

**Universidad de San Carlos de Guatemala**  
**Facultad de Ingeniería**  
**Escuela de Ciencias y Sistemas**  
**Organización de Lenguajes y Compiladores 1**  
**Vacaciones de Diciembre 2024**



**USAC**  
**TRICENTENARIA**  
Universidad de San Carlos de Guatemala

**Catedrático:**

Ing. Mario Bautista

**Tutor académico:**

Fabian Reyna

# CompScript

## Tabla de Contenido

<b>1. Objetivo General</b>	<b>4</b>
<b>2. Objetivos específicos</b>	<b>4</b>
<b>3. Descripción General</b>	<b>4</b>
<b>4. Entorno de Trabajo</b>	<b>5</b>
4.1 Editor	5
4.2 Funcionalidades	5
4.3 Características	5
4.4 Herramientas	5
4.5 Reportes	5
4.6 Área de Consola	6
<b>5. Descripción del Lenguaje</b>	<b>6</b>
5.1 Case Insensitive	6
5.2 Comentarios	6
5.2.1 Comentarios de una línea	6
5.2.2 Comentarios multilínea	7
5.3 Tipos de Dato	7
5.4 Secuencias de escape	8
5.5 Operadores Aritméticos	8
5.5.1 Suma	8
5.5.2 Resta	9
5.5.3 Multiplicación	9
5.5.4 División	10
5.5.5 Potencia	10
5.5.6 Raíz	11
5.5.7 Módulo	11
5.5.8 Negación Unaria	12
5.6 Operadores Relacionales	12
5.7 Operadores Lógicos	13
5.8 Signos de Agrupación	14
5.9 Precedencia de Operaciones	14
5.10 Caracteres de finalización y encapsulamiento de sentencias	15
5.11 Declaración de variables	15
5.12 Asignación de variables	16
5.13 Casteos	16
5.14 Incremento y Decremento	17
5.15 Sentencias de control	17
5.15.1 Sentencia IF	17
5.15.2 Sentencia Match	18
5.16 Sentencias Cíclicas	19
5.16.1 While	19

5.16.2 For	19
5.16.3 Do-While	20
5.17 Sentencias de Transferencia	20
5.17.1 Break	20
5.17.2 Continue	21
5.17.3 Return	21
5.18 Vectores	22
5.18.1 Declaración de Vectores	22
5.18.2 Acceso a vectores	22
5.18.3 Asignación de Vectores	23
5.19 Listas Dinámicas	23
5.19.1 Declaración de Listas	23
5.19.2 Append	24
5.19.3 Acceso a listas	24
5.19.4 Asignación de Listas	24
5.19.5 Remove	24
5.19.6 Pop	25
5.19.7 Reverse	25
5.20 Structs	25
5.20.1 Declaración de Structs	25
5.20.2 Instanciación de Struct	26
5.20.3 Acceso de struct	26
5.20.4 Asignación de struct	26
5.21 Funciones	27
5.22 Métodos	28
5.23 Llamadas	29
5.24 Impresión en consola	30
5.25 Función round	30
5.26 Length	31
5.27 ToString	31
5.29 Función RUN_MAIN	32
<b>6. Reportes</b>	<b>32</b>
6.1 Tabla de tokens	32
6.2 Tabla de errores	32
6.3 AST	33
6.4 Tabla de Símbolos	33
6.4 Salidas en consola	33
<b>7. Restricciones</b>	<b>34</b>
<b>8. Entrega</b>	<b>34</b>

