



Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Organización de Lenguajes y Compiladores 1
Catedrático: Ing. Kevin Adiel Lajpop Ajpacaja
Tutor académico: Moises González Fuentes

SECCION C

Segundo semestre 2022

MANUAL DE USUARIO Proyecto 2

MFMScript

Nombre: Moises David Maldonado de León

Carné: 202010833

3/11/2022

Objetivos del sistema

El siguiente sistema va dirigido principalmente para que los estudiantes del curso de Introducción a la Programación y Computación 1 aprendan a programar y tener conocimiento de todas las generalidades de un lenguaje de programación, no obstante, puede enfocarse para toda persona con el deseo de iniciar en el campo de la programación. Se diseñó como parte de una solución informática que permita ser una aplicación de análisis léxico y sintáctico con la facultad de representar mediante un AST (abstract syntax tree) de una manera más sencilla y amigable a la vista del usuario.

Información del sistema

Este sistema posee un analizador léxico y sintáctico, implementados mediante la herramienta Jison, a su vez, genera una tabla de errores (tanto léxicos como sintácticos) detectados con los datos de fila y columna para una mayor facilidad de detección y corrección. La aplicación permite mostrar el árbol de sintaxis abstracta, así como tabla de símbolos y demás reportes. Este sistema se implementó mediante una arquitectura Cliente-Servidor, con el objetivo de que pueda separar los servicios administrados por el intérprete, de la aplicación cliente que se mostrará al usuario final. Para este caso se trabajó con NodeJs y Angular para el backend y frontend respectivamente.

Requisitos del Sistema

Procesador	Intel Core 2 Duo 2 GHz o superior
Memoria RAM	De 2GB en adelante
Espacio en disco duro	1GB mínimo
Sistema operativo	Windows o Mac OS
Conexión a internet	No es necesaria
Herramientas	<i>Se requiere tener las siguientes librerías:</i> <ul style="list-style-type: none">• <i>Graphviz</i> <i>Para instalar las dependencias ejecute el comando:</i> <ol style="list-style-type: none">1. <i>Situarse desde la consola en la carpeta client: npm install.</i>2. <i>Situarse desde la consola en la capeta server: npm install</i>

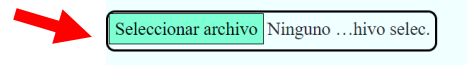
Interfaz gráfica

Página principal

Se despliega un menú principal donde se identifican diferentes componentes.



1. **Seleccionar archivo:** Este botón permite abrir archivos de tipo olc y traslada su contenido hacia el área de código.



2. **Guardar archivo:** Este botón permite al usuario guardar un archivo de olc.



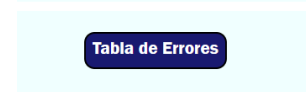
3. **Tabla de símbolos:** Este botón despliega una pestaña nueva del navegador en donde se puede visualizar una imagen con la tabla de símbolos declarados en el código.



4. **Ver AST:** Este botón despliega una pestaña nueva del navegador en donde se puede visualizar una imagen con el AST generado a partir del código.



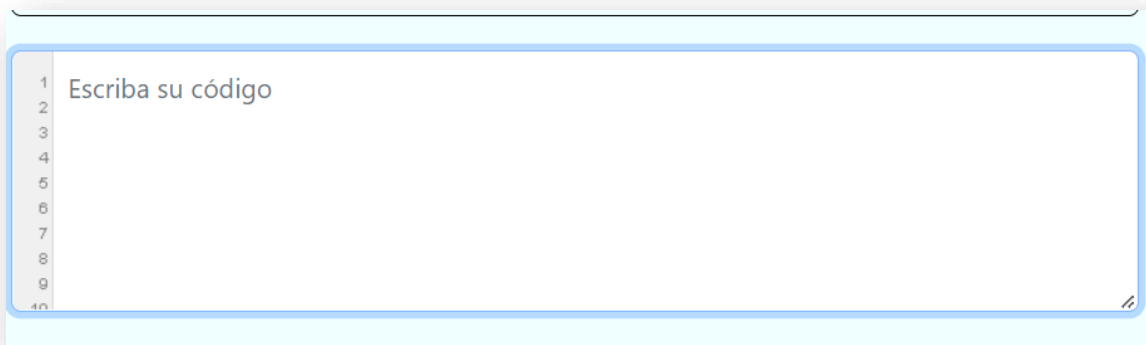
5. **Tabla de errores:** Este botón despliega una pestaña nueva del navegador en donde se puede visualizar una imagen con la tabla de errores encontrados en el código.



