Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Primer semestre 2024
Catedráticos: Ing. Bayron López, Ing. Edgar Saban e Ing. Luis Espino
Tutores Académicos: Henry Mendoza, Daniel Santos y Andres Rodas

OLCScript

1. Competencias

1.1. Competencia General

Que el estudiante realice la fase de análisis de un compilador para el lenguaje de programación **OLCScript**, enfocada en comprender la estructura del código fuente utilizando herramientas para la generación de analizadores léxicos y sintácticos.

1.2. Competencias Específicas

- Que el estudiante utilice una herramienta léxica y una sintáctica para el reconocimiento y análisis del lenguaje
- Que el estudiante implemente la traducción orientada por la sintaxis utilizando reglas semánticas haciendo uso de atributos sintetizados y/o heredados.
- Que el estudiante se familiarice con las herramientas y estructuras de datos disponibles para la creación de un Intérprete.

2. Descripción

OLCScript es un lenguaje de programación basado en el popular Typescript, conocido por su versatilidad al ser un lenguaje multiparadigma que ha ganado considerable popularidad gracias a su sintaxis moderna y diversas características distintivas. Este lenguaje no sólo hereda las ventajas de **Typescript**, sino que también incorpora funcionalidades avanzadas, como programación funcional, tipado estático, inferencia de tipos, entre otras. Estas características lo convierten en una herramienta moderna y eficiente, haciéndolo idóneo para su estudio y comprensión, especialmente en entornos de laboratorio.

2.1. Componentes de la aplicación

Se necesita desarrollar un entorno de desarrollo para la creación de aplicaciones en el lenguaje **OLCScript**. Este entorno, también conocido como IDE (Integrated Development Environment), desempeñará la función de analizar el código fuente escrito en **OLCScript**.

Para llevar a cabo este proyecto, se propone utilizar el lenguaje de programación **Python**. En particular, se emplea la herramienta **PLY** para la implementación del analizador léxico y sintáctico. La gramática correspondiente se diseñará de manera precisa y eficiente con la finalidad de garantizar la correcta interpretación del código **OLCScript** y su posterior procesamiento.

La aplicación contará con los siguientes componentes:

2.1.1. Editor

El editor será parte del entorno de trabajo, cuya finalidad será proporcionar ciertas funcionalidades, características y herramientas que serán de utilidad al 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 la interfaz de usuario (GUI). Queda a discreción del estudiante el diseño.

2.1.2. Funcionalidades

Crear archivos: El editor deberá ser capaz de crear archivos en blanco.
Abrir archivos: El editor deberá abrir archivos **.olc**
Guardar archivo: El editor deberá guardar el estado del archivo en el que se estará trabajando con extensión **.olc**

2.1.3. Herramientas

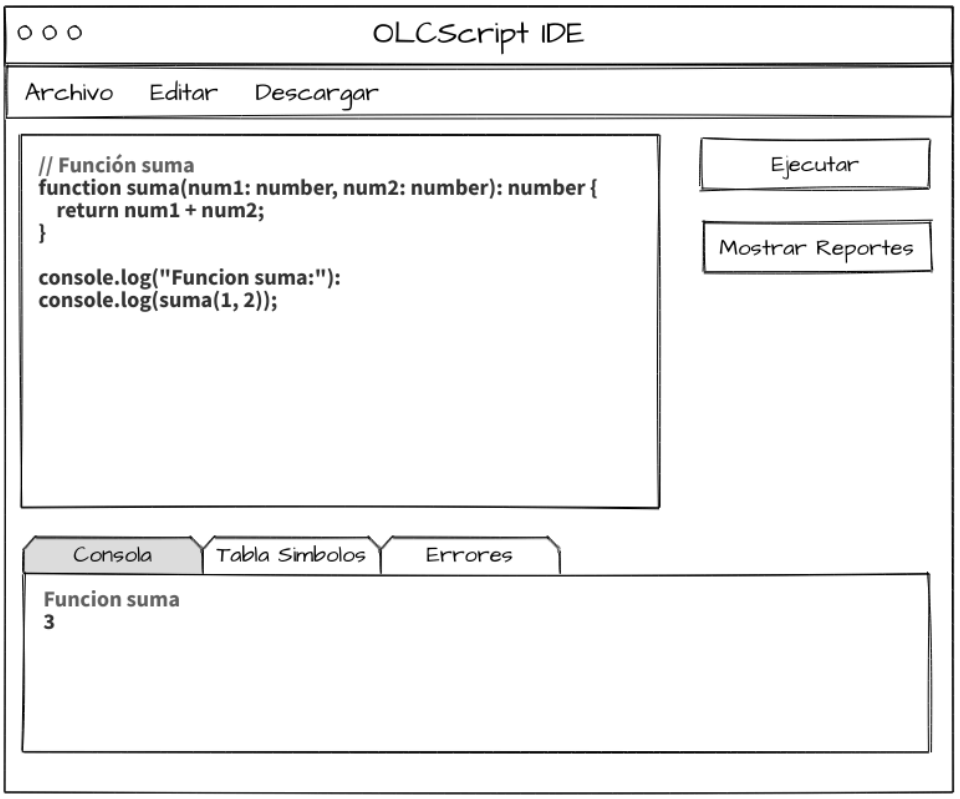
Ejecutar: hará el llamado al intérprete, el cual se hará cargo de realizar los análisis léxico, sintáctico y semántico, además de interpretar todas las sentencias.

2.1.4. Reportes

Reporte de Errores: Se mostrarán todos los errores encontrados al realizar el análisis léxico, sintáctico y semántico.
Reporte de Tabla de Símbolos: Se mostrarán todas las variables, métodos y funciones que han sido declarados dentro del flujo del programa, así como el entorno en el que fueron declarados.

2.1.5. Consola

La consola es un área especial dentro del IDE que permite visualizar las notificaciones, errores, advertencias e impresiones que se produjeron durante el proceso de análisis de un archivo de entrada.

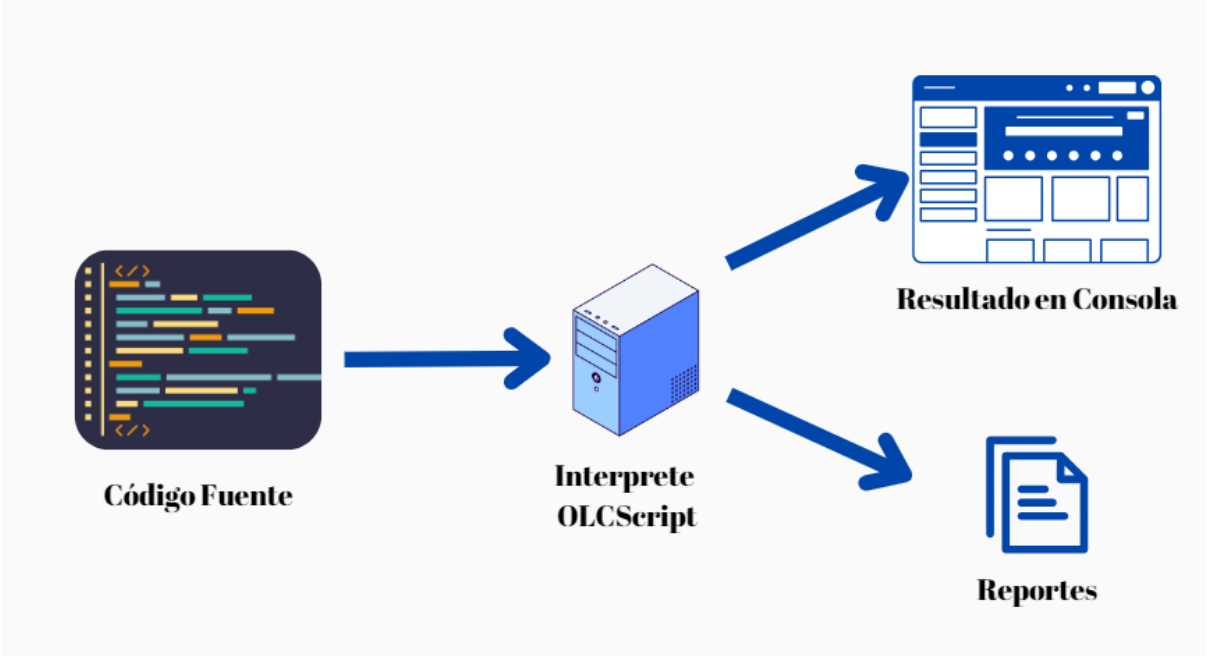


(Interfaz propuesta)