## 1. Objetivo General

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

## 2. Objetivos específicos

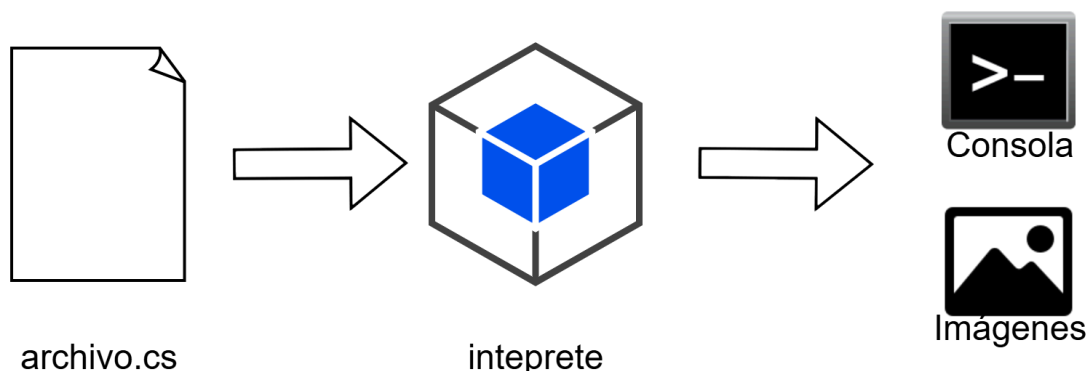
- Reforzar los conocimientos de análisis léxico, sintáctico y semántico para la creación de un lenguaje de programación.
- Aplicar los conceptos de compiladores para implementar un proceso de interpretación de código de alto nivel.
- Construir un árbol de sintaxis abstracta (AST) para la ejecución de código de alto nivel.
- Detectar y reportar errores léxicos, sintácticos y semánticos.

## 3. Descripción General

El curso de Organización de Lenguajes y Compiladores 1, ha puesto en marcha un nuevo proyecto que consiste en crear un lenguaje de programación para poder utilizar en los proyectos de los cursos de Introducción a la Programación y Computación 1 y 2. Este lenguaje está fundamentado en los principales conceptos de programación, con un enfoque particular en el uso de estructuras de datos.

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

### Flujo de la aplicación



## 4. Entorno de Trabajo

### 4.1 Editor

La función principal del editor será el ingreso de código fuente que será analizado. Queda a discreción del estudiante el diseño.

### 4.2 Funcionalidades

- **Nuevo archivo:** El editor deberá tener la capacidad de eliminar todo el contenido actual del editor y la consola.
- **Abrir archivos:** El editor deberá abrir archivos .cs, cuyo contenido se mostrará en el área de entrada.
- **Guardar archivo:** El editor deberá guardar el estado del archivo en el que se está trabajando.

### 4.3 Características

- **Recuperación de errores:** el intérprete debe tener la capacidad de recuperarse de errores, ya sean léxicos, sintácticos o semánticos y poder terminar la ejecución del programa.

### 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 Tokens:** se mostrarán todos los tokens reconocidos por el analizador léxico en el último archivo ejecutado.
- **Reporte de Errores:** se mostrarán todos los errores encontrados al realizar el análisis léxico, sintáctico y semántico y deben mostrarse desde la interfaz.
- **Generar AST:** se debe generar el ast del último archivo analizado y mostrarlo desde la interfaz.
- **Reporte de Tabla de Símbolos:** se mostrarán todas las variables, vectores declarados dentro del flujo del programa y mostrarlo en la interfaz.

## 4.6 Área de Consola

En esta área se mostrarán los resultados, mensajes y todo lo que sea indicado dentro del lenguaje.