6. **Ejecutar:** Este botón captura el código ingresado para su posterior análisis y generación de reportes.



7. **Área de editor:** Es un área de texto con numeración de filas para poder escribir código y analizarlo, así como generar reportes a partir del mismo.



Flujo de funcionalidades del sistema

Paso 1. Escritura del código.

Ingresa el código en el área de texto o de clic en el botón seleccionar archivo y proceda a elegir el archivo (extensión .olc).

-Descripción del lenguaje válido

Tipos de datos reconocidos:

- Int: Números positivos enteros o decimales.
- String: Conjunto de caracteres dentro de comillas dobles.
- Boolean: Tipo de dato verdadero o falso.
- Char: Un carácter dentro de una comilla simple.
- Double: Admite valores numéricos con decimales

Secuencias de escape:

- `\n,\\,\",\\t,\\'`: Puede ingresar estas sentencias dentro de una cadena.

Operaciones básicas:

- Suma: `<operando1> + <operando2>.`
- Resta: `<operando1> - <operando2>`
- Multiplicación: `<operando1> * <operando2>`
- División: `<operando1> / <operando2>`
- Potencia: `<operando1> ^ [<operando2>]`
- Módulo: `<operando1> % <operando2>.`
- Paréntesis: `(<conjunto de operandos>)`
- Negación unaria: `- Operando`

Operadores Relacionales:

- Mayor: `<Expresion> > < Expresion >.`
- Menor: `< Expresion > < < Expresion >`
- Mayor_o_igual: `< Expresion > >= < Expresion >`
- Menor_o_igual: `< Expresion > <= < Expresion >`
- Es_Igual: `< Expresion > == [< Expresion >]`

- Es_Diferente: < Expresion > != < Expresion >.

Operador ternario:

<EXPRESION_BOOLEANA>? <EXPRESION>: <EXPRESION>

Solo como instrucción.

Operadores Lógicos:

- or: <Expresion> || < Expresion >.
- and: < Expresion > && < Expresion >
- not: !< Expresion >

Comentarios:

- //Comentarios: Comentario de una línea.
- /*comentario*/: Comentario de multilínea.

Declaración:

- TIPO: Debe colocar el tipo de dato.
- _ID_: Lista de uno o más nombres de variable separados por comas.
- =: Signo igual.
- Expresión: Puede ser expresión aritmética u otro tipo de dato permitido.
- Cerrar: Debe cerrar con punto y coma la sentencia (;)

Asignación:

- _ID_: Lista de uno o más nombres de variable separados por comas.
- =: Signo igual.
- Expresión: Puede ser expresión aritmética u otro tipo de dato permitido.

Casteos (Aplica para declaración):

(<TIPO>) <EXPRESION>: Debe indicar el tipo de dato entre paréntesis antes de la expresión.

Incremento y decremento:

<EXPRESION>'++'; Para incrementar coloque ++ después de la expresión.
 <EXPRESION> '--'; Para decrementar coloque -- después de la expresión.

Vectores:

```
//DECLARACION TIPO 1 (una dimensión)
<TIPO> '[' ']' <ID> = new <TIPO> '[' <EXPRESION> ']' ';'
<TIPO> '[' '[' ']' <ID> = new <TIPO> '[' <EXPRESION> ']' '[' <EXPRESION> ']' ';'

//DECLARACION TIPO 2
<TIPO> '[' ']' <ID> = '{' <LISTAVALORES> '}' ';'

//Ejemplo de declaración tipo 1
Int [ ] vector1 = new Int[4]; //se crea un vector de 4 posiciones, con 0 en cada posición
Char [ ][ ] vectorDosd = new Char [(Int) "4"] [4]; // se crea un vector de dos dimensiones
de 4x4

//Ejemplo de declaración tipo 2
String [ ] vector2 = {"hola", "Mundo"}; //vector de 2 posiciones, con "Hola" y "Mundo"

Char [ ][ ] vectordosd2 = {{ 0 , 0},{0 , 0}}; // vector de dos dimensiones con valores de 0
en cada posición
```

Acceso a vectores:

```
<ID> '[' EXPRESION ']'
//Ejemplo de acceso
String [ ] vector2 = {"hola", "Mundo"}; //creamos un vector de 2 posiciones de tipo string
String valorPosicion = vector2[1]; //posición 1, valorPosicion = "hola"

Char [ ][ ] vectorDosd = new char [4] [4]; // creamos vector de 4x4
Char valor = vectorDosd[1][1]; // posición 1,1
```

Modificación de vectores:

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

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

String [ ] vector2 = {"hola", "Mundo"}; //vector de 2 posiciones, con "Hola" y "Mundo"
Int [ ] vectorNumero = {2020,2021,2022};
vector2[1] = "OLC1 ";
vector2[2] = "2do Semestre "+vectorNumero[2];
```

Condicional If:

Elif puede repetirse cero, una o más veces, el apartado else es opción opcional. Instrucciones puede generar cualquier sentencia o ciclos descritos en este documento.