2.2. Flujo del proyecto

- **Inicio del Programa:**
  - El programa comienza con la carga o creación de un archivo que contiene el código fuente en OLCScript.
- **Cargar/Crear Archivo:**
  - El usuario tiene la opción de cargar un archivo existente o crear uno nuevo que contenga el código en OLCScript.
- **Interpretación del Código:**
  - Se utiliza el analizador léxico y sintáctico implementado con PLY en Python para interpretar el código OLCScript.
  - Durante este proceso, se generan reportes de errores si se encuentran violaciones a la gramática del lenguaje.
- **Generación de Tabla de Símbolos:**
  - Se construye una tabla de símbolos que contiene información sobre las variables, funciones u otros elementos definidos en el código.
  - La tabla de símbolos facilita el seguimiento y control de los elementos presentes en el código.
- **Interpretación de Instrucciones:**
  - Se realiza la interpretación de las instrucciones del código OLCScript.
  - Cada instrucción es procesada según las reglas y semántica del lenguaje.
- **Generación de Resultados:**
  - Se generan resultados y reportes que pueden incluir:

    - Información detallada sobre errores encontrados durante el análisis.
    - Contenido de la tabla de símbolos.
    - Resultados de la interpretación de las instrucciones mediante la consola.
- **Fin del Programa:**
    - El programa finaliza, y el usuario puede revisar los resultados generados durante el proceso.



Flujo de ejecución del proyecto

Componentes clave:

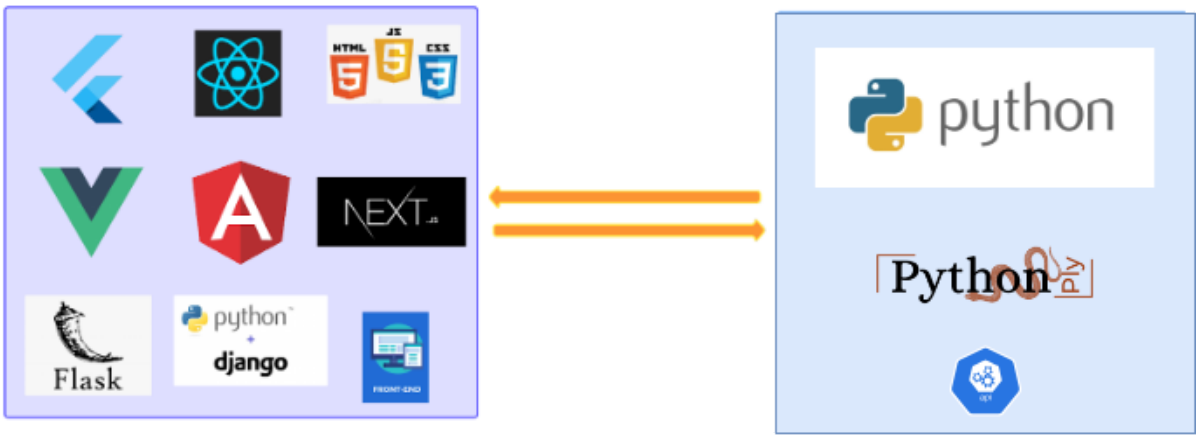
- **Código fuente**: La entrada es código fuente en formato .olc (Más adelante se especifican los detalles del lenguaje).
- **Intérprete**: El código fuente se procesa por el intérprete OLCScript, desarrollado por el estudiante. Como salida se obtienen los resultados en consola y los reportes.
- **Resultado de consola**: Se deberá implementar una consola (Como se expone en interfaz propuesta) donde se pueda visualizar las salidas, resultado del intérprete OLCScript.
- **Reportes**: Se deberán generar reportes de Errores y Tabla de Símbolos. Estos se detallarán más adelante.

2.3. Arquitecturas propuestas

Para la implementación de una solución se sugieren las siguientes arquitecturas, el estudiante tiene la libertad de escoger la herramienta y/o librería a su elección para el desarrollo de la GUI del IDE

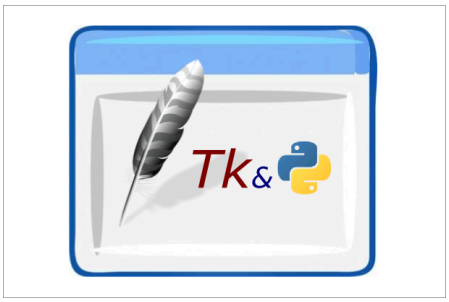
2.3.1. Stack: Frontend / Api - REST

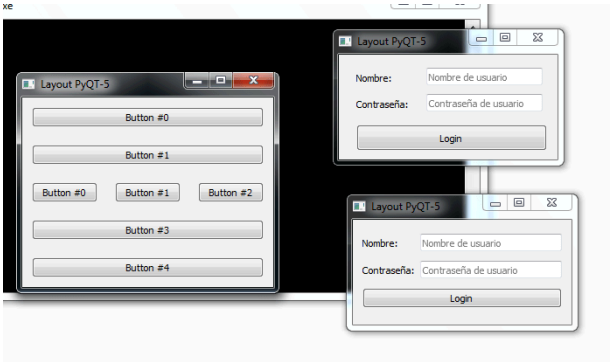
Es permitido crear una Api-REST en Python y luego consumirla con un frontend, la tecnología para desarrollar el frontend queda a discreción del estudiante. Dicha aplicación puede ser implementada de forma local y el framework/librería en Python para construir la API queda a discreción del estudiante.



2.3.2. Aplicación de Escritorio

Se puede implementar una aplicación de escritorio con Python, el framework para desarrollar la GUI queda a discreción del estudiante, sin embargo tomar en cuenta que la implementación de una GUI para la aplicación es **obligatoria** no se calificarán aplicaciones en consola ni APIS consumidas de forma manual.





3. Generalidades del lenguaje OLCScript

El lenguaje OLCScript está inspirado en la sIntaxis del lenguaje TypeScript, por lo tanto se conforma por un subconjunto de instrucciones de este, con la diferencia de que OLCScript tendrá una sintaxis más reducida pero sin perder las funcionalidades que caracterizan al lenguaje original.

3.1. Expresiones en el lenguaje OLCScript

Cuando se haga referencia a una 'expresión' a lo largo de este enunciado, se hará referencia a cualquier sentencia u operación que devuelve un valor. Por ejemplo:

- Una operación aritmética, comparativa o lógica
- Acceso a un variable
- Acceso a un elemento de una estructura
- Una llamada a una función

3.2. Ejecución:

OLCScript por su naturaleza carece de una función **main**, por ello para el inicio de ejecución del programa, el Intérprete deberá de ejecutar las órdenes conforme estas sean declaradas en el archivo de entrada. Dicho comportamiento se detalla a lo largo de este enunciado.

3.3. Identificadores

Un identificador será utilizado para dar un nombre a variables y métodos. Un identificador está compuesto básicamente por una combinación de letras, dígitos, o guión bajo.

Ejemplos de identificadores válidos:

```
IdValido
id_Valido
i1d_valido5
_value
```

Ejemplo de identificadores no válidos

```
&idNoValido
.5ID
true
Tot@l
1d
```

Consideraciones:

- El identificador puede iniciar con una letra o un guión bajo _
- Por simplicidad el identificador no puede contener carácteres especiales (.$,-, etc)
- El identificador no puede comenzar con un número.

