echo	28
5.22 Funciones Nativas	28
5.22.1 Lower	28
5.22.2 Upper	28
5.22.3 Round	29
5.23.4 Length	29
5.23.5 Truncate	30
5.23.6 Función Is	30
5.25.7 ToString	30
5.25.8 toCharArray	30
5.25.9 reverse	30
5.25.10 max	31
5.25.11 min	31
5.25.12 sum	31
5.25.13 average	32
5.26 Función Ejecutar	32
5.27 Estructura de las instrucciones	32
6. Reportes	33
6.1 Tabla de errores	33
6.2 Tabla de Símbolos	33
6.3 AST	33
6.4 Salidas en consola	34
7. Requerimientos Mínimos	34
8. Entregables	35
9. Restricciones	35

10. Fecha de Entrega	36
11. Anexos	36
11.1 Ejemplo de archivo de entrada	36

1. Objetivo General

Aplicar los conocimientos sobre la fase de análisis léxico y sintáctico de un compilador para la realización de un intérprete sencillo, con las funcionalidades principales para que sea funcional.

2. Objetivos específicos

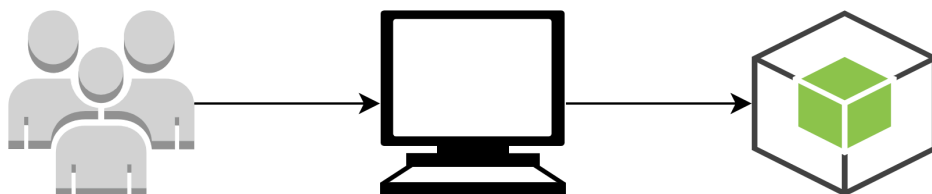
- Reforzar los conocimientos de análisis léxico, sintáctico y semántico para la creación de un lenguaje de programación.
- Aplicar los conceptos de compiladores para implementar un proceso de interpretación de código de alto nivel.
- Aplicar los conceptos de compiladores para analizar un lenguaje de programación y producir las salidas esperadas.
- Aplicar la teoría de compiladores para la creación de soluciones de software.
- Generar aplicaciones utilizando la arquitectura cliente-servidor.

3. Descripción General

El curso de Organización de Lenguajes y Compiladores 1, perteneciente a la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, necesita un intérprete capaz de reconocer un nuevo lenguaje de programación y generar reportes.

Por tanto, a usted, que es estudiante del curso de Compiladores 1, se le encomienda realizar el proyecto llamado ComInterpreter. dado sus altos conocimientos en temas de análisis léxico, sintáctico y semántico.

Arquitectura Cliente - Servidor



4. Entorno de Trabajo

4.1 Editor

El editor será parte del entorno de trabajo, cuya finalidad será proporcionar ciertas funcionalidades, características y herramientas que serán de utilidad para el usuario. La función principal del editor será el ingreso del código fuente que será analizado. En este se podrán abrir diferentes archivos al mismo tiempo y deberá mostrar la línea actual. El editor de texto se tendrá que mostrar en el navegador. Queda a discreción del estudiante el diseño.

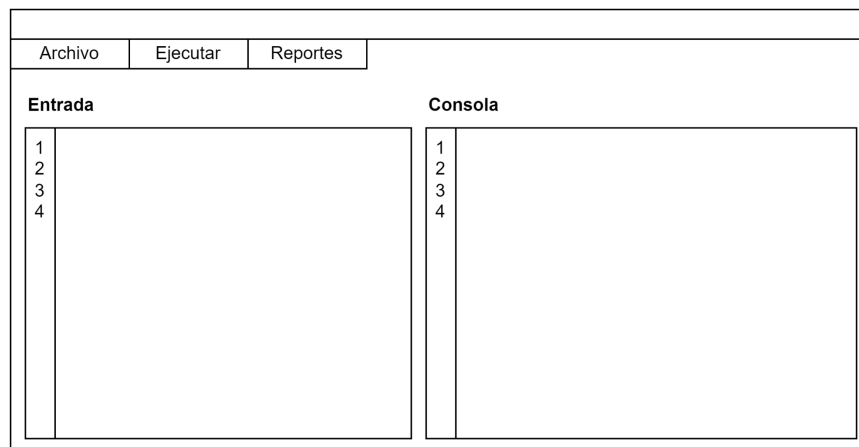


Figura 1: Propuesta Interfaz

4.2 Funcionalidades

- **Nuevo archivo:** El editor deberá de ser capaz de crear archivos en blanco.
- **Abrir archivos:** El editor deberá abrir archivos .ci y colocar su contenido en el área de entrada
- **Guardar:** El editor debe tener la capacidad de guardar el estado del archivo que se está trabajando.

4.4 Herramientas

- **Ejecutar:** hará el llamado al intérprete, el cual se hará cargo de realizar los análisis léxico, sintáctico y semántico, además de ejecutar todas las sentencias.

4.5 Reportes

- **Reporte de Errores:** se mostrarán todos los errores encontrados al realizar el análisis léxico, sintáctico y semántico.

- **Generar Árbol de Análisis Sintáctico:** se deberá generar una imagen del árbol de análisis sintáctico resultante del último archivo ejecutado.
- **Reporte de Tabla de Símbolos:** se mostrarán todas las variables, métodos y funciones que han sido declarados dentro del flujo del programa.

4.6 Área de Consola

En esta área se mostrarán los resultados, mensajes y todo lo que sea indicado dentro del lenguaje. Tiene como restricción el no ser editable por el usuario y únicamente puede mostrar información.