```
'if' '(' [<EXPRESION>] ')' '{' [<INSTRUCCIONES>]
}'
| 'if' '(' [<EXPRESION>] ')' '{' [<INSTRUCCIONES>]
}' 'else' '{'
[<INSTRUCCIONES>]
}'
| 'if' '(' [<EXPRESION>] ')' '{' [<INSTRUCCIONES>]
}' 'elif' '(' [<EXPRESION>] ')' '{'
[<INSTRUCCIONES>]
}'
}'
```

Switch Case:

Este permite ejecutar una lista de instrucciones en base a un valor ingresado, existen varias opciones posibles y cuando el valor coincida con una de las opciones se ejecuta un conjunto de instrucciones. Existe una palabra reservada default para ejecutar instrucciones cuando la lista de opciones no se cumple, pero es de forma opcional.

```
'switch' '(' [<EXPRESION>] ')' '{' [<CASES_LIST>] [<DEFAULT>]
}'
| 'switch' '(' [<EXPRESION>] ')' '{' [<CASES_LIST>]
}'
| 'switch' '(' [<EXPRESION>] ')' '{' [<DEFAULT>]
}'
```

```
'case' [<EXPRESION>] ':' [<INSTRUCCIONES>]
```

Ciclo For:

El ciclo o bucle for, es una sentencia que nos permite ejecutar N cantidad de veces la secuencia de instrucciones que se encuentra dentro de ella. Es necesario ingresar un valor inicial y también un valor final, el valor final es el que le indica al ciclo, en momento para terminar de realizar las repeticiones. Otro de los elementos necesarios en este ciclo es el número de pasos que realizará entre cada repetición.

```
'for' '(' ([<DECLARACION>|<ASIGNACION>]); [<CONDICION>]; [<ACTUALIZACION>] ')' '{'
[<INSTRUCCIONES>]
}'
```

Ciclo while:

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

```
'while' '(' [<EXPRESION>] ')' '{' [<INSTRUCCIONES>]
}'
```

Ciclo Do While:

Es una sentencia que ejecuta al menos una vez el conjunto de instrucciones que se encuentran dentro de ella y que se sigue ejecutando mientras la condición sea verdadera.

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


Ciclo Do Until:

Al igual que do while, do until ejecuta al menos una vez el código con la diferencia de que el do while se ejecuta MIENTRAS la condición se cumpla, caso contrario el do until se ejecuta HASTA que la condición se cumpla.

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

Break:

- Break: Palabra reservada.
- Cerrar: Debe cerrar la sentencia con punto y coma (;).

Upper y Lower:

- toUpper (EXPRESION): Recibe como parámetro una expresión de tipo cadena.
- toLower (EXPRESION): Recibe como parámetro una expresión de tipo cadena.

Round:

- Round (EXPRESION): Recibe como parámetro un valor numérico.
- Cerrar: Debe cerrar la sentencia con punto y coma (;).

Otros:

- length (<VALOR>):: Recibe como parámetro un vector, una lista o una cadena.
- typeof (<VALOR>) :: Retorna una cadena con el nombre del tipo de dato evaluado.
- toString (<VALOR>) :: Convertir un valor de tipo numérico o booleano en texto.
- toCharArray (<VALOR>); Permite convertir una cadena en un vector de caracteres.

Push:

Esta instrucción permite ingresar un elemento a un vector de una dimensión.

```
//vector de ejemplo  
string[ ] vector2 = ["hola", "Mundo"];  
  
vector2.push("bonito");
```

Pop:

Esta instrucción eliminara el último elemento de un vector de una dimensión.