3.4. Case Sensitive

El lenguaje OLCScript es case sensitive, esto quiere decir que diferenciará entre mayúsculas con minúsculas, por ejemplo, el identificador variable hace referencia a una variable específica y el identificador Variable hace referencia a otra variable. Las palabras reservadas también son case sensitive por ejemplo la palabra **if** no será la misma que **IF.**

3.5. Comentarios

Un comentario es un componente léxico del lenguaje que no es tomado en cuenta para el análisis sintáctico de la entrada. Existirán dos tipos de comentarios:

- Los comentarios de una línea que serán delimitados al inicio con el símbolo de *//* y al final como un carácter de finalización de línea.
- Los comentarios con múltiples líneas que empezarán con los símbolos */** y terminarán con los símbolos ***/**

```
// Esto es un comentario de una línea
/*
Esto es un comentario multilínea
*/
```

3.6. Tipos estáticos

El lenguaje OLCScript no soportará múltiples asignaciones de diferentes tipos para una variable, por lo tanto si una variable es asignada con un tipo, solo será posible asignar valores de ese tipo a lo largo de la ejecución, sí alguna variable se le asignase un valor que no corresponde a su tipo declarado, el programa debe mostrar un mensaje detallando el error.

3.7. Tipos de datos primitivos

Se utilizan los siguientes tipos de datos primitivos:

Tipo primitivo	Definición	Rango (teorico)	Valor por defecto
`number`	**Acepta valores números enteros**	**-2,147,483,648 a +2,147,483,647**	`0`
`float`	**Acepta valores numéricos de punto flotante**	**1.2E-38 a 3.4E+38 (6 dígitos de precisión)**	`0.0`

Tipo primitivo	Definición	Rango (teorico)	Valor por defecto
`string`	**Acepta cadenas de caracteres**	**[0, 65535] caracteres (acotado por conveniencia)**	**“”**
boolean	**Acepta valores lógicos de verdadero y falso**	**true false**	`true`
`char`	**Acepta un solo caracter ASCII**	**[0, 65535] caracteres**	**‘’**

Consideraciones:

- Por conveniencia y facilidad de desarrollo, el tipo String será tomado como un tipo primitivo.
- Cuando se haga referencia a *tipos numéricos* se estarán considerando los tipos **`Number`** y **`Float`**
- Cualquier otro tipo de dato que no sea primitivo tomará el valor por defecto de **null** al no asignarle un valor en su declaración.
- Cuando se declare una **variable** y no se defina su valor, automáticamente tomará el valor por defecto del tipo, **`null`** esto para evitar la lectura de basura durante la ejecución.
- El literal `0` se considera tanto de tipo **`Number`** como **`Float`**.

3.8. Tipos Compuestos

Cuando hablamos de tipos compuestos nos vamos a referir a ellos como no primitivos, en estos tipos vamos a encontrar las estructuras básicas del lenguaje OLCScript.

- Array
- Interface

Estos tipos especiales se explicarán más adelante.

3.9. Valor nulo (**`null`**)

En el lenguaje OLCScript se utiliza la palabra reservada **`null`** para hacer referencia a la nada, esto indicará la ausencia de valor, por lo tanto cualquier operación sobre **`null`** será considerada un **error y dará `null`** como resultado.

3.10. 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	Definición
\"	**Comilla Doble**
****	**Barra invertida**
\n	**Salto de línea**
\r	**Retorno de carro**
\t	**Tabulación**

4. Sintaxis del lenguaje OLCScript

A continuación, se define la sIntaxis para las sentencias del lenguaje OLCScript

4.1. Bloques de Sentencias

Será un conjunto de sentencias delimitado por llaves “{ }”, cuando se haga referencia a esto querrá decir que se está definiendo un ámbito local con todo y sus posibles instrucciones, además tendrá acceso a las variables del ámbito global.
Las variables declaradas en dicho ámbito únicamente podrán ser utilizadas en este ámbito o en posibles ámbitos anidados. Cada instrucción del lenguaje OLCScript deberá terminar con el delimitador **punto y coma** (;).

```
// <sentencias de control>
{
// sentencias
}

var i: number = 10 // variable global es accesible desde este ámbito
if( i == 10 )
{
    var j:number = 10 + i; // i es accesible desde este ámbito
    if ( i == 10 )
    {
        var k: number = j + 1;// i y j son accesibles desde este ámbito
    }
    j = k; // error k ya no es accesible en este ámbito
}
```

Consideraciones:

- Estos bloques estarán asociados a alguna sentencia de control de flujo por lo tanto no podrán ser declarados de forma independiente.

4.2. Signos de agrupación

Para dar orden y jerarquía a ciertas operaciones aritméticas se utilizarán los signos de agrupación. Para los signos de agrupación se utilizarán los paréntesis ( )

```
3 - (1 + 3) * 32 / 90 // 1.5
```

4.3. Variables

Una variable es un elemento de datos cuyo valor puede cambiar durante el curso de la ejecución de un programa **siempre y cuando sea el mismo tipo de dato**. Una variable cuenta con un nombre y un valor, los nombres de variables no pueden coincidir con una palabra reservada.

Para poder utilizar una variable se tiene que definir previamente, la declaración nos permite crear una variable y asignarle un valor o sin valor.

Además la definición de tipo durante la declaración puede ser implícita o explícita, es explícita cuando se indica el tipo del cual será la variable e implícita cuando esta toma el tipo del valor al cual se está asignando.

SIntaxis:

```
// declaración con tipo y valor
var <identificador> : <Tipo> = <Expresión> ;
// declaración con valor
var <identificador> = <Expresión> ;
// declaración con tipo y sin valor
var <identificador> : <Tipo> ;
```

Consideraciones:

- Las variables solo pueden tener **un tipo de dato** definido y este no podrá cambiar a lo largo de la ejecución.
- Solo se puede declarar **una** variable por sentencia.

- Si la variable ya existe se debe actualizar su valor por el nuevo, validando que el nuevo valor sea del mismo tipo del de la variable.
- El nombre de la variable **no puede** ser una **palabra reservada** ó del de una variable **previamente** definida.
- El lenguaje al ser case sensitive distinguirá a las variables declaradas como por ejemplo `id` y `Id` se toman como dos variables diferentes.
- Si la expresión tiene un tipo de dato **diferente** al definido previamente se tomará como **error** y la variable obtendrá el valor de **`null`** para fines prácticos.
- Cuando se asigna un valor de tipo ***`Number`*** a una variable de tipo ***`Float`*** el valor será considerado como ***`Float`***, esta es la única conversión implícita que habrá.

Ejemplo:

```
// declaración de variables

var valor: string//correcto, declaración sin valor

// correcto, declaración de una variable tipo Int con valor
var valor1 = 10:

var valor1:number = 10.01; // Error: no se puede asignar un float a un number
var valor2:float = 10.2; // correcto

var valor2_1:float = 10 + 1; // correcto será 11.0

var valor3 = "esto es una variable";  //correcto variable tipo String

var char:char = "A";  //correcto variable tipo Character

var valor4:bool = true;   //correcto

//debe ser un error ya que los tipos no son compatibles
var valor4:string = true;

// debe ser un error ya que existe otra variable valor3 definida previamente
var valor3:number = 10;

var .58 = 4; // debe ser error porque .58 no es un nombre válido

var if = "10"; // debe ser un error porque "if" es una palabra reservada

// ejemplo de asignaciones
```

```
valor1 = 200; // correcto

valor3 = "otra cadena"; //correcto

valor4 = 10;  //error tipos incompatibles (Bool, Int)

valor2 = 200; // correcto conversión implícita (Float, Int)

char = "otra cadena"; //error tipos incompatibles (Character, String)
```

4.4. Constantes

Las constantes en el lenguaje OLCScript se definen anteponiendo la palabra reservada **const**, esto indica que el valor de dicha constante no podrá cambiar a lo largo de la ejecución, por **lo tanto no será válida cualquier asignación** que se haga sobre una constante y deberá ser notificada como un **error.**

Consideraciones:

- Solo será posible crear constantes con tipos de datos primitivos
- Una constante no puede ser definida sin un valor específico.
- El nombre de las constantes tiene las mismas propiedades que las variables.
- Las constantes cuentan con declaración con tipo de forma implícita y explícita como las variables.