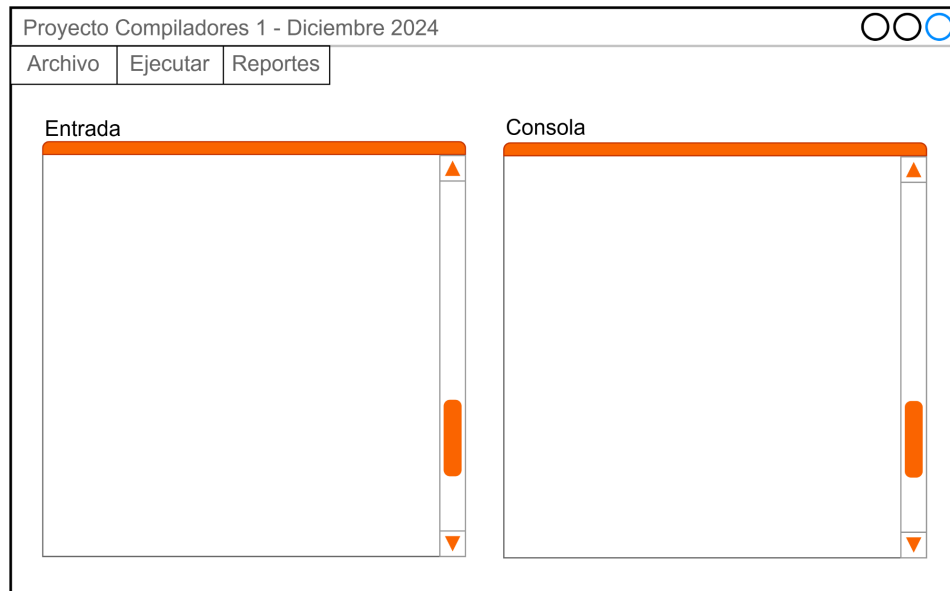


Figura 1. Propuesta de interfaz

## 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 = 1;  
LeT A :Int = 1;  
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 o 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 multilínea

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

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

```
/*  
Este es un  
comentario  
multilínea  
*/
```

### 5.3 Tipos de Dato

Los tipos de dato que soportará el lenguaje se definen a continuación:

Tipo	Definición	Descripción	Ejemplo	Observaciones	Default
Entero	int	Este tipo de dato aceptará solamente números enteros	1, -120, etc	No admite decimales.	0
Decimal	double	Admite valores numéricos con decimales	1.2, -00.34, etc	Se maneja cualquier cantidad de decimales	0.0
Booleano	bool	Admite valores que indican verdadero o falso	true, false		true
Carácter	char	Tipo de dato que únicamente aceptará un único carácter, y estará delimitado por comillas simples	'a', 'b', 'c', 'E', '1', '&', '\n', etc	Se permitirá cualquier carácter entre las comillas simples, incluyendo las <b>secuencias de escape</b>	'\u0000' (carácter 0)
Cadena	string	Es un conjunto de caracteres que pueden tener cualquier carácter, y este se encontrará delimitado por comillas dobles. “ “	“cadena”, “- cad”, etc	Se permitirá cualquier carácter entre las comillas dobles, incluyendo las <b>secuencias de escape</b>	“” (string vacío)

## 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	Booleano	Carácter	Cadena
Entero	Entero	Decimal	Entero	Entero	Cadena
Decimal	Decimal	Decimal	Decimal	Decimal	Cadena
Booleano	Entero	Decimal			Cadena
Carácter	Entero	Decimal		Cadena	Cadena
Cadena	Cadena	Cadena	Cadena	Cadena	Cadena

**Nota:** Cualquier combinación marcada con rojo u otra 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	Carácter
Entero	Entero	Decimal	Entero
Decimal	Decimal	Decimal	Decimal
Carácter	Entero	Decimal	

**Nota:** Cualquier combinación marcada con rojo u otra 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 combinación marcada con rojo u otra 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 combinación marcada con rojo u otra 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.

##### 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 combinación marcada con rojo u otra 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 \$ b** donde a es el valor de la base y b es el índice de la raíz. Por ejemplo 5 \$ 3, donde a = 5 y b = 3 representa la raíz cúbica de 5.

#### Especificaciones de la operación potencia

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 combinación marcada con rojo u otra no especificada en esta tabla es invalida y será considerado un error de tipo semántico.

### 5.5.7 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	Doble
Entero	Decimal	Decimal
Decimal	Decimal	Decimal

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

### 5.5.8 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 combinación marcada con rojo u otra no especificada en esta tabla es invalida 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. A continuación, se definen los símbolos que serán aceptados dentro del lenguaje:

Operador	Descripción	Ejemplo
==	<b>Igualación:</b> compara ambos valores y verifica si son iguales	1==1 25.654 == 54.34
!=	<b>Diferenciación:</b> compara ambos lados y verifica si son distintos	1 != 2 50 != 30
<	<b>Menor que:</b> compara ambos lados y verifica si el izquierdo es mayor que el derecho.	25.5 < 30 50 < 'F'
<=	<b>Menor o igual que:</b> Compara ambos lados y verifica si el derecho es mayor o igual que el izquierdo.	25.5 <= 30 50 <= 'F'
>	<b>Mayor que:</b> compara ambos lados y verifica si el izquierdo es mayor que el derecho.	25.5 > 30 50 > 'F'
>=	<b>Mayor o igual que:</b> Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho.	25.5 >= 30 50 >= 'F'

## Especificaciones de los operadores relacionales

A continuación, se especifica en una tabla los resultados que se deberán obtener con estas operaciones.

