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

Desarrollar un compilador para el lenguaje de programación **OLCScript** que abarque la fase de análisis, comprendiendo la estructura del código fuente y aplicando la traducción dirigida por la sintaxis utilizando herramientas adecuadas para la generación de analizadores léxicos y sintácticos, además tendrá que generar una traducción equivalente en un lenguaje ensamblador para la arquitectura RISC-V.

1.2. Competencias Específicas

- Utilizar herramientas léxicas y sintácticas, como PLY en Python, para implementar un analizador léxico y sintáctico que reconozca y analice el código fuente de OLCScript, permitiendo al estudiante comprender la estructura del lenguaje y su sintaxis.
- Implementar la traducción orientada por la sintaxis en el compilador, haciendo uso de reglas semánticas y atributos sintetizados y/o heredados.
- Integrar la generación de código assembler para la arquitectura RISC-V dentro del flujo de compilación de OLCScript, asegurando que el código generado sea compatible y eficiente para dicha arquitectura.
- Familiarizar al estudiante con las herramientas y estructuras de datos disponibles para la creación de un compilador, proporcionando una comprensión profunda de los procesos involucrados en la traducción y ejecución de programas escritos en OLCScript en entornos basados en la arquitectura RISC-V.

2. Descripción

La presente sección proporciona una visión general del proyecto que se solicita a los estudiantes, centrándose en el desarrollo de un compilador. Aunque la definición del lenguaje específico se abordará en secciones posteriores, se espera que los estudiantes trabajen con un lenguaje similar derivado de TypeScript. Para lograr este propósito, se empleará **PLY** para el análisis sintáctico y léxico.

Este compilador tiene una estructura particular, ya que no se enfocará en las fases tradicionales de generación de código de tres direcciones ni en la optimización correspondiente. En lugar de ello, se saltará directamente a la etapa de traducción a código ensamblador.

El proyecto requerirá que los estudiantes adquieran un entendimiento profundo de los principios fundamentales de la compilación, así como de los aspectos específicos de la traducción a código ensamblador para la arquitectura RISC-V. Este enfoque permite a los estudiantes explorar de manera práctica cómo se realiza la traducción de un lenguaje de alto nivel a instrucciones de bajo nivel en una arquitectura de conjunto de instrucciones reducido (RISC).

Además, para validar la traducción realizada, se empleará la herramienta en línea ripes.me para ejecutar el código ensamblador resultante. Esto permitirá verificar que el código fuente tenga una traducción equivalente en ensamblador y que el comportamiento del programa sea el esperado.

2.1. Componentes de la aplicación

El objetivo es desarrollar un entorno de desarrollo integral para la creación de aplicaciones en el lenguaje OLCScript. Este entorno, también conocido como IDE (Entorno de Desarrollo Integrado), estará diseñado para analizar el código fuente escrito en OLCScript y proporcionar herramientas esenciales para facilitar el desarrollo de software.

La aplicación contará con los siguientes componentes:

2.1.1. Editor

El editor de código constituirá el núcleo del entorno de trabajo, permitiendo al usuario ingresar y modificar el código fuente. Entre sus características destacadas se incluyen:

2.1.2. Funcionalidades

- Creación de archivos en blanco.
- Apertura de archivos con extensión .olc.
- Guardado de archivos con extensión .olc.
- Soporte para múltiples archivos abiertos simultáneamente.
- Visualización de la línea actual en el código.
- Guardar traducción a Ensamblador
    - El usuario deberá de ser capaz de guardar la traducción generada en un archivo con extensión “.s” Ej: calculadora.s

2.1.3. Herramientas

Este conjunto de herramientas proporcionará funcionalidades adicionales para facilitar el desarrollo y análisis del código OLCScript.

- Ejecución del Compilador
    - Llamada al compilador OLCScript para llevar a cabo los análisis léxico, sintáctico y semántico del código. Además deberá generar una traducción equivalente en código Assembler.