Ejemplo

```
// declaración de constantes

//Incorrecto, la constante debe tener un valor asignado
const valor: string;

// correcto, declaración de una constante tipo Int con valor
const valor1 = 10;

const valor1:number = 10.01; // Error: no se puede asignar un float a un
nomber

const valor2:float = 10.2; // correcto

const valor3 = "esto es una variable";  //correcto constante tipo String

const valor4:boolean = true;   //correcto
```

```
const .58 = 4; // debe ser error porque .58 no es un nombre válido

const if = "10"; // debe ser un error porque "if" es una palabra reservada

// error valor1 al ser una constante no puede cambiar su valor
valor1 = 200;
```

4.5. Operadores Aritméticos

Los operadores aritméticos toman valores numéricos de expresiones y retornan un valor numé rico **único** de un determinado **tipo**. Los operadores aritméticos estándar son adición o suma +, sustracción o resta -, multiplicación *, y división /, adicionalmente vamos a trabajar el módulo %.

4.5.1. Suma

La operación suma se produce mediante la suma de tipos numéricos o Strings concatenados, debido a que OLCScript está pensado para ofrecer una mayor versatilidad ofrecerá conversión de tipos de forma implícita como especifica la siguiente tabla:

Operandos	Tipo resultante	Ejemplo
`number + number` `number + float`	`number` `float`	`1 + 1 = 2` `1 + 1.0 = 2.0`
`float + float` `float + number`	`float` `float`	`1.0 + 13.0 = 14.0` `1.0 + 1 = 2.0`
`string + string`	`string`	`"ho" + "la" = "hola"`

Consideraciones:

- Cualquier otra combinación será inválida y se deberá reportar el error.

4.5.2. Resta

La resta se produce cuando existe una sustracción entre tipos numéricos, de igual manera que con otras operaciones habrá conversión de tipos implícita en algunos casos

Operandos	Tipo resultante	Ejemplo
`number - number` `number - float`	`number` `float`	`1 - 1 = 0` `1 - 1.0 = 0.0`
`float - float` `float - number`	`float` `float`	`1.0 - 13.0 = -12.0` `1.0 - 1 = 0.0`

4.5.3. Multiplicación

La multiplicación se produce cuando existe un producto entre tipos numéricos, de igual manera que con otras operaciones habrá conversión de tipos implícita en algunos casos.

Operandos	Tipo resultante	Ejemplo
`number * number` `number * float`	`number` `float`	`1 * 10 = 10` `1 * 1.0 = 1.0`
`float * float` `float * number`	`float` `float`	`1.0 * 13.0 = 13.0` `1.0 * 1 = 1.0`

Consideraciones:

- Cualquier otra combinación será inválida y se deberá reportar el error.

4.5.4. División

La división produce el cociente entre tipos numéricos, de igual manera que con otras operaciones habrá conversión de tipos implícita en algunos casos a su vez truncamiento cuando sea necesario.

Operandos	Tipo resultante	Ejemplo
`number / number` `number / float`	`number` `float`	`10 / 3 = 3` `1 / 3.0 = 0.3333`
`float / float` `float / number`	`float` `float`	`13.0 / 13.0 = 1.0` `1.0 / 1 = 1.0`

Consideraciones:

- Cualquier otra combinación será inválida y se deberá reportar el error.
- Se debe verificar que no haya división por 0, de lo contrario se debe mostrar una advertencia y por conveniencia el resultado será un valor **`null`**.

4.5.5. Módulo

El módulo produce el residuo entre la división entre tipos numéricos de tipo **`number`**.

Operandos	Tipo resultante	Ejemplo
`number % number`	`number`	`10 % 3 = 1`

Consideraciones:

- Cualquier otra combinación será inválida y se deberá reportar el error.
- Se debe verificar que no haya división por 0, de lo contrario se debe mostrar una advertencia y por conveniencia el resultado será un valor **`null`**.

4.5.6. Operador de asignación

4.5.6.1. Suma

El operador += indica el incremento del valor de una **expresión** en una **variable** de tipo ya sea **`number`** o de tipo **`Float`** . El operador += será como una suma implícita de la forma: `variable = variable + expresión` Por lo tanto tendrá las validaciones y restricciones de una suma.

Ejemplos:

```
var var1:number = 10;
var var2:float = 0.0;

var1 += 10; //var1 tendrá el valor de 20

var1 += 10.0; // error, no puede asignar un valor de tipo Float a un
number

var2 += 10: // var2 tendrá el valor de 10.0

var2 += 10.0; //var tendrá el valor de 20.0

var str:string = "cad";

str += "cad"; //str tendrá el valor de "cadcad"

str += 10; //operación inválida String + number
```

4.5.6.2. Resta

El operador -= indica el decremento del valor de una **expresión** en una **variable** de tipo ya sea **`number`** o de tipo **`Float`** . El operador -= será como una resta implícita de la forma: `variable = variable - expresión` Por lo tanto tendrá las validaciones y restricciones de una resta.

Ejemplos:

```
var var1:number = 10;
var var2:float = 0.0;

var1 -= 10; //var1 tendrá el valor de 0

var1 -= 10.0; // error, no puede asignar un valor de tipo Float a un
number
```

```
var2 -= 10; // var2 tendrá el valor de -10.0

var2 -= 10.0; //var tendrá el valor de -20.0
```

4.5.7. Negación unaria

El operador de negación unaria precede su operando y lo niega (*-1) esta negación se aplica a tipos numéricos

Operandos	Tipo resultante	Ejemplo
`-number`	`number`	`-(-(10)) = 10`
`-Float`	`Float`	`-(1.0) = -1.0`

4.6. Operaciones de comparación

Compara sus operandos y devuelve un valor lógico en función de si la comparación es verdadera (**`true`**) o falsa (**`false`**). Los operandos pueden ser numéricos, Strings o lógicos, *permitiendo únicamente la comparación de expresiones del mismo tipo*.

4.6.1. Igualdad y desigualdad

- **El operador de igualdad** (==) devuelve **`true`** si ambos operandos tienen el mismo valor, en caso contrario, devuelve **`false`**.
- **El operador no igual a** (!=) devuelve **`true`** si los operandos no tienen el mismo valor, de lo contrario, devuelve **`false`**.

Operandos	Tipo resultante	Ejemplo
`number [==,!=] number`	`boolean`	`1 == 1 = true` `1 != 1 = false`
`float [==,!=] float`	`boolean`	`13.0 == 13.0 = true` `0.001 != 0.001 = false`
`boolean [==,!=] boolean`	`boolean`	`true == false = false` `false != true = true`
`string [==,!=] string`	`boolean`	`"ho" == "Ha" = false` `"Ho" != "Ho" = false`
`char [==,!=] char`	`boolean`	`"h" == "a" = false` `"H" != "H" = false`

Consideraciones

- Cualquier otra combinación será inválida y se deberá reportar el error.
- Las comparaciones entre cadenas se hacen lexicográficamente (carácter por carácter).

4.6.2. Relacionales

Las operaciones relacionales que soporta el lenguaje OLCScript son las siguientes:

- **Mayor que**(>) Devuelve **`true`** si el operando de la izquierda es mayor que el operando de la derecha.
- **Mayor o igual que:**(>=)Devuelve true si el operando de la izquierda es mayor o igual que el operando de la derecha.
- **Menor que**(<) Devuelve true si el operando de la izquierda es menor que el operando de la derecha.
- **Menor o igual que**(<=) Devuelve true si el operando de la izquierda es menor o igual que el operando de la derecha.