5. Descripción del Lenguaje

5.1 Case Insensitive

El lenguaje es case insensitive por lo que no reconoce entre mayúsculas y minúsculas.

```
let a:int = 10;
```

```
LeT A: iNt = 0;
```

Nota: Ambos casos son lo mismo

5.2 Comentarios

Los comentarios son una forma elegante de indicar que función tiene cierta sección del código que se ha escrito, simplemente para dejar un mensaje en específico. El lenguaje deberá soportar dos tipos de comentarios que son los siguientes:

5.2.1 Comentarios de una línea

Este comentario comenzará con `//` y deberá terminar con un salto de línea.

5.2.2 Comentarios multilinea

Este comentario comenzará con `/*` y terminará con `*/`.

```
// Esto es un comentario de una sola línea
```

```
/* Esto es un comentario
```

```
Multilínea */
```

5.3 Tipos de Dato

Los tipos de dato que soportará el lenguaje en concepto de un tipo de variable se definen a continuación:

Tipo	Definición	Descripción	Ejemplo	Observaciones	Valor por defecto
Entero	int	Admite únicamente valores numéricos enteros	1, -120, etc	Se maneja cualquier cantidad de dígitos.	0
Decimal	double	Admite valores numéricos con decimales	1.2, -5.3, etc	Se maneja cualquier cantidad de decimales	0.0
Booleano	bool	Admite valores que indican verdadero o falso	true o false	Si se asigna un valor booleano a un entero se tomará como 1 o 0 respectivamente	true
Carácter	char	Tipo de dato que únicamente aceptará un único carácter, y estará delimitado por comillas simples	'a', 'b', 'c', etc	Se permitirá cualquier carácter entre las comillas simples, incluyendo las secuencias de escape	'\u0000' (carácter 0)
Cadena	string	Es un grupo o conjunto de caracteres que pueden tener cualquier carácter, y este se encontrará delimitado por comillas dobles. "	"cadena"	Se permitirá cualquier carácter entre las comillas dobles, incluyendo las secuencias de escape	"" (string vacío)
Nulo	null	El tipo de dato null se utiliza para representar un valor desconocido o no aplicable.	null		null

5.4 Secuencias de escape

Las secuencias de escape se utilizan para definir ciertos caracteres especiales dentro de cadenas de texto. Las secuencias de escape disponibles son las siguientes:

Secuencia	Descripción	Ejemplo
\n	Salto de línea	"Hola\nMundo"
\\	Barra invertida	"C:\\miCarpeta"
\"	Comilla doble	"\"esto es una cadena\""
\t	Tabulación	"\tEsto es una tabulación"
\'	Comilla simple	"\'Estas son comillas simples\'"

5.5 Operadores Aritméticos

5.5.1 Suma

Es la operación aritmética que consiste en realizar la suma entre dos o más valores. Para esta se utiliza el signo más (+).

Especificaciones de la operación suma

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

+	Entero	Decimal	Boolean	Carácter	Cadena
Entero	Entero	Decimal	Entero	Entero	Cadena
Decimal	Decimal	Decimal	Decimal	Decimal	Cadena
Boolean	Entero	Decimal			Cadena
Carácter	Entero	Decimal		Cadena	Cadena
Cadena	Cadena	Cadena	Cadena	Cadena	Cadena

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.2 Resta

Es la operación aritmética que consiste en realizar la resta entre dos o más valores. Para esta se utiliza el signo menos (-).

Especificaciones de la operación resta

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

-	Entero	Decimal	Boolean	Carácter
Entero	Entero	Decimal	Entero	Entero
Decimal	Decimal	Decimal	Decimal	Decimal
Boolean	Entero	Decimal		
Carácter	Entero	Decimal		

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.3 Multiplicación

Es la operación aritmética que consiste en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). El signo para representar la operación es el asterisco (*).

Especificaciones de la operación multiplicación

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

*	Entero	Decimal	Carácter
Entero	Entero	Decimal	Entero
Decimal	Decimal	Decimal	Decimal
Carácter	Entero	Decimal	

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.4 División

Es la operación aritmética que consiste en partir un todo en varias partes, al todo se le conoce como dividendo, al total de partes se le llama divisor y el resultado recibe el nombre de cociente. El operador de la división es la diagonal (/).

Especificaciones de la operación división

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

/	Entero	Decimal	Carácter
Entero	Decimal	Decimal	Decimal
Decimal	Decimal	Decimal	Decimal
Carácter	Decimal	Decimal	

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.5 Potencia

Es una operación aritmética de la forma a^b , donde a es el valor de la base y b es el valor del exponente que nos indicará cuantas veces queremos multiplicar el mismo número. Por ejemplo 5^3 , $a=5$ y $b=3$ tendríamos que multiplicar 3 veces 5 para obtener el resultado final; $5 \times 5 \times 5$ que da como resultado 125.

Especificaciones de la operación potencia

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

$^$	Entero	Decimal
Entero	Entero	Decimal
Decimal	Decimal	Decimal

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.6 Raíz

Es una operación aritmética de la forma $a \sqrt[b]{}$, donde a es el valor de la base y b es el valor del índice de la raíz. Por ejemplo $5 \sqrt[3]{}$, $a = 5$ y $b=3$ representa la raíz cúbica de 5.

Especificaciones de la operación raíz

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

\$	Entero	Decimal
Entero	Decimal	Decimal
Decimal	Decimal	Decimal

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.6 Módulo

Es una operación aritmética que obtiene el resto de la división de un número entre otro. Para realizar la operación se utilizará el signo (%).

Especificaciones de la operación módulo

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

%	Entero	Decimal
Entero	Decimal	Decimal
Decimal	Decimal	Decimal

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.5.7 Negación Unaria

Es una operación que niega el valor de un número, es decir que devuelve el contrario del valor original. Se utiliza el símbolo menos (-).