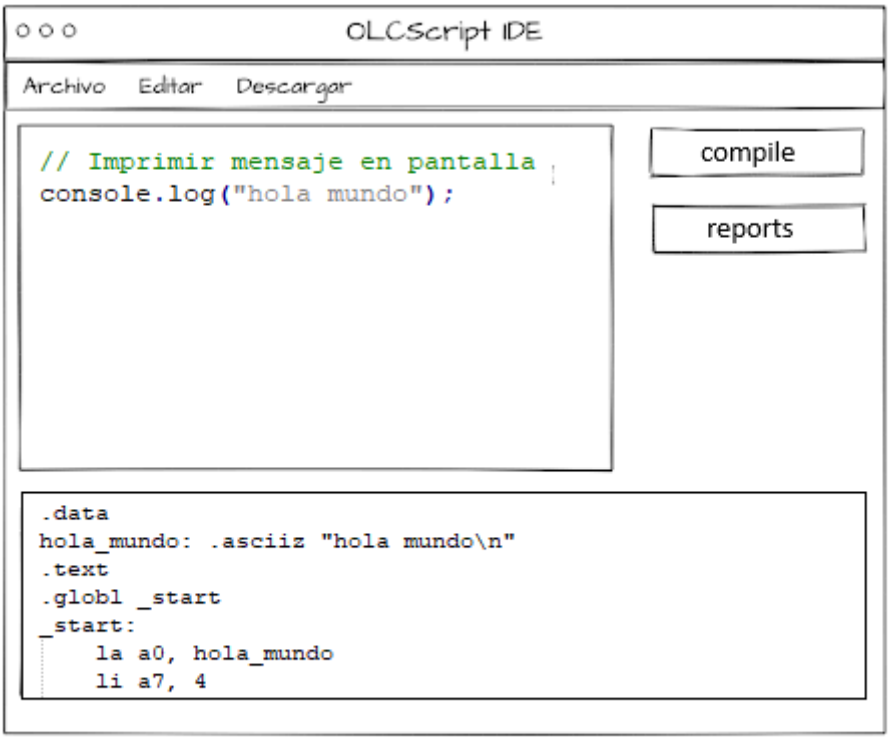
2.1.4. Reportes

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 proporcionará un espacio dedicado para visualizar notificaciones, errores, advertencias e impresiones generadas durante el proceso de análisis del código de entrada.

Impresión de Traducción a Ensamblador:
Después de completar la traducción del código OLCScript a código ensamblador para la arquitectura RISC-V, la traducción resultante se imprimirá en la consola para que el usuario pueda revisar y validar la salida.



(Interfaz propuesta)

2.2. Flujo del proyecto

La ejecución del proyecto sigue un flujo definido según los procesos que los usuarios realizan durante su interacción con el sistema. A continuación, se presenta un desglose paso a paso del flujo del proyecto:

1. Inicio del Programa:
    - El usuario inicia el programa desde la aplicación de escritorio o página web.

2. Código Fuente:
    - El usuario tiene la opción de cargar un archivo existente que contiene el código en OLCScript o crear uno nuevo desde cero.

- Si elige cargar un archivo existente, se abre el documento en el editor de texto integrado.
- Si decide crear un nuevo archivo, se inicia un nuevo documento en blanco en el editor.
- El código fuente puede contener todas las instrucciones disponibles para el lenguaje OLCScript descritas en la sección 3.

3. Compilador OLCScript: En el caso del compilador OLCScript, se diseñará como un compilador de una pasada, significa que las fases para la construcción del compilador se realiza de manera secuencial, sin tener que volver a examinar el código fuente. Esto permite simplificar el diseño y la implementación del compilador. Se construye una tabla de símbolos que contiene información sobre las variables, funciones y otros elementos definidos en el código. De igual manera por simplicidad se omiten algunas fases del proceso y otras se reemplazan como el caso de generacion de código de tres direcciones suplantado por la generación de código ensamblador, a continuación se describen las fases que llevaran a cabo el proceso de compilación:
    - Análisis Léxico: El compilador escanea el código fuente y lo divide en tokens (unidades léxicas como palabras clave, identificadores, operadores, etc.).
    - Análisis Sintáctico: Utiliza la gramática del lenguaje OLCScript para verificar la estructura sintáctica del código y construir un árbol de análisis sintáctico (AST).
    - Análisis Semántico: Verifica que el código cumpla con las reglas semánticas del lenguaje, como el tipo correcto de las variables, la correcta aplicación de operadores, etc.
    - Generación de Código Assembler en RISC-V: Utilizando la información del AST y del análisis semántico, el compilador genera código en lenguaje ensamblador compatible con la arquitectura RISC-V.

4. Generación de Reportes:
    - Se generan informes detallados sobre los errores encontrados durante el análisis, incluyendo errores léxicos, sintácticos y semánticos.
    - Además, se produce un informe que muestra el contenido de la tabla de símbolos, proporcionando una visión general de las variables y funciones declaradas en el código.