Operandos	**Tipo resultante**	**Ejemplo**
`number[>,<,>=,<=] number`	**`boolean`**	`1 < 1 = false`
`float [>,<,>=,<=] float`	**`boolean`**	`13.0 >= 13.0 = true`
`string[>,<,>=,<=] string`	**`boolean`**	`"aAA" <= "bA" = true`
`char [>,<,>=,<=] char`	**`boolean`**	`‘a’ <= ‘b’ = true`

Consideraciones

- Cualquier otra combinación será inválida y se deberá reportar el error.
- La limitación de las operaciones también se aplica a comparación de literales.

4.7. Operadores Lógicos

Los operadores lógicos comprueban la veracidad de alguna condición. Al igual que los operadores de comparación, devuelven el tipo de dato **`boolean`** con el valor **`true`** ó **`false`**.

- **Operador and** (&&) devuelve **`true`** si ambas expresiones de tipo **`boolean`** son **`true`**, en caso contrario devuelve **`false`**.
- **Operador or** (||) devuelve **`true`** si alguna de las expresiones de tipo **`boolean`** es **`true`**, en caso contrario devuelve **`false`**.
- **Operador not** (!) Invierte el valor de cualquier expresión Booleana.

A	B	A && A	A || B	! A
`true`	`true`	`true`	`true`	`false`
`true`	`false`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`true`
`false`	`false`	`false`	`false`	`true`

Consideraciones:

- Ambos operadores deben ser Booleanos, si no se debe reportar el error.

4.8. Precedencia y asociatividad de operadores

La precedencia de los operadores indica el orden en que se realizan las distintas operaciones del lenguaje. Cuando dos operadores tengan la misma precedencia, se utilizará la asociatividad para decidir qué operación realizar primero.

A continuación, se presenta la precedencia en orden de mayor a menor de operadores lógicos, aritméticos y de comparación .

Operador	Asociatividad
( ) [ ]	izquierda a derecha
! -	derecha a izquierda
/ % *	izquierda a derecha
+ -	izquierda a derecha
< <= >= >	izquierda a derecha
== !=	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha

4.9. Operador Ternario

El operador ternario es un operador que hace uso de 3 operandos para simplificar la instrucción ‘if’ por lo que a menudo este operador se le considera como un atajo para la instrucción ‘if’. El primer operando del operador ternario corresponde a la condición que debe de cumplir una expresión para que el operador retorna 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.

```
<CONDICIÓN> ?  <EXPRESIÓN> : <EXPRESIÓN> ;
```

Ejemplo:

```
var edad: number = 10;
var es_mayor = falso;
es_mayor = edad > 18 ? true : false;
```

4.10. Sentencias de control de flujo

Las estructuras de control permiten regular el flujo de la ejecución del programa. Este flujo de ejecución se puede controlar mediante sentencias condicionales que realicen ramificaciones e iteraciones. Se debe considerar que estas sentencias se encontrarán únicamente dentro funciones.

4.10.1. Sentencia If Else

Ejecuta un bloque de sentencias si una condición especificada es evaluada como verdadera. Si la condición es evaluada como falsa, otro bloque de sentencias puede ser ejecutado.
SIntaxis:

```
SI -> if ( <EXPRESIÓN> ) { <BLOQUE_SENTENCIAS> }
| if ( <EXPRESIÓN> ) { <BLOQUE_SENTENCIAS> } else { <BLOQUE_SENTENCIAS>}
| if ( <EXPRESIÓN> ) { <BLOQUE_SENTENCIAS> } else SI
```

Ejemplo:

```
if( 3 < 4 ){
// Sentencias
} else if( 2 < 5 ){
// Sentencias
} else {
// Sentencias
}
if (true) { // Sentencias }
if (false) {// Sentencias } else { // Sentencias }
if (false){// Sentencias } else if (true) { // Sentencias }
```

Consideraciones:

- Puede venir cualquier cantidad de if de forma anidada

- La expresión debe devolver un valor tipo **`Bool`** en caso contrario debe tomarse como error y reportarlo.

4.10.2. Sentencia Switch - Case

Evalúa una expresión, comparando el valor de esa expresión con un case, y ejecuta declaraciones asociadas a ese case, así como las declaraciones en los case que siguen.

Si ocurre una coincidencia, el programa ejecuta las declaraciones asociadas correspondientes. Si la expresión coincide con múltiples entradas, la primera será la seleccionada. Si no se encuentra una cláusula de case coincidente, el programa busca la cláusula **`default`** opcional, y si se encuentra, transfiere el control a ese bloque, ejecutando las declaraciones asociadas. Si no se encuentra un default el programa continúa la ejecución en la instrucción siguiente al final del switch. Por convención, el default es la última cláusula.

Sintaxis:

```
case -> switch  (<Expresión>)  {
case expr1:
# Declaraciones ejecutadas cuando el resultado de expresión coincide con
el expr1
break
case expr2:
# Declaraciones ejecutadas cuando el resultado de expresión coincide con
el expr2
break
...
case exprN:
# Declaraciones ejecutadas cuando el resultado de expresión coincide con
exprN
break
default:
# Declaraciones ejecutadas cuando ninguno de los valores coincide con el
valor de la expresión
}
```

Ejemplo:

```
const numero = 2
switch (numero) {
  case 1:
    console.log("Uno");
    break;
  case 2:
```

```
    console.log("Dos");
    break;
  case 3:
    console.log("Tres");
    break;
  default:
    console.log("Invalid day");
}
/* Salida esperada:
Dos
*/
```

4.10.3. Sentencia While

Crea un bucle que ejecuta un bloque de sentencias especificadas mientras cierta condición se evalúe como verdadera (**true**). Dicha condición es evaluada **antes** de ejecutar el bloque de sentencias y al final de cada iteración.

SIntaxis:

```
WHILE -> while  (<Expresión)  {
 <BLOQUE SENTENCIAS>
}
```

Ejemplo:

```
while (true) {
   //sentencias
}
var num: number = 10;
while (num != 0) {
    num -= 1;
    console.log(num);
}
/* Salida esperada:
9
8
7
6
5
4
3
2
1
```

```
0
*/
```

Consideraciones:

- El ciclo while recibirá una expresión de tipo **`Bool`**, en caso contrario deberá mostrar un error.

4.10.4. Sentencia For

Un bucle **`for`** en el lenguaje OLCScript se comportará como un for moderno, que recorrerá alguna estructura compuesta. La variable que recorre los valores se comportará como una constante, por lo tanto no se podrán modificar su valor en el bloque de sentencias, su valor únicamente cambiará con respecto a las iteraciones.

Ejemplo

```
// for que recorre un rango
for (var i: number = 1; i <= 5; i++) {
   console.log(i);
}

var letras: string[] = ["O", "L", "C", "2"];
// for que recorre un arreglo unidimensional
for (var letra of letras ) {
    console.log(letra);
    letra = "cadena"; //error no es posible asignar algo a letra
}
/*Salida esperada:
1
2
3
4
5
O
L
C
2
*/
```

Consideraciones:

- La constante que recorre un arreglo, su tipo será del tipo de dato que contiene el arreglo, cuando recorre un arreglo numérico será de tipo **`Number`** y cuando recorre un arreglo de cadenas será de tipo **`String`**.

4.11. Sentencias de transferencia

Estas sentencias transferirán el control a otras partes del programa y se podrán utilizar en entornos especializados.

4.11.1. Break

Esta sentencia termina el bucle actual ó sentencia switch y transfiere el control del programa a la siguiente sentencia a la sentencia de terminación de estos elementos.

Ejemplo:

```
while  (true) {
  i = 0;
  break; //finaliza el bucle en este punto
}
```

Consideraciones:

- Si se encuentra un **break** fuera de un ciclo y/o sentencia switch se considerará como un error.

4.11.2. Continue

Esta sentencia termina la ejecución de las sentencias de la iteración actual (en un bucle) y continúa la ejecución con la próxima iteración.

Ejemplo:

```
while (3 < 4) {
    continue
}
var i = 0;
var j = i;
while (i < 2) {
    if  j == 0{
          i = 1;
          i += 1
          continue;
    }
    i += 1
}
// i posee el valor de 2 al finalizar el ciclo
```

Consideraciones:

- Si se encuentra un **`continue`** fuera de un ciclo se considerará como un error.