Relacional	Entero	Decimal	Booleano	Carácter	Cadena
Entero	Booleano	Booleano		Booleano	
Decimal	Booleano	Booleano		Booleano	
Booleano			Booleano		
Carácter	Booleano	Booleano		Booleano	
Cadena					Booleano

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

### 5.7 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
	<b>OR:</b> compara expresiones lógicas y si al menos una es verdadera, entonces devuelve verdadero y en otro caso retorna falso.	true    5<2 Devuelve true
&&	<b>AND:</b> compara expresiones lógicas y si ambas son verdaderas, entonces devuelve verdadero y en otro caso retorna falso.	true && "hola" == "hola" Devuelve true
!	<b>NOT:</b> devuelve el valor inverso de una expresión lógica si esta es verdadera entonces devolverá false, de lo contrario retorna verdadero	!true Devuelve false

## 5.8 Signos de Agrupación

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

```
let edad:int = (10 + 10) - 5;  
  
let bandera: bool = (5 == 5) == true;  
  
let bandera2: bool = (1 + 2) > (3 - 1);
```

## 5.9 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

## 5.10 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 ;

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

- **Encapsular sentencias:** para encapsular sentencias dadas por ciclos, métodos, funciones, etc, se utilizará los signos { y }.

```
if (true) {  
    let edad:int = 18;  
}
```

## 5.11 Declaración de variables

Las variables deben ser declaradas antes de su uso, especificando su mutabilidad, un nombre de identificador y un tipo de dato. Pueden declararse tanto a nivel global como local.

Aquellas variables definidas con la palabra reservada **const** conservarán su valor constante durante toda la ejecución, mientras que las declaradas con **let** podrán cambiar su valor. Si no se especifica un valor inicial, la variable tomará el valor por defecto correspondiente a su tipo de dato.

```
<MUTABILIDAD> <ID> : <TIPO> ;  
<MUTABILIDAD> <ID> : <TIPO> = <EXPRESION> ;
```

```
let num : int;  
const nota : double;  
  
let num2 : int = 10;  
const nota2 : double = 20.6;
```

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 Asignación de variables

Esta instrucción permitirá modificar el valor de la variable que fue previamente declarada, siempre y cuando sea posible modificarla.

```
let num : int;  
const nota : double;  
  
num = 10;  
nota = 20.6;
```

**Nota:** La asignación de num, actualizará el valor de esta de 0 (valor por defecto) a 10, mientras que nota no sería actualizada ya que fue declarada como const y debería ser un error semántico

### 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 char
- char a int
- char a double

```
let edad : int = cast(18.6 as int);  
const letra : char = cast(70 as char);  
let numero : double = cast(16 to double);
```

**Nota:**

- La variable edad tomaría el valor de 18.
- La variable letra tomaría el valor de 'F' ya que 70 es el ascii de F.
- La variable numero tomaría 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. Esta instrucción es válida para tipos numéricos

```
<EXPRESION> + + ;  
<EXPRESION> - - ;  
  
let edad : int = 18;  
edad++; // tiene el valor de 19  
edad--; // tiene el valor de 18
```

### 5.15 Sentencias de control

Estas sentencias modifican el flujo del programa introduciendo condicionales. Las sentencias de control para el programa son el IF y el Match.

#### 5.15.1 Sentencia IF

El lenguaje posee la sentencia IF similar a otros lenguajes de programación, la cual permite ejecutar bloques de código se ejecuten si la condición a evaluarse es verdadera. Esta sentencia se define por las instrucciones if, if else, if else if.