Especificaciones de la operación negación

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

- exp	Resultado
Entero	Entero
Decimal	Decimal

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico.

5.6 Operadores Relacionales

Son los símbolos que tienen como finalidad comparar expresiones, dando como resultado valores booleanos.

Operador	Descripción	Ejemplo
==	Igualación: compara ambos valores y verifica si son iguales - Iguales → True - No iguales → False	1 == 1 "hola" == "hola" 25.654 == 54.34
!=	Diferenciación: compara ambos lados y verifica si son distintos - Iguales → False - No iguales → True	1 != 2 var1 != var2 50 != 30
<	Menor que: compara ambos lados y verifica si el derecho es mayor que el izquierdo. - Derecho mayor → True - Izquierdo mayor → False	25.5 < 30 54 < 25 50 < 'F'
<=	Menor o igual que: Compara ambos lados y verifica si el derecho es mayor o igual que el izquierdo. - Derecho mayor o igual → True - Izquierdo mayor → False	25.5 <= 30 54 <= 25 50 <= 'F'
>	Mayor que: compara ambos lados y verifica si el izquierdo es mayor que el derecho. - Derecho mayor → False - Izquierdo mayor → True	25.5 > 30 54 > 25 50 > 'F'

>=	Mayor o igual que: Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho. - Derecho mayor → False - Izquierdo mayor o igual → True	25.5 >= 30 54 >= 25 50 >= 'F'
--------------	---	-------------------------------------

Especificaciones de los operadores relacionales

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

	Entero	Decimal	Boolean	Carácter	Cadena	Nulo
Entero						
Decimal						
Boolean						
Carácter						
Cadena						
Nulo						

Nota: Cualquier otra combinación no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

5.7 Operador ternario

El operador ternario es un operador que hace uso de 3 operandos para simplificar la instrucción “if”. El primer operando del operador ternario corresponde a la condición que debe de cumplir una expresión para que el operador retorne como valor el resultado de la expresión segundo operando del operador y en caso de no cumplir con la expresión el operador debe de retornar el valor de la expresión del tercer operando del operador.

if (<CONDICIÓN>) <EXPRESION> ':' <EXPRESION>




// Ejemplo

let edad: int = 18;

let mensaje: string = if (edad > 17) “legal”: “ilegal”;

5.8 Operadores Lógicos

Son los símbolos que tienen como finalidad comparar expresiones a nivel lógico (verdadero o falso). A continuación, se definen los símbolos que serán aceptados dentro del lenguaje:

Operador	Descripción	Ejemplo	Observaciones
	OR : compara expresiones lógicas y si al menos una es verdadera, entonces devuelve verdadero y en otro caso retorna falso.	bandera 5<2 Devuelve true	bandera es true
	AND : compara expresiones lógicas y si ambas son verdaderas, entonces devuelve verdadero y en otro caso retorna falso.	flag && flag Devuelve true	flag es true
	NOT : devuelve el valor inverso de una expresión lógica si esta es verdadera entonces devolverá false, de lo contrario retorna verdadero	!var Devuelve false	var es true

5.9 Signos de Agrupación

Los signos de agrupación serán utilizados para agrupar operaciones aritméticas, lógicas o racionales. Los símbolos de agrupación están dados por (y).

5.10 Precedencia de Operaciones

La precedencia de operadores nos indica la importancia de que una operación debe realizarse por encima del resto. A continuación, se define la misma:

Nivel	Operador	Asociatividad
0	-	Derecha
1	^, \$	No asociativa
2	*, /, %	Izquierda
3	+, -	Izquierda

4	==, !=, <, <=, >, >=	Izquierda
5	!	Derecha
6	&&	Izquierda
7		Izquierda

Nota: el nivel 0 es el nivel de mayor importancia

5.11 Caracteres de finalización y encapsulamiento de sentencias

El lenguaje se verá restringido por dos reglas que ayudan a finalizar una instrucción y encapsular sentencias:

- **Finalización de instrucciones:** para finalizar una instrucción se utilizará el signo ;
- **Encapsular sentencias:** para encapsular sentencias dadas por ciclos, métodos, funciones, etc, se utilizará los signos { y }.

// Ejemplo de finalización de instrucciones

```
int edad = 18;
```

// Ejemplo de encapsulamiento de sentencias

```
if(1==1){
```

```
    int a = 10;
```

```
    int b = 20;
```

```
}
```

5.12 Declaración de Variables

Una variable deberá de ser declarada antes de poder ser utilizada. Todas las variables tendrán un tipo de dato y un nombre de identificador. Las variables podrán ser declaradas global y localmente.

Durante la declaración de variables también se tendrá la opción de poder crear múltiples variables al mismo tiempo, al crear múltiples variables al mismo tiempo se tendrá la opción de crear todas las variables con un mismo valor, para ello se realizará una asignación al final del listado de las variables, en caso de no indicar esta asignación se dejará con valor nulo.

Las variables no pueden cambiar de tipo de dato, se deben mantener con el tipo declarado inicialmente, por lo que se debe de validar que el tipo de la variable y el valor sean compatibles.