4.11.3. Return

Sentencia que finaliza la ejecución de la función actual, puede o no especificar un valor para ser devuelto a quien llama a la función.

```
function funcion1(): number {
    return 1;  // retorna un valor number
}

function funcion() {
   return; // no retorna nada
}
```

5. Estructuras de datos

Las estructuras de datos en el lenguaje OLCScript son los componentes que nos permiten almacenar un conjunto de valores agrupados de forma ordenada, las estructuras básicas que incluye el lenguaje son los **Array.**

5.1. Array

Los array son la estructura compuesta más básica del lenguaje OLCScript, los tipos de array que existen son con base a los tipos **primitivos e interfaces** del lenguaje. Su notación de posiciones por convención comienza con 0.

5.1.1. Creación de array

Para crear vectores se utiliza la siguiente sIntaxis.
SIntaxis:

```
<Declaracion_array> ->
           <Tipo_declaracion> Id : <TIPO>[] =  <Defincion_array>
<Definicion_array> ->
           [<LISTA EXPRESIONES>]
           | []
           | <Expresion>
<Tipo_declaracion> -> var | const
```

Consideraciones:

- Un array al declararse sin elementos se convertirá en un array vacío
- La lista de expresiones deben ser del mismo tipo que el tipo del array.
- El tamaño del array puede aumentar o disminuir a lo largo de la ejecución.
- Cuando la definición de un array sea otro array, *se hará una copia* del array dando origen a otro nuevo array con los mismos datos del array.

5.1.2. Función push(<Expresión>)

Esta función se encarga de colocar insertar un valor al final del vector, no retorna nada.

5.1.3. Función pop()

Esta función remueve el último elemento de la lista, en caso que el vector esté vació deberá retornar **null**, de lo contrario deberá retornar el elemento que se está sacando.

5.1.4. Función indexOf(<Expresion>)

Retorna el índice de la primer coincidencia que encuentre, de lo contrario retornará -1

5.1.5. Join()

Une todos los elementos del array en un string, separado por comas **Ej**: [1,2,3] -> “1,2,3”

5.1.6. length

Este atributo indica la cantidad de elementos que posee el vector, dicha cantidad la devuelve con un valor de tipo Int

5.1.7. Acceso de elemento:

Los vectores soportan la notación para la asignación, modificación y acceso de valores, únicamente con los valores existentes en la posición dada, en caso que la posición no exista deberá mostrar un mensaje de error y retorna el valor de null.

Ejemplo:

```
//vector con valores
var vec1:number[] = [10,20,30,40,50];
//vector vacío
var vec2: Float[] = [];
//vector vacío
var vec3: String[] = [];

//imprime 0
console.log(vec2.length);
```

```
//inserta 100 al final
vec1.push(100); //[10,20,30,40,50,100]

//inserciones en vacíos
vec2.push(1.0); // [1.0]
vec3.push("cadena"); // ["cadena"]


//elimina la última posición
const valor = vec1.pop(); //[10,20,30,40,50]
console.log(valor); // output: 100

console.log(vec1.indexOf(20)); // output: 1
console.log(vec1.indexOf(100)); // output: -1

const cadena = vec1.join();
console.log(cadena); // output: “10,20,30,40,50”

console.log(vec1.length); // output: 5

//se realiza una copia completa de vector
var copiaVec: int[] = vec1;

//Acceso a un elemento
const val: number = vec1[3]; // val = 40

//asignación con []
vec1[1] = vec1[0]; //[10,10,40,50]
```

5.2. Matrices / Arrays multidimensionales

Las matrices en OLCScript nos permiten almacenar solamente datos de ***tipo primitivo***, la diferencia principal entre el array y la matriz es que esta última organiza sus elementos en n dimensiones y la manipulación de datos es con la notación [ ] *además que* ***su tamaño no puede cambiar en tiempo de ejecución.***

5.2.1. Creación de matrices

Las matrices en OLCScript pueden ser de **2 a n dimensiones** pero solo de un tipo específico, además su tamaño será constante y será definido durante su declaración.

Consideraciones:

- La declaración del tamaño puede ser explícita o en base a se definición.

- Si la declaración es explícita pero su definición no es acorde a esta declaración se debe marcar como un error. Por lo tanto se debe verificar que la cantidad de dimensiones sea acorde a la definida.
- La asignación y lectura valores se realizará con la notación `[ ]`
- Los índices de declaración comienzan a partir de `1`
- Los índices de acceso comienzan a partir de `0`
- Las matrices **no** van a cambiar su tamaño durante la ejecución.
- Si se hace un acceso con índices en fuera de rango se devuelve **`null`** y se debe notificar como un error.
- Si se declara una matriz con índices negativos o 0, será considerado un error

SIntaxis:

```
<Decl_Mat> -> <Tipo_declaracion> Id : <TIPO><Dimensiones_Matriz> = <Listavalores_Mat>

<Tipo_declaracion> -> var | const

<Dimensiones_Matriz> -> <Dimensiones_Matriz>[<Expresion>]
              | [ <Expresion> ]

// const test: number[][] = [[1,2], [1,2], [2,3]]
// const test: number[][][] = [[[1,2], [3,4]], [[1,2], [3,4]], [[4,5], [6,7]]]
<ListaValores_Mat> -> [ <ListaValores_Mat2> ]

<ListaValores_Mat2>-> <ListaValores_Mat2>, [<arg>]
       | [<arg>]

arg -> <ListaValores_Mat2> | <lista_expresion>
```

Ejemplo:

```
const test: number[][][] = [[[0,0], [0,0]], [[0,0], [0,0]], [[0,0],
[0,0]]];

/*
esta matriz sería:
[
  [
   [0, 0], [0, 0]
  ],
  [
   [0, 0], [0, 0]
  ],
  [
   [0, 0], [0, 0]
  ],
  [
   [0, 0], [0, 0]
  ]
]
*/
```

```
//otro ejemplo
var matriz2: string[][][] = [[[“OLC2”, “OLC2”]], [[“OLC2”, “OLC2”]],
[[“OLC2”, “OLC2”]]];

/*
[
 [
  ["OLC2", "OLC2"]
 ],
 [
  ["OLC2", "OLC2"]
 ],
 [
  ["OLC2", "OLC2"]
 ]
]
*/
// declaración mediante definición
var mtx1 : number[][] = [[1,2,3],[4,5,6],[7,8,9]];

var mtx2 : number[][][] =  [[[1,2,3],[4,5,6],[7,8,9]],
[[10,11,12],[13,14,15],[16,17,18]],
[[19,20,21],[22,23,24],[25,26,27]]];


//asignacion de valores
mtx1[1][1] = 10;  //cambia 5 por 10
console.log(mtx1[0][0] ); //imprime 1
console.log(mtx2[0][1][2]); //imprime 6

//error indices fuera de rango - error
mtx1[100][100] = 10;
```

6. Interfaces

El lenguaje OLCScript tiene la capacidad de permitir al programador en crear sus propios tipos compuestos personalizados, estos elementos se les denomina interfaces, las interfaces permiten la creación de estructuras de datos y manipulación de información de una manera más versátil. Estos están compuestos por *tipos primitivos* o por otras interfaces. Las interfaces también pueden ser utilizadas por funciones y arrays.

Las interfaces en el lenguaje OLCScript se manejan por **valor**, esto implica que las interfaces que contenga un interface *no pueden ser del mismo tipo que el interface que los contiene,* esto se aplica para referencias de forma directa e indirecta.

En el caso que un interface posea atributos de tipo interface, estos se manejan por medio de referencia así como sus instancias en el flujo de ejecución. Si un interface es el tipo de retorno o parámetro de una función, también se maneja por referencia.

6.1. Definición:

Consideraciones:

- Las interfaces **solo** pueden ser *declarados* en el ámbito global
- Las interfaces deben tener al menos un atributo.
- No se podrán agregar más atributos a un interface una vez ha sido definido.