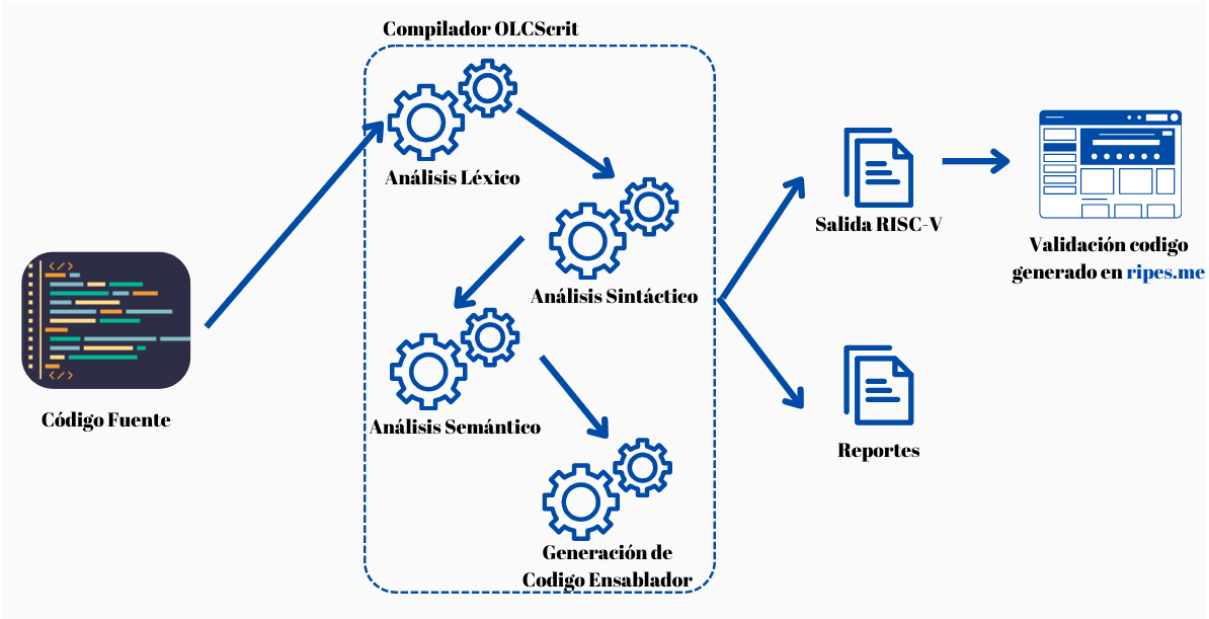
5. Salida en código ensamblador RISC-V: El código ensamblador generado por el compilador estará en formato RV32I, que es adecuado para la arquitectura RISC-V. RV32I se refiere a un conjunto de instrucciones de 32 bits que incluye operaciones necesarias para la traducción. Además el programa debe tener la capacidad de realizar las siguientes acciones:
    - El usuario tiene la opción de guardar la traducción del código en un archivo con la extensión ".s" para la arquitectura RISC-V.

- La traducción del código ensamblador también se muestra en la consola para su visualización inmediata.

6. Validación de Resultados:
   - Para validar los resultados obtenidos, se utiliza la traducción del código ensamblador generado y se ejecuta en ripes.me, una plataforma en línea que simula la arquitectura RISC-V.
   - Se verifica que el programa se ejecute correctamente en el simulador en línea, asegurando así la precisión de la traducción y la funcionalidad del código generado.

7. Fin del Programa:
   - Una vez completadas todas las etapas, el usuario puede cerrar el programa.

Este flujo del proyecto proporciona una guía clara sobre cómo los usuarios interactúan con la aplicación y cómo se llevarán a cabo los procesos clave durante el desarrollo y análisis del código en OLCScript.



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).
- **Compilador**: El código fuente es procesado por **OLCScript**, desarrollado por el estudiante. Como salida se obtendrá la traducción a ensamblador para risc-v
- **Resultado de consola**: Se deberá implementar una consola (Como se expone en interfaz propuesta) donde se pueda visualizar la traducción generada.
- **Reportes**: Se deberán generar reportes de Errores y Tabla de Símbolos. Estos se detallarán más adelante.