5.12.1 Variables mutables

A este tipo de variables se les podrá modificar su valor después de que hayan sido declaradas.

```
Let identificador : <TIPO> ;  
Let id1, id2, id3 : <TIPO> ;  
Let identificador : <TIPO> = <EXPRESION>;  
Let id1, id2, id3 : <TIPO> = <EXPRESION>;
```

// Ejemplos

```
let numero : int;  
let var1, var2, var3: int;  
let cadena: string = "hola";  
let c1, c2, c3 : char = '1';
```

5.12.2 Variables inmutables

Estas variables no permiten una reasignación de valores, es decir, la variable almacena el valor con el que fue declarado, en caso contrario deberá indicar un error.

```
const identificador : <TIPO> ;  
const id1, id2, id3 : <TIPO> ;  
const identificador : <TIPO> = <EXPRESION>;  
const id1, id2, id3 : <TIPO> = <EXPRESION>;
```

// Ejemplos

```
const numero : int;  
const var1, var2, var3: int;  
const cadena: string = "hola";  
const c1, c2, c3 : char = '1';
```

5.13 Casteos

Los casteos son una forma de indicar al lenguaje que convierta un tipo de dato en otro, por lo que, si queremos cambiar un valor a otro tipo, es la forma adecuada de hacerlo. Para hacer esto, se colocará la palabra reservada del tipo de dato destino entre paréntesis seguido de una expresión.

```
CAST(<EXPRESION> AS <TIPO>)
```

El lenguaje aceptará los siguientes casteos:

- int a double
- double a int
- int a string
- int a char
- double a string
- char a int
- char a double

// Ejemplos

```
let e:int = cast(18.6 as int); // toma el valor entero de 18
```

```
let l:char = cast(70 as char); // toma el valor 'F' ya que el 70 en ascii es F
```

```
let n:double = cast(16 as double); // toma el valor de 16.0
```

5.14 Incremento y Decremento

Los incrementos y decrementos nos ayudan a realizar la suma o resta continua de un valor de uno en uno, es decir si incrementamos una variable, se incrementará de uno en uno, mientras que, si realizamos un decremento, hará la operación contraria.

```
<EXPRESION> ++ ;
```

```
<EXPRESION> -- ;
```

// Ejemplos

```
let edad:int = 18;
```

```
edad++; // tiene el valor de 19
```

```
edad--; // tiene el valor de 18
```

5.15 Vectores

Los vectores son una estructura de datos de tamaño fijo que pueden almacenar valores de forma limitada, y los valores que pueden almacenar son de un único tipo; int, double, bool, char o string. **El lenguaje permitirá únicamente el uso de arreglos de una o dos dimensiones.**

Observaciones:

- La posición de cada vector será N-1. Por ejemplo, si se quiere acceder al primer valor de un vector se accede como vector[0]

5.15.1 Declaración de Vectores

Al momento de declarar un vector, tenemos dos tipos que son:

- **Declaración tipo 1:** en esta declaración, se indica por medio de una expresión numérica del tamaño que se desea el vector colocando como nulo cada posición .
- **Declaración tipo 2:** En esta declaración, se indica por medio de una lista de valores separados por coma, los valores que tendrá el vector, en este caso el tamaño del vector será el de la misma cantidad de valores de la lista.

// Declaración de tipo 1

```
let <ID> : <TIPO> [ ] = new vector <TIPO> [ <EXPRESION> ] ;
```

```
let <ID> :<TIPO> [ ] [ ]=new vector <TIPO> [ <EXPRESION> ] [ <EXPRESION> ] ;
```

// Declaración de tipo 2

```
let <ID> : <TIPO> [ ] = [ <LISTA_VALORES> ] ;
```

```
let <ID> : <TIPO> [ ] [ ] = [ <LISTA_VALORES2> ] ;
```

// Ejemplo

```
let vector1: int[] = new vector int[4];
```

```
let vector2:char[][] = new vector char[4][4];
```

```
let vector3:string[] = ["Hola", "Mundo"];
```

```
let vector4:int [][] = [ [1, 2], [3, 4] ];
```

5.15.2 Acceso a vectores

Para acceder al valor de una expresión de un vector, se colocará el nombre del vector seguido de [EXPRESION].

<ID> [<EXPRESION>]

// Ejemplos

```
let vector3:string[] = ["Hola", "Mundo"];
```

```
let valor3:string = vector3[0]; // Almacena el valor "hola"
```

```
let vector4: int [][] = [ [1, 2], [3, 4] ];
```

```
let valor4: int = vector4[0][0]; // Almacena el valor 1
```

5.15.3 Modificación de Vectores

Para modificar el valor de una posición de un vector, se debe colocar el nombre del vector seguido de '[' EXPRESION ']' = EXPRESION;

Observaciones:

- A una posición de un vector se le puede asignar el valor de otra posición de otro vector o del mismo vector.
- A una posición de un vector se le puede asignar el valor de una posición de una lista.

<ID> [<EXPRESION>] = <EXPRESION> ;

<ID> [<EXPRESION>] [<EXPRESION>] = <EXPRESION> ;

// Ejemplo

```
let vector3: string [] = ["Hola", "Mundo"];
```

```
vector3[0] = "OLC1";
```

```
vector3[1] = "2do Semestre";
```

5.16 Sentencias de control

Estas sentencias modifican el flujo del programa introduciendo condicionales o saltos en la ejecución del código. Son esenciales para tomar decisiones dentro del código. Las sentencias de control para el programa son el IF y el SWITCH.

5.16.1 Sentencia IF

La sentencia if ejecuta las instrucciones sólo si se cumple una condición. Si la condición es falsa, se omiten las sentencias dentro de la sentencia. Esta sentencia incluye las instrucciones if, else if y else.

```
if ( <EXPRESION> ) { <INSTRUCCIONES> }  
|  
if ( <EXPRESION> ) { <INSTRUCCIONES> } else {  
<INSTRUCCIONES> }  
|  
  
if ( <EXPRESION> ) { <INSTRUCCIONES> } else <IF>  
  
/*  
Son 3 variantes en total  
1. IF  
2. IF-ELSE  
3. IF-ELSE IF  
*/
```

5.16.2 Switch case

Switch case es una estructura utilizada para agilizar la toma de decisiones múltiples, trabaja de la misma manera que lo harían sucesivos if.