Sintaxis:

```
<Def_Interface> -> interface ID {  <Lista_Atributos>   }

<Lista_Atributos> ->  <Lista_Atributos> ; ID : <TIPO>
                    | ID : <TIPO>
```

6.2. Uso de atributos

El lenguaje OLCScript permite la edición y acceso a atributos de los structs por medio del operador '.' , el cual nos permite acceder a los atributos ya sea para asignarles un valor ó acceder al valor.

SIntaxis:

```
Atributo -> ID.ID;
```

6.3. Funciones Especiales

El lenguaje OLCScript ofrece una serie de funciones para las interfaces.

6.3.1. Funcion Object.Keys(<Expresion>)

Esta función permitirá obtener un array con los atributos del interface

6.3.2. Funcion Object.Values(<Expresion>)

Esta función permitirá obtener un array con los valores del interface.

Ejemplo:

```
// Definición de un interface
interface Persona{
    nombre: string;
    edad: number
};

//  creación de una instancia
var persona1:Persona = { nombre: 'Pablo', edad:20};
// acceso a un atributo
console.log(persona1.nombre);
// modificion de un atributo
persona1.nombre = 'Juanito';

console.log("Edad", persona1.edad)

// El atributo no existe, debe lanzar un error
console.log(persona1.apellido)

// Object.Keys()
const llaves: string[] = Object.keys(persona1);
console.log(llaves) // ['nombre', 'edad']

// Object.values()
const valores = Object.values(persona1);
console.log(valores); // ['Juanito', '20']

// uso de la funcion typeof
console.log(typeof persona1); // Persona
```

7. Funciones

En términos generales, una función es un "subprograma" que puede ser llamado por código externo (o Interno en caso de recursión) a la función. Al igual que el programa en sí mismo, una función se compone de una secuencia de declaraciones y sentencias, que conforman el cuerpo de la función.

Se pueden pasar valores a una función y la función puede devolver un valor. Para devolver un valor específico, una función debe tener una sentencia return, que especifique el valor a devolver. Además la función debe tener especificado el tipo de valor que va a retornar.

En el lenguaje OLCScript se manejan los atributos por copia o por valor de forma implícita, esto aplica para cualquier tipo de parámetro para cualquier tipo de dato excepto interfaces y arrays. Las funciones al igual que las variables se identifican con un ID válido.

7.1. Declaración de funciones

Consideraciones:

- Las funciones y variables si pueden tener el mismo nombre.
- Las funciones pueden o no retornar un valor.
- El valor de retorno debe de ser del **mismo** tipo del tipo de retorno de la función.
- Las funciones únicamente pueden ser declaradas en el ámbito global.
- Las funciones solo pueden retornar un valor a la vez.
- No pueden existir funciones con el mismo nombre aunque tengan diferentes parámetros o diferente tipo de retorno.
- El nombre de la función no puede ser una palabra reservada.

Ejemplo:

```
// Ejemplo de función:
function func1(): number{
  return 1;
}

function fn2(): string{
  return "cadena";
}

function funcion(){
  console.log("Hola");
  console.log(" ");
  console.log("Mundo");
}

// función que retorna un array
```

```
function obtenerNumeros(): number[] {
  return [1, 2, 3, 4, 5];
}

// Función que devuelve una interfaz
function obtenerProducto(): Producto {
  const producto: Producto = { id: 1, nombre: 'Producto A'};
  return producto;
}

// error: ya se ha declarado una función llamada funcion previamente
function funcion(): string{
   return "valor";
}

// error: nombre inválido
function if(): number{
   return 21;
}

// error: valor de retorno incompatible con el tipo de retorno
function valor(): string{
  return 10;
}

// error: no se define un tipo de retorno
function invalida() {
   return 1000;
}
```

7.1.1. Parámetros de funciones

Los parámetros en las funciones son variables que podemos utilizar mientras nos encontremos en el ámbito de la función.

Consideraciones:

- Los parámetros de tipo interfaz y array son pasados por referencia, mientras que el resto de tipos primitivos son pasados por valor.
- No pueden existir parámetros con el mismo nombre.
- Pueden existir funciones sin parámetros.
- Los parámetros deben tener indicado el tipo que poseen, en caso contrario será considerado un error.

Ejemplos:

```
// Función suma
function suma(num1: number, num2: number): number {
  return num1 + num2;
}

// función que recibe un array
function obtenerNumArr(arr: number[], index: number): number {
  return arr[index];
}

function addValue(arr: number[], value: number){
  arr.push(value);
  console.log("Se agregó el nuevo valor...");
}

// función suma
function suma(x: number, y: number): number{
   return x + y;
}

//funcion resta
function resta(x: number, y: number): number{
   return x - y;
}

//función mul
function mul(x: number, y: number): number{
   return x * y;
}

// Función que devuelve una interfaz
function obtenerProducto(id: number): Producto {
  const productoUno: Producto = { id: 1, nombre: 'Producto A'};
  const productoDos: Producto = { id: 2, nombre: 'Producto B'};
  if (productoUno.id == id) {
    return productoUno;
  } else {
    return productoDos;
  }
}
```

7.2. Llamada a funciones

Los parámetros en la llamada a una función son los argumentos de la función. Los argumentos se pasan a las funciones por **valor** o **referencia** según sea el caso.

Las llamadas a funciones pueden ser una sentencia o una expresión.

Consideraciones:

- Si se realiza una llamada de una función sin retorno dentro de una expresión, se deberá marcar como un error.
- Se deben verificar que los parámetros sean del mismo tipo esperado que se definió en la declaración.
- Una llamada se puede realizar ya sea si la función fue declarada antes o después de la llamada

Ejemplos:

```
var numero1 = 1;
var numero2 = 1;
var arr = [1,2,3,4,5,6,7];

console.log(suma(numero1, numero2)); //imprime 2

console.log(resta(numero1, numero2)); //imprime 0

console.log(mul(numero1, numero2)); //imprime 1

// llamada por referencia
console.log(arr); //imprime [1,2,3,4,5,6,7]

addValue(arr, 8);
addValue(arr, 9);

console.log(arr); //imprime [1,2,3,4,5,6,7,8,9]

// Obtener una interfaz
console.log(obtenerProducto(1)); //imprime {id:1, nombre:'Producto A'}

console.log(obtenerProducto(4)); //imprime {id:2, nombre:'Producto B'}
```

7.3. Funciones Embebidas

El lenguaje OLCScript está basado en Typescript y este a su vez es un superset de sentencias de Javascript, por lo que en OLCScript contamos con algunas de las funciones embebidas más utilizadas de este lenguaje.

7.3.1. console.log()

Esta función nos permitirá imprimir en cualquier tipo de expresión definida en OLCScript.

Consideraciones

- Puede venir cualquier cantidad de expresiones separadas por coma.
- Se debe de imprimir un salto de línea al final de toda la salida.

Ejemplo:

```
console.log("cadena1","cadena2")  //mostraría: cadena1 cadena2
console.log("cadena1")  // mostraría cadena1
console.log("cadena1 \n cadena2")  // mostraría cadena1
                                   //            cadena2
console.log("valor", 10) // mostraría valor 10
console.log(true) // mostraría true
console.log(1.00001) //imprime 1.00001
```

7.3.2. parseInt()

Esta función permite convertir una expresión de tipo **`String`** en una expresión de tipo **`Number`**. Si la cadena que recibe como parámetro no se puede convertir a un valor numérico se debe desplegar un mensaje de error.