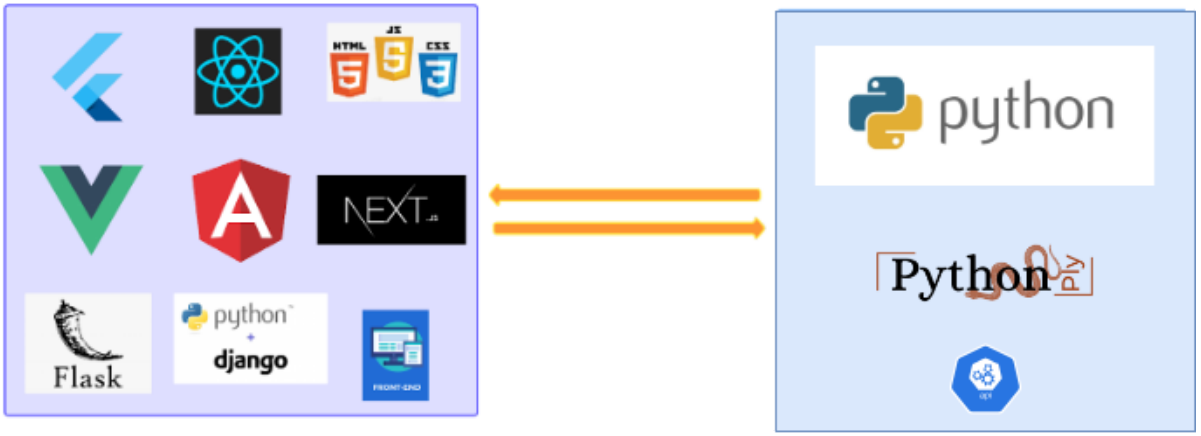
- **Salida:** Se tendrá la posibilidad de guardar la traducción resultante en una archivo con extensión ".s", este se utilizará para hacer la ejecución y validación de la traducción en ripes.me.

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

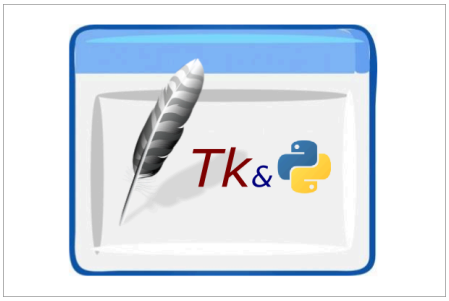
2.3.1. Stack: Frontend / Api - REST

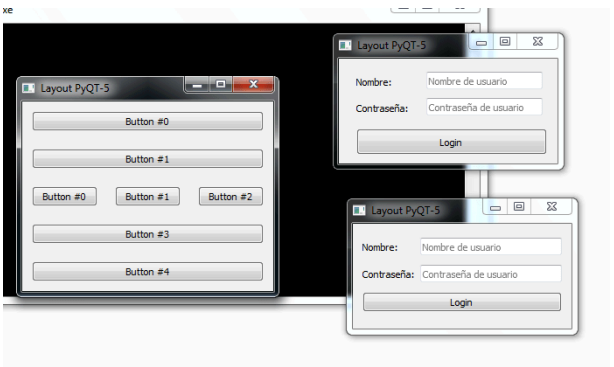
Es permitido crear una Api-REST en Python y luego consumirla con un frontend, la tecnología para desarrollar el frontend queda a discreción del estudiante. Dicha aplicación puede ser implementada de forma local y el framework/librería en Python para construir la API queda a discreción del estudiante.



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 compilador deberá traducir el código fuente en un lenguaje ensamblador. 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`
`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`

Operandos	Tipo resultante	Ejemplo
`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

Operador	Asociatividad
&&	izquierda a derecha
||	izquierda a derecha

4.9. 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.

4.9.1. 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.

4.9.2. 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
*/
```

4.9.3. 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.

4.9.4. 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`**.

4.10. Sentencias de transferencia

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

4.10.1. 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.

4.10.2. 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.

4.10.3. 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. length

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

5.1.6. 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


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]
```

6. 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.