5.16.2.1 Switch

Estructura principal del switch, donde se indica la expresión a evaluar.

```
switch ( <EXPRESION> ) {  
    <LISTA_CASOS>  
    <DEFAULT>  
}  
|  
switch ( <EXPRESION> ) {  
    <LISTA_CASOS>
```

```
}  
|  
switch ( <EXPRESION> ) {  
    <DEFAULT>  
}
```

5.16.2.2 Case

Estructura que contiene las diversas opciones a evaluar con la expresión establecida en el switch

```
case <EXPRESION> :  
    <INSTRUCCIONES>
```

5.16.2.2 Default

Estructura que contiene las sentencias si en dado caso no haya salido del switch por medio de una sentencia **break**.

```
default :  
    <INSTRUCCIONES>
```

Observaciones:

- Si la cláusula "case" no posee ninguna sentencia "break", al terminar todas las sentencias del case ingresado, el lenguaje seguirá evaluando las demás opciones.

5.17 Sentencias Cíclicas

Los ciclos o bucles son una secuencia de instrucciones de código que se ejecutan una vez tras otra mientras la condición, que se ha asignado para que pueda ejecutarse, sea verdadera. En el lenguaje actual, se podrán realizar 3 sentencias cíclicas que se describen a continuación.

Observaciones:

- Es importante destacar que pueden tener ciclos anidados entre las sentencias a ejecutar.
- También, entre las sentencias pueden tener ciclos diferentes anidados

5.17.1 While

El ciclo o bucle While, es una sentencia que ejecuta una secuencia de instrucciones mientras la condición de ejecución se mantenga verdadera.

```
while ( <EXPRESION> ) {  
    <INSTRUCCIONES>  
}
```

5.17.2 For

El ciclo o bucle for, es una sentencia que nos permite ejecutar N cantidad de veces la secuencia de instrucciones que se encuentra dentro de ella.

Observaciones:

- Para la actualización de la variable del ciclo for se puede utilizar
 - **Incremento o Decremento:** $i++$ o $i--$
 - **Asignación:** como $i = i+1$, $i = i-1$, etc, es decir, cualquier tipo de asignación
- Dentro pueden venir N instrucciones

```
for ( <DECLARACION> ; <CONDICIÓN> ; <ACTUALIZACIÓN> ) {  
    <INSTRUCCIONES>  
}  
  
for ( <ASIGNACIÓN> ; <CONDICIÓN> ; <ACTUALIZACIÓN> ) {  
    <INSTRUCCIONES>  
}
```

5.17.3 Do-Until

El ciclo o bucle Do-Until, es una sentencia que ejecuta al menos una vez el conjunto de instrucciones que se encuentra dentro de ella y que se ejecuta hasta que la condición sea verdadera.

Observaciones:

- Dentro pueden venir N instrucciones

```
do {  
    <INSTRUCCIONES>  
} until ( <EXPRESION> ) ;
```

5.17.4 Loop

Este ciclo o bucle se comporta como un ciclo infinito.

Observaciones:

- Dentro pueden venir N instrucciones
- La forma para salir de este ciclo es mediante una instrucción “break” o “return” que se definen en la siguiente sección

```
loop {  
    <INSTRUCCIONES>  
}
```

5.18 Sentencias de Transferencia

Las sentencias de transferencia nos permiten manipular el comportamiento de los bucles, ya sea para detenerlo o para saltarse algunas iteraciones. El lenguaje soporta las siguientes sentencias:

5.18.1 Break

La sentencia break hace que se salga del ciclo inmediatamente, es decir que el código que se encuentre después del break en la misma iteración no se ejecutara y este se saldrá del ciclo.

```
break ;
```

5.18.2 Continue

La sentencia continue puede detener la ejecución de la iteración actual y saltar a la siguiente. La sentencia continue siempre debe de estar dentro de un ciclo, de lo contrario será un error

```
continue ;
```


5.18.3 Return

La sentencia return finaliza la ejecución de un método o función y puede especificar un valor para ser devuelto a quien llama a la función.

```
return ;  
return <EXPRESION> ;
```

5.19 Funciones

Una función es una subrutina de código que se identifica con un nombre, tipo y un conjunto de parámetros. Para este lenguaje las funciones serán declaradas definiendo primero su tipo, luego un identificador único, seguido de una lista de parámetros dentro de paréntesis (esta lista de parámetros puede estar vacía en el caso de que la función no utilice parámetros).

Cada parámetro debe consistir en un identificador seguido de su tipo, y opcionalmente, un valor inicial por defecto. Si se proporciona un valor por defecto, el parámetro se puede omitir al llamar a la función. Para el caso de que sean varios parámetros se debe utilizar comas para separar cada parámetro y en el caso de que no se usen parámetros no se deberá incluir nada dentro de los paréntesis. Luego de definir la función y sus parámetros se declara el cuerpo de la función, el cual es un conjunto de instrucciones delimitadas por llaves {}.

Para las funciones es obligatorio que las mismas posean un valor de retorno que coincida con el tipo con el que se declaró la función, en caso de que no sea el mismo tipo o de que no venga un retorno dentro del cuerpo de la función debería lanzarse un error de tipo semántico.

```
function <TIPO> <ID> ( <PARAMETROS> ) {  
    <INSTRUCCIONES>  
}  
  
PARAMETROS -> PARAMETROS , PARAMETRO  
            | PARAMETRO  
  
PARAMETRO -> <ID> : <TIPO> = <EXPRESION>  
            | <ID> : <TIPO>
```

```
//Ejemplos
function int conversion (size:double, tipo: string){
    if(tipo=="metro"){
        return size/3*3.281;
    } else{
        return -1;
    }
}

function double calculaFuerza(masa:double, aceleracion:double=9.8){
    return masa*aceleracion;
}
```

Cabe destacar que **no habrá sobrecarga de funciones y métodos** dentro de este lenguaje por lo que solo puede existir una función o método con el id declarado por lo que si se crea otra función o método con un id previamente utilizado esto debe de generar un error de tipo semántico.