#### Consideraciones

- Las instrucciones else if y else son opcionales
- La instrucción else if se puede utilizar tantas veces como se desee

```
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
*/

```

### 5.15.2 Sentencia Match

El lenguaje posee la sentencia Match similar al switch case de otros lenguajes de programación, la cual es una estructura utilizada para agilizar la toma de decisiones múltiples, trabaja de la misma manera que lo harían sucesivos if y no requiere el uso de la sentencia break.

```

/* Estructura principal del match, donde se indica la expresión a
evaluar. */

```

```

match
<EXPRESION> {
    <CASES_LIST>
    <DEFAULT>
}

```

```

match
<EXPRESION> {
    <CASES_LIST>
}

```

```

match
<EXPRESION> {
    <DEFAULT>
}

```

```

/* Casos: Estructura que contiene las diversas opciones a evaluar
con la expresión establecida en el match */

```

```

<EXPRESION> => {
    <INSTRUCCIONES>
}

```

```

/* Default: Estructura que contiene las sentencias si en dado caso no
se ha encontrado coincidencias con las anteriores. */

```

```

default =>{
    <INSTRUCCIONES>
}

```

## 5.16 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:

- Dentro pueden venir N instrucciones o sentencias anidadas.

#### 5.16.1 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>  
}
```

#### 5.16.2 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 | Decremento:** `i++` | `i--`
  - **Asignación:** como `i = i+1`, `i = i-1`, etc, es decir, cualquier tipo de asignación
- Dentro pueden venir N instrucciones o sentencias anidadas
- Si la variable que se le realiza la asignación no existe se debe reportar como error semántico.

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

### 5.16.3 Do-While

El ciclo o bucle Do-While, es una sentencia que ejecuta al menos una vez el conjunto de instrucciones que se encuentran dentro de ella y que se sigue ejecutando mientras la condición sea verdadera.

#### Observaciones:

- Dentro pueden venir N instrucciones

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

## 5.17 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:

### 5.17.1 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 ;  
  
let i : int = 0;  
for(i = 0; i < 3; i++){  
    if (i==2){  
        break; // me salgo en i = 2 y ya no se ejecuta lo demás  
    }  
    console.log(i);  
}  
  
// Salida  
/*  
0  
1  
*/
```

### 5.17.2 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 ;**

```
let i: int = 0;
for(i = 0; i <= 2; i++){
    console.log(i);
    if (i==1){
        continue;
        /*me salte el resto de instrucciones u seguiré en i == 3 */
    }
    console.log(i*5);
}
// Salida
/*
0
0
1
2
10
*/
```

### 5.17.3 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> ;**

**// Ejemplos**  
return 10 \* 10;  
return ;

## 5.18 Vectores

Los vectores son una estructura de datos de tamaño fijo que pueden almacenar valores de forma limitada. **El lenguaje permitirá únicamente el uso de vectores 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 a la posición 0.

#### 5.18.1 Declaración de Vectores

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.

```
<MUTABILIDAD> <ID> : <TIPO> [ ] = [ <LISTAVALORES> ] ;  
<MUTABILIDAD> <ID> : <TIPO> [ ] [ ] = [ <LISTAVALORES2> ] ;  
  
let vector1 : string [ ] = [ "Hola", "Mundo" ];  
const vector2 : int [ ] [ ] = [ [ 1, 2 ], [ 3, 4 ] ] ;
```

#### 5.18.2 Acceso a vectores

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

```
<VECTOR> [ <EXPRESION> ]  
  
let vector3:string[ ] = [ "Hola", "Mundo" ];  
let valor3 : string = vector3[0]; // Almacena el valor "hola"  
  
const vector4:int [ ] [ ] = [ [ 1, 2 ], [ 3, 4 ] ];  
const valor4:int = vector4[0][0]; // Almacena el valor 1
```

### 5.18.3 Asignación de Vectores

Para modificar el valor de una posición de un vector, se debe colocar el nombre del vector seguido del acceso y la expresión a asignar.

#### Observaciones:

- A una posición de un vector se le puede asignar el valor de otra posición de otro vector o del mismo vector.
- Si el vector fue declarado como const no se puede realizar ninguna modificación al valor que posee. Se considera como error semántico.

```
<VECTOR> [ <EXPRESION> ] = <EXPRESION> ;  
<VECTOR> [ <EXPRESION> ] [ <EXPRESION> ] = <EXPRESION> ;  
  
let vector3:string[] = ["Hola", "Mundo"];  
vector3[0] = "OLC1";  
vector3[1] = "2do Semestre";
```

## 5.19 Listas Dinámicas

Las listas dinámicas son una estructura de datos de tamaño variable que pueden almacenar valores de forma ilimitada.