6.1. 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;
}
```

6.1.1. 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;
}
```

6.2. 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]
```

6.3. 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.

6.3.1. 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
```

6.3.2. 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
```

6.3.3. 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
```

6.3.4. 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
```

6.3.5. 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
```

6.3.6. 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
```

6.3.7. typeof

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

```
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"
```

7. 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.

8. Generación de Código Assembler

La generación de código assembler para la arquitectura RISC-V implica la traducción de las estructuras sintácticas y semánticas del lenguaje OLCScript a instrucciones específicas del conjunto de instrucciones RISC-V. Durante este proceso, se mapean las construcciones del lenguaje, como declaraciones de variables, expresiones aritméticas, control de flujo y llamadas a funciones, a secuencias de instrucciones RISC-V que implementan la funcionalidad deseada.

Para la generación de código assembler para la arquitectura RISC-V, se optará por utilizar las instrucciones del conjunto RV32I, que representa una implementación base de 32 bits de RISC-V, ofreciendo un equilibrio óptimo entre simplicidad y funcionalidad. Este conjunto

de instrucciones proporciona un conjunto fundamental de operaciones aritméticas, lógicas y de transferencia de datos que son esenciales para la ejecución de programas OLCScript. Al seleccionar este conjunto de instrucciones, se garantiza la portabilidad y la compatibilidad del código generado en una amplia gama de dispositivos y sistemas que admiten la arquitectura RISC-V de 32 bits. Además, la elección de RV32I simplifica el proceso de desarrollo del compilador al reducir la complejidad asociada con características avanzadas presentes en otros conjuntos de instrucciones, permitiendo un enfoque más centrado en la generación de código eficiente y confiable para aplicaciones OLCScript.

Para verificar la salida del compilador, se emplea el entorno Ripes, es un simulador visual de arquitectura informática y un editor de código ensamblador creado para la arquitectura del conjunto de instrucciones RISC-V que funciona en cualquier navegador web. Ripes proporciona un entorno interactivo para escribir, ejecutar y depurar programas en lenguaje ensamblador RISC-V, asimismo brinda una interfaz gráfica con la capacidad de visualizar los registros en memoria, el funcionamiento del procesador, la interacción con la memoria caché . Utilizando esta plataforma, se podrá cargar el código assembler generado por el compilador para evaluar su correctitud y eficiencia en la arquitectura RISC-V. A través de las capacidades de Ripes, se podrán realizar pruebas exhaustivas y verificar el comportamiento del código generado en diferentes escenarios y condiciones.

Asimismo, se requerirá que el compilador produzca un archivo de salida con extensión ".s" que contenga el código assembler generado. Este archivo será utilizado para la validación dentro del simulador de RISC-V, lo que permitirá ejecutar el código generado en un entorno simulado y verificar su funcionamiento. La generación de este archivo facilitará la integración del compilador con otras herramientas de desarrollo y depuración, así como con flujos de trabajo de compilación más complejos que puedan requerir la interacción con herramientas externas. De esta manera, se garantizará que el código generado por el compilador sea compatible con las herramientas y entornos de desarrollo estándar para la arquitectura RISC-V.

- Para acceder al simulador utilizar el siguiente enlace: https://ripes.me/
- Para acceder a la documentación oficial de RISC-V utilizar el siguiente enlace: https://riscv-programming.org/book/riscv-book.html

8.1 Comentarios

Un comentario es un componente léxico del lenguaje que no es tomado en cuenta para el análisis de la entrada. Se utilizará el formato del mismo RISC-V para comentarios:

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

```
# Este es un comentario en RISC-V
# La siguiente línea carga el valor 10 en el registro x10
li x10, 10
/* Esto es un
comentario
multilínea*/
```

En este ejemplo, las líneas que comienzan con # son comentarios y no tienen ningún impacto en el código ensamblador. Los comentarios son útiles para explicar el código y hacer anotaciones.

8.2 Registros

En ensamblador RISC-V, los tipos de datos generalmente se manejan a nivel de registros y direcciones de memoria. No hay tipos de datos explícitos como en lenguajes de más alto nivel. Sin embargo, podemos considerar los registros como almacenando diferentes tipos de datos según su uso.

8.2.1 Registros de propósito general:

Estos registros se pueden utilizar para almacenar enteros, direcciones de memoria o cualquier otro tipo de datos que se pueda representar en un tamaño de registro (normalmente 32 bits o 64 bits).

```
# Cargar un entero en el registro x10
li x10, 42

# Cargar una dirección de memoria en el registro x11
la x11, mi_etiqueta
```

8.2.2 Registros de coma flotante:

Estos registros se utilizan para operaciones de punto flotante y pueden contener números de punto flotante en formatos como IEEE-754 de precisión simple o doble.