5.20 Métodos

Un método también es una subrutina de código que se identifica con un nombre y un conjunto de parámetros, aunque a diferencia de las funciones estas subrutinas no deben de retornar un valor. Para este lenguaje los métodos serán declarados haciendo uso de la palabra reservada 'void', seguido de un identificador del método, seguido de una lista de parámetros dentro de paréntesis (esta lista de parámetros puede estar vacía en el caso de que la función no utilice parámetros).

Cada parámetro debe consistir en un identificador seguido de su tipo, y opcionalmente, un valor inicial por defecto. Si se proporciona un valor por defecto, el parámetro se puede omitir al llamar a la función. Para el caso de que sean varios parámetros se debe utilizar comas para separar cada parámetro y en el caso de que no se usen parámetros no se deberá incluir nada dentro de los paréntesis. Luego de definir la función y sus parámetros se declara el cuerpo de la función, el cual es un conjunto de instrucciones delimitadas por llaves {}.

```
function void <ID> ( <PARAMETROS> ) {  
    <INSTRUCCIONES>  
}  
  
PARAMETROS -> PARAMETROS , PARAMETRO  
              | PARAMETRO  
  
PARAMETRO -> <ID> : <TIPO> = <EXPRESION>  
              | <ID> : <TIPO>
```

Cabe destacar que **no habrá sobrecarga de funciones y métodos** dentro de este lenguaje por lo que solo puede existir una función o método con el id declarado por lo que si se crea otra función o método con un id previamente utilizado esto debe de generar un error de tipo semántico.

5.21 Llamadas

La llamada a una función devuelve un resultado que ha de ser recogido, bien asignándole a una variable del tipo adecuado, bien integrándose en una expresión.

La sintaxis de las llamadas de los métodos y funciones será la misma.

```
LLAMADA -> <ID> ( <PARAMETROS_2> )  
           | <ID> ( )  
  
PARAMETROS_2 -> PARAMETROS_2, <PARAMETRO_2 >  
                | <PARAMETRO_2 >  
  
PARAMETRO_2 -> <ID> = <EXPRESION>
```

Observaciones:

- Si el parámetro tiene valor por defecto en la llamada puede omitirse en la llamada o bien se puede sobrescribir su valor.
- Al momento de ejecutar cualquier llamada, no se diferenciarán entre métodos y funciones, por lo tanto, podrá venir una función que retorne un valor como un método, pero la expresión retornada no se asignará a ninguna variable.

- Se podrán podrán llamar métodos y funciones antes que se encuentren declaradas, para ello se recomienda realizar 2 pasadas del AST generado: La primera para almacenar todas las funciones, y la segunda para las variables globales y la función ejecutar.
- Para la llamada a métodos como una instrucción se deberá de agregar el punto y coma (;) como una instrucción.

5.21 Función echo

Esta función nos permite imprimir expresiones agregando un salto de línea al final del contenido.

```
echo <EXPRESION> ;
```

```
//Ejemplo
```

```
echo "hola Mundo";
```

```
let variable1: int = 10;
```

```
echo variable1;
```

```
/*
```

```
hola Mundo
```

```
10
```

```
*/
```

5.22 Funciones Nativas

5.22.1 Lower

Esta función recibe como parámetro una expresión de tipo cadena y retorna una nueva cadena con todas las letras minúsculas.

```
lower ( <EXPRESION> ) ;
```

```
//Ejemplo
```

```
let cadena: string = lower("HOLa MundO"); //almacena 'hola mundo'
```

5.22.2 Upper

Esta función recibe como parámetro una expresión de tipo cadena y retorna una nueva cadena con todas las letras mayúsculas.

```
upper ( <EXPRESION> ) ;
```

//Ejemplo

```
let cadena: string = upper("HOLa"); //almacena 'HOLA'
```

5.22.3 Round

Esta función recibe como parámetro un valor numérico. Permite redondear los números decimales según las siguientes reglas:

- Si el decimal es mayor o igual a 0.5, se aproxima al entero superior.
- Si el decimal es menor que 0.5, se aproxima al número inferior.

```
round ( <EXPRESION> ) ;
```

//Ejemplo

```
let valor : int = round(15.51); //almacena 16
```

```
let valor2: int = round(9.40); //almacena 9
```

5.23.4 Length

Esta función recibe como parámetro un vector o una cadena y devuelve el tamaño de este.

Observación:

- Si se usa para otro parámetro de tipo de dato no especificado, se considera un error semántico.

```
len ( <EXPRESION> ) ;
```

//Ejemplos

```
let vector:string[] = ["Hola", "Mundo"];
```

```
let cadena:string = "compi1";
```

```
let valor : int = len(vector); //Almacena 2
```

```
let valor2: int = len(cadena); //Almacena 6
```

5.23.5 Truncate

Trunca un número decimal para eliminar los decimales.

```
truncate ( <EXPRESION> ) ;
```

//Ejemplo

```
let valor : int = truncate (15.51); //almacena 15
```

```
let valor2: int = truncate (9.40); //almacena 9
```

5.23.6 Función Is

Esta función retorna verdadero si el tipo de dato coincide con el evaluado

```
<EXPRESION> is <TIPO>
```

//Ejemplo

```
let valor : bool = 10 is int; //almacena true
```

```
let valor2: bool = "hola mundo" is int; //almacena false
```

5.25.7 ToString

Esta función permite convertir un valor de tipo numérico o bool a texto.

```
toString ( <EXPRESION> ) ;
```

5.25.8 toCharArray

Esta función permite convertir una cadena en un vector de caracteres.

```
toCharArray ( <EXPRESION> ) ;
```

5.25.9 reverse

Este método invierte el orden de los elementos en el vector.