### 5.19.1 Declaración de Listas

```
let <ID> : List< <TIPO> > ;  
  
// Ejemplo  
let miLista : List<int>;
```

#### Observaciones:

- Las listas dinámicas siempre son variables, por lo tanto no se pueden declarar usando la palabra const. Esto sería un error de tipo semántico.

### 5.19.2 Append

Para agregar valores a la lista, se utiliza la palabra reservada **append**, la cual agrega el elemento al final de la lista.

```
<LISTA>.append( <EXPRESION>);
```

```
let miLista : List<int>;  
miLista.push(1);
```

### 5.19.3 Acceso a listas

Para acceder al valor de una expresión de una lista, se utilizará la palabra reservada **get**.

```
<LISTA>.get( <EXPRESION> )
```

```
let miLista : List<int>;  
miLista.append(1);  
const valor:int = miLista.get(0); // Almacena el valor 1
```

### 5.19.4 Asignación de Listas

Para modificar el valor de una posición de una lista, se utilizará la función **set**. Esta función recibe dos argumentos: el primero indica la posición que se desea modificar, y el segundo especifica el nuevo valor que se asignará.

```
<LISTA>.set( <EXPRESION>, <EXPRESION> ) ;
```

```
let miLista : List<int>;  
miLista.append(1);  
miLista.set(0,10);
```

### 5.19.5 Remove

Para eliminar valores a la lista, se utiliza la palabra reservada **remove**, la cual recibe la posición del elemento a eliminar de la lista y lo retorna.

```
<LISTA>.remove( <EXPRESION>)
```

```
let miLista : List<int>;  
miLista.append(1);  
miLista.append(2);  
let miVar:int = miLista.remove(0); //almacena el valor de 1
```



### 5.19.6 Pop

Funciona similar al remove, pero este elimina el último elemento de la lista.

```
<LISTA>.pop() ;

let miLista : List<int>;
miLista.append(1);
miLista.append(2);
let miVar:int = miLista.pop(); //almacena el valor de 2
```

### 5.19.7 Reverse

Este método invierte el orden de los elementos del vector.

```
<LISTA>.reverse() ;

let miLista : List<int>;
miLista.append(1);
miLista.append(2);
miLista.reverse(); //pasa de [1,2] a [2,1]
```

## 5.20 Structs

Los structs son tipos de datos que permiten agrupar diferentes variables bajo un único nombre.

### 5.20.1 Declaración de Structs

Se definen mediante la palabra reservada "struct", seguida de un bloque de campos, cada uno con su identificador y tipo correspondiente.

```
struct <ID>{
    <LISTA_STRUCT>
};

// EJEMPLO
struct estudiante{
    nombre:string;
    edad:int;
};
```

### 5.20.2 Instanciación de Struct

La instanciación de un struct se refiere al proceso de crear una variable utilizando la definición de la estructura.

```
<MUTABILIDAD>    <ID>    :    <NOMBRE_STRUCT>    =    {  
<VALORES_STRUCT> };
```

```
struct estudiante{  
    nombre:string;  
    edad:int;  
};
```

```
let e1:estudiante = { nombre: "prueba", edad: 10 };  
const e2: estudiante = { edad: 20, nombre: "prueba2 "};  
// el orden no afecta ya que se asocia con el id y no con una posición
```

### 5.20.3 Acceso de struct

El acceso a un struct se refiere a la capacidad de acceder a los campos individuales dentro de una estructura utilizando su nombre seguido de un operador de acceso, como el punto ".".

```
<STRUCT> .<CAMPO> ;
```

```
struct estudiante{  
    nombre:string;  
    edad:int;  
};
```

```
let e1:estudiante = { nombre: "prueba", edad: 10 };  
  
const edadEstudiante:int = e1.edad; // debe ser 10
```

### 5.20.4 Asignación de struct

Para poder modificar un struct, primero se debe tener una instancia de este. Luego, se debe acceder a cada uno de los campos del struct utilizando el operador de acceso punto '.' y asignarles nuevos valores según sea necesario. Si la instancia fue declarada como const no se puede cambiar y se considera un error semántico.