```
# Cargar un número de punto flotante de precisión simple en el registro
f10
flw f10, etiqueta_fp

# Cargar un número de punto flotante de doble precisión en el registro
f11
fld f11, etiqueta_fp_doble
```

8.2.3 Memoria:

La memoria se puede considerar como un tipo de dato, ya que se pueden almacenar diferentes valores en ella. Sin embargo, en ensamblador RISC-V, no hay un tipo de datos explícito para la memoria, sino que se trata más como un almacén de bytes direccionables.

```
# Almacenar un valor en la memoria
sw x10, 0(x11) # Almacena el valor de x10 en la dirección apuntada por
x11

# Cargar un valor de la memoria en un registro
lw x12, 0(x11) # Carga el valor de la dirección apuntada por x11 en x12
```

8.3. Etiquetas

Las etiquetas serán identificadores únicos que indican una ubicación específica en el código fuente, permitiéndonos realizar saltos a lo largo en el código en la ejecución, estas se definirán durante el proceso de compilación, el formato de la etiqueta está definido de la siguiente manera:

```
# Ejemplos
L1:
   # Código para inicializar un bucle
   # ...

L3:
   # Código de bucle
   # ...

   # Actualización del contador del bucle
   addi x10, x10, 1   # Incrementa el contador en 1

   # Comprobación de la condición de salida del bucle
   bne x10, x11, bucle # Salta al inicio sí x10 no es igual a x11

L100:
   # Código después del bucle
```

En el ejemplo anterior se presenta un bucle utilizando etiquetas.

8.4. Saltos

Los saltos son instrucciones que permiten cambiar el flujo de ejecución del programa. En RISC-V, los saltos condicionales e incondicionales se realizan mediante distintas instrucciones.

Instrucciones de salto condicional: Estas instrucciones se utilizan para realizar saltos condicionales basados en el estado de los registros.

```
beq x2, x3, L1 # Salta a la L1 si x2 es igual a x3
bne x4, x5, L2 # Salta a la L2 si x4 no es igual a x5
blt x6, x7, L3 # Salta a la L3 si x6 es menor que x7
```

Instrucciones de salto incondicional: Estas instrucciones siempre realizan un salto a la ubicación especificada, independientemente de alguna condición.

```
j L1 # Salta a la etiqueta L1
jal L2 # Salta a la etiqueta L2 y almacena la dirección de retorno
jr x9  # Salta a la dirección almacenada en el registro x9
```

8.5. Operadores

A continuación se detallan algunas de las operaciones aritméticas básicas que se pueden realizar en lenguaje ensamblador RISC-V. Dependiendo de la extensión específica y de las características del conjunto de instrucciones, puede haber otras operaciones aritméticas disponibles, pero las principales son las siguientes:

Suma:

```
add x3, x1, x2  # x3 = x1 + x2
```

Resta:

```
sub x3, x1, x2  # x3 = x1 - x2
```

Multiplicación: En este ejemplo, mul multiplica el contenido de x1 y x2, y almacena el resultado en x3.

```
mul x3, x1, x2  # x3 = x1 * x2
```

División: En este ejemplo, div divide el contenido de x1 por x2, y almacena el resultado en x3.

```
div x3, x1, x2  # x3 = x1 / x2
```

Incremento y decremento:

```
addi x3, x3, 1  # Incrementa x3 en 1
subi x3, x3, 1  # Decrementa x3 en 1
```

8.5. Ejemplo de Entrada y Salida

Entrada:

```
var a: number = 10;
var b: number = 7;
var result: number;

if (a > b) {
    result = a;
} else {
    result = b;
}

console.log(result);
```

Salida:

```
    .text
    .align 2
    .globl _start

_start:
    # Asignar valores a y b
    li a0, 10   # a = 10
    li a1, 7    # b = 7

    # Comparar a y b
    bge a0, a1, greater_than   # if (a > b) jump to greater_than
    mv a0, a1                  # b to result
    j end                      # jump to end
```

```
greater_than:
    mv a0, a0                   # a to result

end:
    # Imprimir el resultado
    li a7, 93                   # syscall: imprimir entero
    ecall

    # Terminar el programa
    li a7, 10                   # syscall: terminar programa
    ecall

    .data
    .align 2
```

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 compilador 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 compilador 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

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. Para la generación de código ensamblador se deben utilizar las instrucciones RV32 de RISC-V.
11.5. 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.

11.6. El desarrollo y la entrega del proyecto son de manera **individual**.
11.7. 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: **25 de Abril de 2024**