- Para vectores de tipo int, double, y char, simplemente se invierte el orden de los elementos.
- Para vectores de tipo bool, el método invertirá el orden de los valores de true y false.

- Para vectores de tipo string, se invierte el orden de las cadenas dentro del vector, pero no el contenido interno de cada cadena. Es decir, la primera cadena pasará a la última posición, la segunda cadena a la penúltima, y así sucesivamente, pero las letras dentro de cada cadena no se modificarán.

```
reverse(<ID>);
```

5.25.10 max

- Para vectores de tipo int y double, se devuelve el valor máximo numérico.
- Para vectores de tipo char, se devuelve el carácter con el mayor valor ASCII.
- Para vectores de tipo bool, se considera que true es mayor que false, por lo que se devolverá true si está presente.
- Para vectores de tipo string, se devuelve la cadena que sea mayor en orden lexicográfico (alfabéticamente).

```
max(<ID>);
```

5.25.11 min

- Para vectores de tipo int y double, se devuelve el valor mínimo numérico.
- Para vectores de tipo char, se devuelve el carácter con el menor valor ASCII.
- Para vectores de tipo bool, se considera que false es menor que true, por lo que se devolverá false si está presente.
- Para vectores de tipo string, se devuelve la cadena que sea menor en orden lexicográfico (alfabéticamente).

```
min(<ID>);
```

5.25.12 sum

- Para vectores de tipo int y double, se suma el valor de todos los elementos.
- Para vectores de tipo char, se suma el valor ASCII de cada carácter.

- Para vectores de tipo bool, true se considera como 1 y false como 0, sumando los valores resultantes.
- Para vectores de tipo string, el método concatenar todas las cadenas en una sola.

```
sum(<ID>);
```

5.25.13 average

- Para vectores de tipo int y double, se calcula el promedio de los valores.
- Para vectores de tipo char, se calcula el promedio de los valores ASCII de los caracteres.
- Para vectores de tipo bool, el promedio se calcula tomando true como 1 y false como 0.
- Para vectores de tipo string, este método no es aplicable y debe lanzar un error de tipo.

```
average(<ID>);
```

5.26 Función Ejecutar

Para poder ejecutar todo el código generado dentro del lenguaje, se utilizará la sentencia EJECUTAR para poder indicar qué método es el que iniciará con la lógica del programa.

```
ejecutar <ID> ( ) ;  
ejecutar <ID> ( <PARAMETROS> ) ;
```

Nota: Solo se aceptaran métodos ya que no se espera ningún valor de retorno.

5.27 Estructura de las instrucciones

Fuera de las funciones no se permitirá manejar instrucciones ejecutables, lo único permitido será declaración o asignación de variables, constantes y vectores, estas deberán ser accesibles en cualquier punto del programa al igual que las funciones y métodos.

6. Reportes

Los reportes son parte fundamental de ComplInterpreter, ya que muestra de forma visual las herramientas utilizadas para realizar la ejecución del código.

A continuación, se muestran ejemplos de estos reportes. Queda a discreción del estudiante el diseño de estos, solo se pide que sean totalmente legibles.

6.1 Tabla de errores

El reporte de errores debe contener la información suficiente para detectar y corregir errores en el código fuente.

#	Tipo	Descripción	Línea	Columna
1	Léxico	El carácter "\$" no pertenece al lenguaje	5	3
2	Sintáctico	Se encontró Identificador y se esperaba Expresión.	6	3
3	Semántico	No se puede realizar la resta entre CADENA y CADENA	8	10

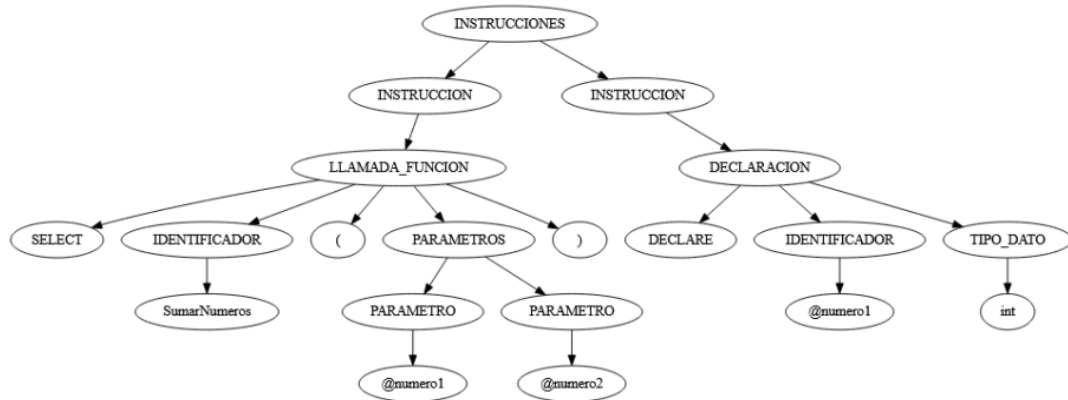
6.2 Tabla de Símbolos

Este reporte mostrará la tabla de símbolos después de la ejecución. Deberá mostrar las variables y arreglos declarados, así como su tipo, valor y toda la información que considere necesaria.