**<STRUCT> . <CAMPO> = <EXPRESION> ;**

```
struct estudiante{  
    nombre:string;  
    edad:int;  
};
```

```
let e1: estudiante = {nombre: "prueba", edad: 10};  
e1.edad = 28;
```

## 5.21 Funciones

Una función es una subrutina de código que se identifica con un tipo, nombre 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.

```
<TIPO> <ID> ( <PARAMETROS> ) {  
    <INSTRUCCIONES>  
}
```

```
PARAMETROS -> PARAMETROS , <PARAMETRO>  
    | <PARAMETRO>
```

**PARAMETRO-> <ID> : <TIPO> = <EXPRESION>  
| <ID> : <TIPO>**

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

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.

## 5.22 Métodos

Un método también es una subrutina de código que se identifica 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 {}.

Para los métodos no es obligatorio tener un valor de retorno, pero de haberlo debe de ser de tipo void, en caso contrario debe lanzarse un error de tipo semántico.

```

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.

### 5.23 Llamadas

La llamada a una función devuelve un resultado, mientras que las llamadas a un método no devuelven nada. Los parámetros normalmente se asocian por posición, pero en este lenguaje se asocian por identificador.

```

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 definición de la función en la llamada puede omitirse o bien sobrescribir su valor.
- Al momento de ejecutar cualquier llamada, no se diferenciarán entre métodos y funciones.
- Se podrán llamar métodos y funciones antes de declararse, por ello se recomienda realizar 3 pasadas al AST: la primera para almacenar funciones, métodos y structs, la segunda para las declaraciones y asignaciones y la tercera para la función main

## 5.24 Impresión en consola

Esta función nos permite imprimir expresiones y agrega un salto de línea al final del contenido

```
console.log ( <EXPRESION> ) ;

let arreglo : int [] = [1, 2, 3];

struct estudiante{
    nombre:string;
    edad:int;
};

let e1: estudiante = {nombre: "prueba", edad: 10};

console.log("Hola mundo");
console.log(arreglo);
console.log(arreglo[0]);
console.log(e1.edad);

//Salida esperada
/*
Hola mundo
[1, 2, 3]
1
10
*/
```

## 5.25 Función 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> )
```

### 5.26 Length

Esta función recibe como parámetro un vector, una lista 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.

```
length ( <EXPRESION> )
```

### 5.27 ToString

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

#### Observación:

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

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

// Ejemplo

```
struct estudiante{  
    nombre:string;  
    edad:int;  
};
```

```
let e1: estudiante = {nombre: "prueba", edad: 10};  
console.log(toString(e1));
```