```
//vector de ejemplo  
string[ ] vector2 = ["hola", "Mundo", "bonito"];  
  
vector2.pop();
```

Continue:

- Continue: Palabra reservada.
- Cerrar: Debe cerrar la sentencia con punto y coma (;).

Retorno:

- retornar: Palabra reservada.
- Valor: Puede ser expresión aritmética, condición o número
- Cerrar: Debe cerrar la sentencia con punto y coma (;).

Método:

Es una subrutina de código que se identifica con un nombre y un conjunto de parámetros, aunque a diferencia de las funciones estas subrutinas no deben de retornar un valor. Es posible agregar parámetros al método, estos parámetros tendrán definido el tipo de dato y su respectivo nombre. Cada uno de los parámetros estará separado por un carácter coma:

```
<ID> ( [<PARAMETROS>] ) : void {  
  [<INSTRUCCIONES>]  
}  
PARAMETROS -> [<PARAMETROS>] , [<TIPO>] [<ID>]  
| [<TIPO>] [<ID>]
```

Funciones:

Una función es una subrutina de código que se identifica con un nombre, tipo y un conjunto de parámetros. Esta instrucción si es posible reconocer una instrucción de "Retorno" en su estructura. Es posible agregar parámetros al método, estos parámetros tendrán definido el tipo de dato y su respectivo nombre. Cada uno de los parámetros estará separado por un carácter coma.

```
<ID> ( [<PARAMETROS>] ) : <TIPO> {  
  [<INSTRUCCIONES>]  
}  
PARAMETROS ->  
  [<PARAMETROS>] , [<TIPO>] [<ID>]  
|  [<TIPO>] [<ID>]
```

Llamada de funciones y métodos:

Este tipo de instrucción realiza la ejecución de un método o función, para poder realizarlo es necesario ingresar el identificador de la función o método y la lista de parámetros necesarios. En <Lista de parámetros> los parámetros están separados por el carácter coma.

```
LLAMADA -> [<ID>] ( [<PARAMETROS_LLAMADA>] )  
| [<ID>] ()  
  
PARAMETROS_LLAMADA -> [<PARAMETROS_LLAMADA>] , [<ID>]  
| [<ID>]
```

Impresión:

Esta instrucción muestra el contenido de una expresión o valor de una variable. Para poder utilizarla es necesario una expresión o valor de una variable. Al terminar de realizar la impresión se genera un salto de línea por defecto.

```
Println ( <EXPRESION> );
```

```
Print ( <EXPRESION> );
```

Run:

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

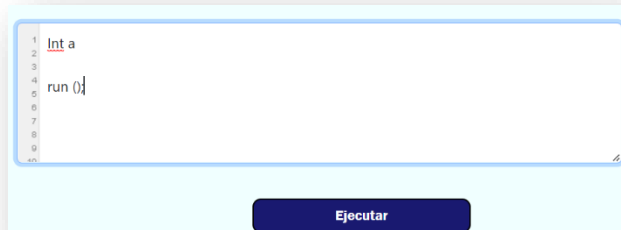
```
run <ID> ( );  
run <ID> ( <LISTAVALORES> )
```

Observaciones:

- ✓ El lenguaje no hace distinción entre mayúsculas y minúsculas.
- ✓ Asegúrese de escribir correctamente la sintaxis.

Paso 2.

De clic en “Ejecutar” para iniciar con el análisis, de existir errores se mostrará una alerta y puede consultar la tabla de errores.



```
1 int a
2
3
4 run 0
5
6
7
8
9
```

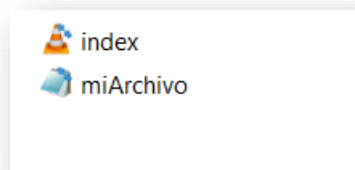
Ejecutar

No.	Tipo	Descripción	Fila	Columna
0	Error sintáctico	No se esperaba ;	3	6

Tabla de Errores

Paso 3 (Opcional).

De clic en guardar archivo. El archivo se creará en la carpeta “server/src” con el nombre predeterminado de “miArchivo”.



Paso 4 (Opcional).

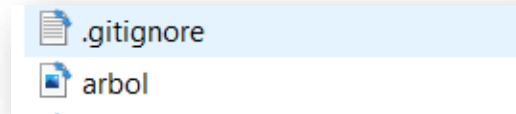
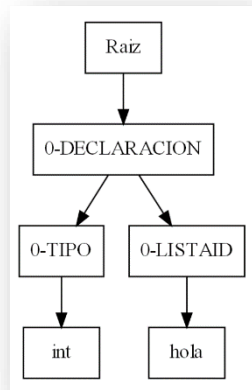
De clic en el botón “Tabla de símbolos” para visualizar la tabla de símbolos.

No.	Nombre	Tipo	Tipo	Fila	Columna
0	hola	Variable	int	1	0

Tabla de Símbolos

Paso 5 (Opcional).

De clic en “Ver AST”. Se generará la imagen y se mostrará automáticamente. La ruta de la imagen será en el directorio del código fuente, en la carpeta server.



Paso 9 (Opcional).

De clic en seleccionar archivo y elija uno de tipo “.olc” para poder ver su contenido.