Ejemplo:

```
const w: number = parseInt("3"); // w obtiene el valor de 3

const x: number = parseInt("3.99999"); // x obtiene el valor de 3

const x1: number = parseInt(10.999999) // error: tipo de dato incorrecto

const y: number = parseInt("Q10.00") //error no puede convertirse a
number
```

7.3.3. parseFloat()

Esta función permite convertir una cadena de caracteres en una variable de tipo **`Float`**. Si la cadena que recibe como parámetro no se puede convertir a un valor numérico con punto flotante se debe desplegar un mensaje de error.

```
const w = parseFloat("10") // w obtiene el valor de 10.00

const x = parseFloat("10.001") //x adopta el valor de 10.001

const y = parseFloat("Q10.00") //error no puede convertirse a Float
```

7.3.4. toString()

Esta función es la contraparte de las dos anteriores, es decir, toma como parámetro un valor numérico y retorna una cadena de tipo **`String.`** Además sí recibe un valor **`Bool`** lo convierte en **`"true"`** o **`"false"`**. Para valores tipo **`Float`** la cantidad de números después del punto decimal queda a discreción del estudiante.

```
let int: number = parseInt("1.99999");
let num: number = 23;

console.log(int.toString() + num.toString()); //imprime 123

console.log(true.toString()+"false"); //imprime truefalse

const cadena: string = false.toString() + "->" + int.toString();
console.log(cadena); // imprime false->1
```

7.3.5. toLowerCase()

Esta función es aplicable a cualquier expresión de tipo **string** y su objetivo es convertir el texto reconocido en letras minúsculas.

```
const mayusculas: string = "HOLA MUNDO";
const minusculas: string = mayusculas.toLowerCase();

console.log(minusculas);  // Imprime: hola mundo

const num: number = 10;
console.log(num.toLowerCase());  // Error: tipo de dato incorrecto
```

7.3.6. toUpperCase()

Esta función es la contraparte de la anterior, es aplicable a cualquier expresión de tipo **string** y su objetivo es convertir el texto reconocido en letras mayúsculas.

```
const minusculas: string = "hola mundo";
const mayusculas: string = minusculas.toUpperCase();

console.log(mayusculas);  // Imprime: HOLA MUNDO

const num: number = 10;
console.log(num.toUpperCase());  // Error: tipo de dato incorrecto
```

7.3.7. typeof

Esta función retorna el tipo de dato asociado, este funcionará con tipos primitivos como lo son [string, number, float] y tipos compuestos como las interfaces.

```
let numero: number = 42;
let tipoNumero: string = typeof numero;

console.log(tipoNumero); // imprime "number"

let esVerdadero: boolean = true;
let tipoBooleano: string = typeof esVerdadero;

console.log(tipoBooleano); // imprime "boolean"
```

8. Recuperación de Errores

El compilador se diseñará con la capacidad de identificar y manejar de manera integral todos los errores semánticos, léxicos y sintácticos que puedan surgir durante el proceso de compilación. Cada tipo de error detectado será recolectado y presentado en un reporte detallado al usuario.

Errores Léxicos y Sintácticos:
La detección de errores léxicos y sintácticos se realizará mediante el análisis exhaustivo del código fuente.
En caso de encontrar errores de este tipo, el compilador informará al usuario sobre la naturaleza específica del problema y su ubicación en el código.

Errores Semánticos:
Los errores semánticos, que surgen cuando la sintaxis es correcta pero la lógica no se ajusta a la intención del programador, también serán detectados.
La recuperación de errores semánticos implica la exclusión de la instrucción donde se generó el error, preservando la integridad del resto del programa.

Ejemplos de errores semánticos incluyen el uso de variables no declaradas, entre otros.

Nota:
Cada uno de estos errores deberán de ser "**recopilados**" y presentados en el reporte que se explicará en la siguiente sección.

9. Reportes Generales

Como se indicaba al inicio, el lenguaje OLCScript genera una serie de reportes sobre el proceso de análisis de los archivos de entrada. Los reportes son los siguientes:

9.1. Reporte de errores

El Intérprete deberá ser capaz de detectar todos los errores que se encuentren durante el proceso de compilación. Todos los errores se deberán de recolectar y se mostrará un reporte de errores en el que, como mínimo, debe mostrarse el tipo de error, su ubicación y una breve descripción de por qué se produjo.

No.	Descripción	Ámbito	Línea	Columna	Tipo
1	La interfaz "Persona" no fue encontrada.	Global	5	1	semántico
2	No se puede dividir entre cero.	Global	19	6	semántico
3	El símbolo "¬" no es aceptado en el lenguaje.	Ackerman	55	2	léxico

9.2. Reporte de tabla de símbolos

Este reporte mostrará la tabla de símbolos después de la ejecución del archivo. Se deberán de mostrar todas las variables, funciones y procedimientos que fueron declarados, así como su tipo y toda la información que el estudiante considere necesaria para demostrar que el Intérprete ejecutó correctamente el código de entrada.

ID	Tipo símbolo	Tipo dato	Ámbito	Línea	Columna
x	Variable	Number	Global	2	5
Ackerman	Función	Float	Global	5	1
vector1	Variable	Array	Ackerman	10	5

10. Entregables

El estudiante deberá entregar únicamente el link de un repositorio en GitHub, el cual será privado y contendrá todos los archivos necesarios para la ejecución de la aplicación, así como el código fuente, la gramática y la documentación. El estudiante es responsable de verificar el contenido de los entregables, los cuales son:

10.1. Código fuente de la aplicación
10.2. Gramáticas utilizadas
10.3. Manual Técnico
10.4. Manual de Usuario

Se deben conceder los permisos necesarios a los auxiliares para acceder a dicho repositorio. Los usuarios son los siguientes:

- AndresRodas
- Henry2311
- danielSG95

11. Restricciones

11.1. El proyecto deberá realizarse como una aplicación de escritorio o web utilizando el lenguaje **Python**.
11.2. Es válido el uso de cualquier herramienta y/o librería en Python para el desarrollo de la interfaz gráfica. Se permite la creación de un Frontend con Angula, Vue.js, React, etc. Si el estudiante lo considera necesario.
11.3. Para el analizador léxico y sintáctico se debe implementar una gramática con la herramienta **PLY** de Python.
11.4. Las copias de proyectos tendrán de manera automática una nota de **0 puntos** y los involucrados serán reportados a la Escuela de Ciencias y Sistemas.
11.5. El desarrollo y la entrega del proyecto son de manera **individual**.
11.6. El sistema operativo queda a elección del estudiante.

12. Consideraciones

12.1. Durante la calificación se realizarán preguntas sobre el código para verificar la autoría de este, de no responder correctamente la mayoría de las preguntas ***los tutores harán una verificación exhaustiva en busca de copias*****.**

12.2. Se necesita que el estudiante al momento de la calificación tenga el entorno de desarrollo y las herramientas necesarias para realizar compilaciones, ya que se pedirán pequeñas modificaciones en el código para verificar la autoría de este, en caso que el estudiante no pueda realizar dichas modificaciones en un tiempo prudencial, el estudiante tendrá 0 en la sección ponderada a dicha sección y ***los tutores harán una verificación exhaustiva en busca de copias*****.**

12.3. Se tendrá un máximo de 45 minutos por estudiante para calificar el proyecto. La calificación será de manera virtual y se grabará para tener constancia o verificación posterior.

12.4. La hoja de calificación describe cada aspecto a calificar, por lo tanto, si la funcionalidad a calificar falla en la sección indicada se tendrá 0 puntos en esa funcionalidad y esa nota no podrá cambiar si dicha funcionalidad funciona en otra sección.

12.5. Si una función del programa ya ha sido calificada, esta no puede ser penalizada si en otra sección la función falla o es errónea.

12.6. Los archivos de entrada permitidos en la calificación son únicamente los archivos preparados por los tutores.

12.7. Los archivos de entrada podrán ser modificados solamente antes de iniciar la calificación eliminando funcionalidades que el estudiante indique que no desarrolló.

12.8. La sintaxis descrita en este documento son con fines descriptivos, el estudiante es libre de diseñar la gramática que crea apropiada para el reconocimiento del lenguaje OLCScript.

13. Entrega del proyecto

13.1. La entrega se realizará de manera virtual, se habilitará un apartado en la plataforma de UEDI para que el estudiante realice su entrega.

13.2. No se recibirán proyectos fuera de la fecha y hora estipulada.

13.3. La entrega de cada uno de los proyectos es **individual.**

13.4. Fecha límite de entrega del proyecto: **15 de marzo de 2024**