```
const cad:string = toString(1); // almacena "1" como cadena  
const cad2:string = toString(true); // almacena "true" como cadena
```

//Salida

// estudiante { nombre: "prueba", edad: 10 }

## 5.29 Función RUN\_MAIN

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

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

## 6. Reportes

Los reportes son parte fundamental, 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 tokens

El reporte de tokens debe contener la información suficiente para identificar los tokens reconocidos por el analizador léxico.

#	Lexema	Tipo	Línea	Columna
1	let	let	1	1
2	miVar	id	1	10

### 6.2 Tabla de errores

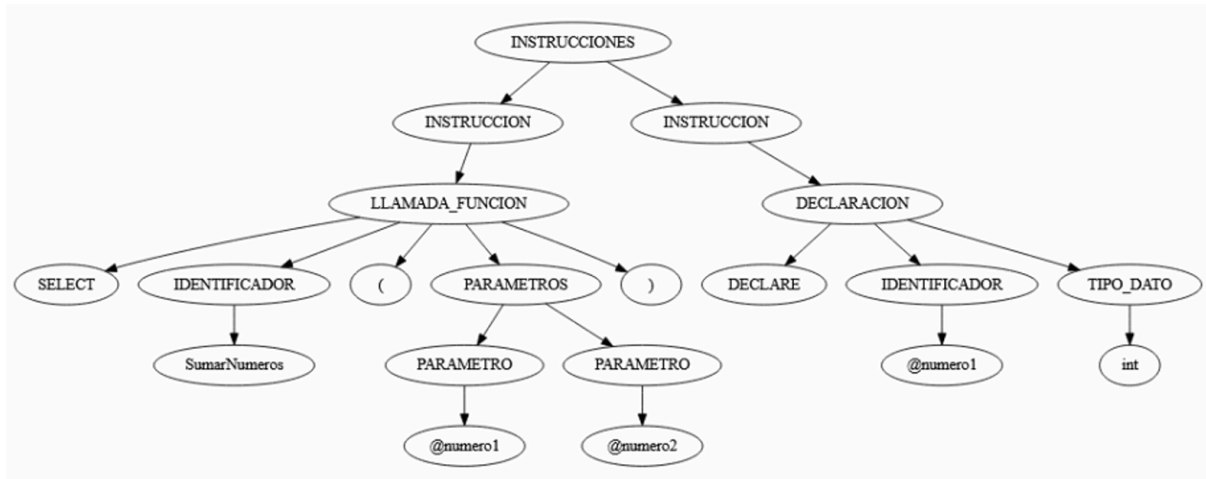
El reporte de errores debe contener la información suficiente para detectar y corregir errores en el código fuente.

#	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 resta entre CADENA y CADENA	8	10



### 6.3 AST

Este reporte muestra el AST 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.



### 6.4 Tabla de Símbolos

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

#	Id	Tipo	Tipo	Entorno	Valor	Línea	Columna
1	var	Variable	Entero	Funcion1	1	15	20
2	var2	Variable	Bool	Funcion2	true	20	13

### 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 "console.log", 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. Restricciones

- La aplicación deberá de desarrollarse utilizando el lenguaje **JAVA**.
- Las herramientas para realizar los analizadores serán **JFLEX y CUP**.
- **El proyecto debe ser realizado de forma individual.**
- **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 será sobre el archivo ejecutable (.jar) de su aplicación, de cumplir con este requerimiento **NO SE CALIFICARÁ**.
- Para tener derecho a calificación se debe tener interfaz gráfica (**NO PROGRAMAS SOLAMENTE EN CONSOLA**).
- El nombre del repositorio de Gitlab debe ser **OLC1\_VD24\_#Carnet**
- Se debe agregar al auxiliar como **developer** del repositorio.
- El archivo de gramática debe contener su gramática libre del contexto y debe de ser limpio, entendible y **no debe ser una copia del archivo de CUP**. La gramática debe estar escrita en formato **BNF**
- La entrega debe ser realizada mediante UEDI
- Debe haber evidencia en el repositorio de su trabajo, de lo contrario no se calificará.
- Reportes o salidas en consola quemadas tendrán nota de 0 y se penalizará.

## 8. Entrega

El proyecto estará dividido en dos fases, a continuación se define como se van a realizar las entregas

- FASE 1
  - Funcionalidades a entregar
    - Interfaz gráfica
    - Comentarios
    - Tipos de datos
    - Secuencias de escape
    - Expresiones (Aritméticas, lógicas, relacionales y casteos)
    - Signos de agrupación
    - Declaración de variables
    - Asignación de variables
    - Acceso de variables
    - Incremento y decremento
    - Sentencias de control
    - CASTEOS

- Sentencias cíclicas
  - Sentencias de transferencia (menos el return)
  - Impresión en consola
  - Reporte Tabla de Símbolos
  - Reporte de Errores
  - Reporte de Tokens
  - Entregables
    - Código fuente de la aplicación
    - Archivo ejecutable de la aplicación
    - Archivo de Gramática
  - Fecha de entrega
    - Domingo 15 de diciembre hasta las 11:59 hrs
- FASE 2
  - Funcionalidades a entregar
    - Vectores
    - Listas
    - Structs
    - Funciones
    - Métodos
    - Llamadas
    - Función RUN\_MAIN
    - Todos los reportes
    - Resto de funciones nativas
  - Entregables
    - Código fuente de la aplicación
    - Archivo ejecutable de la aplicación
    - Archivo de gramática
  - Fecha de entrega
    - Domingo 29 de diciembre hasta las 11:59 hrs