#	ID	Tipo	Tipo	Entorno	Valor	Línea	Columna
1	var	Variable	Entero	Funcion1	10	15	20
2	var2	Vector	Entero	Funcion2	[1, 2, 3]	20	13
3	var3	Variable	Bool	Funcion2	true	25	10

6.3 AST

Este reporte muestra el árbol de sintaxis producido al analizar los archivos de entrada. Este debe de representarse como un grafo. Se deben mostrar los nodos que el estudiante considere necesarios para describir el flujo realizado para analizar e interpretar sus archivos de entrada.



Nota: Se sugiere a los estudiantes utilizar la herramienta Graphviz para graficar su AST

6.4 Salidas en consola

La consola es el área de salida del intérprete. Por medio de esta herramienta se podrán visualizar las salidas generadas por la instrucción "echo", así como los errores léxicos, sintácticos y semánticos.

```

> Este es un mensaje desde mi interprete de compi 1.
>
>
---> Error léxico: Símbolo "#" no reconocido en línea 10 y columna 7
---> Error Semántico: Se ha intentado asignar un entero a una variable booleana
  
```

7. Requerimientos Mínimos

Para que el estudiante tenga derecho a calificación, deberá cumplir con lo siguiente:

- Interfaz Gráfica funcional
- Operaciones aritméticas, lógicas y relacionales
- Declaración y asignación de variables
- Sentencias de control
- Sentencias cíclicas
- Métodos (con o sin parámetros)
- Llamadas a métodos (con o sin parámetros)
- Instrucción echo
- Sentencia ejecutar
- Reporte AST
- Consola
- Documentación completa (manual de usuario, manual técnico y archivo de gramáticas)

8. Entregables

- **Código fuente del proyecto**
- **Manual de Usuario en un archivo pdf**
 - Capturas de pantalla detallando cómo funciona su entorno de trabajo y los reportes que se generan.
- **Manual Técnico en un archivo pdf**
 - Información importante del proyecto para que se pueda realizar el mantenimiento en el futuro. Especificar el lenguaje, herramientas utilizadas, métodos y funciones más importantes.
- **Archivo de Gramática en un archivo txt**
 - El archivo debe contener su gramática y debe de ser limpio, entendible y no debe ser una copia del archivo de Jison.
 - La gramática debe estar escrita en formato **BNF(Backus-Naur form)**.
 - Solamente se debe escribir la gramática independiente del contexto.

9. Restricciones

- La entrega debe ser realizada mediante UEDI enviando el enlace del **repositorio privado** de GitLab en donde se encuentra su proyecto.
- El nombre del repositorio de Gitlab debe ser **OLC1_Proyecto2_#Carnet**
- Se debe agregar al auxiliar encargado como colaborador al repositorio de Gitlab.
- Lenguaje de Programación a utilizar: **Javascript/TypeScript**
- Herramientas para el análisis léxico y sintáctico: **Jison**
- **El proyecto debe ser realizado de forma individual.**
- Para graficar se puede utilizar cualquier librería (Se recomienda graphviz)
- Puede utilizar un framework como Angular, React, Vuejs, etc, para generar su entorno gráfico. Queda a discreción del estudiante cuál utilizar.
- **Copias completas/parciales** de: código, gramática, etc. serán merecedoras de una **nota de 0 puntos**, los responsables serán reportados al catedrático de la sección y a la Escuela de Ciencias y Sistemas.

- La calificación tendrá una duración de 30 minutos, acorde al programa del laboratorio.
- Se debe visualizar todo desde el navegador

10. Fecha de Entrega

Domingo, 20 de octubre de 2024 a las 23:59. La entrega será por medio de la plataforma UEDI. Entregas fuera de la fecha indicada, no se calificarán.

SE LE CALIFICARA DEL ÚLTIMO COMMIT REALIZADO ANTERIOR A ESTA FECHA.

11. Anexos

11.1 Ejemplo de archivo de entrada

```
// Comentarios de una línea y multilínea
// Esto es un comentario de una sola línea
/* Este es un comentario
de múltiples líneas */

// Declaración de variables globales
let x: int = 10;
let y: double = 5.5;
let mensaje: string = "Hola Mundo";
let suma: double;
let resta: int;
let potencia: double;
let modulo: int;
let esMayor: bool;
let resultado: string;

// Función principal
function void main() {
    // Operaciones aritméticas
    suma = x + y;
    resta = x - 3;
    potencia = x ^ 2;
    modulo = x % 3;

    // Uso de operadores relacionales y lógicos
```

```
esMayor = x > y && x != 0;
```

```
// Condicionales con IF
```

```
if (esMayor) {  
    echo "X es mayor que Y y no es cero";  
} else {  
    echo "X no es mayor que Y o es cero";  
}
```

```
// Uso de un operador ternario
```

```
resultado = if (suma > 15) "Mayor a 15" : "Menor o igual a 15";
```

```
// Ciclo FOR
```

```
for (let i: int = 0; i < 5; i++) {  
    echo "El valor de i es: " + i;  
}
```

```
// Llamadas a funciones y métodos
```

```
let resultadoSuma: int = sumaNumeros(a=3, b=7);  
    imprimirMensaje(mensaje = "La suma de 3 y 7 es: " +  
resultadoSuma);  
}
```

```
// Funciones y métodos
```

```
function int sumaNumeros(a: int, b: int) {  
    return a + b;  
}
```

```
function void imprimirMensaje(mensaje: string) {  
    echo mensaje;  
}
```

```
// Sentencia ejecutar
```

```
ejecutar main();
```