Universidad de San Carlos de Guatemala
Facultad de Ingeniería Escuela de Ciencias y Sistemas
Organización de Lenguajes y Compiladores 1
Primer Semestre 2019
Catedráticos:
Ing. Mario Bautista, Ing. Manuel Castillo, Ing. César Batz
Tutores académicos:
Nery Gálvez, Erick Tejaxún, Miguel Ruano

OLC1 Interpreter

Segundo Proyecto de laboratorio

1. OBJETIVOS

1.1 Objetivo General

- Aplicar los conocimientos del curso de Organización de Lenguajes y Compiladores 1 en la creación de soluciones de software.

1.2 Objetivos Específicos

- Aplicar los conceptos de compiladores para implementar el proceso de interpretación de código de alto nivel.
- Aplicar los conceptos de compiladores para analizar un lenguaje de programación y producir las salidas esperadas.
- Aplicar la teoría de compiladores para la creación de soluciones de software.

2. DESCRIPCIÓN GENERAL DEL PROYECTO

- El proyecto número 2 consiste en implementar la fase de análisis léxico, sintáctico y semántico de un compilador para poder ejecutar código de alto nivel a través de la construcción del árbol de sintaxis abstracta (AST). Este proyecto será la continuación de la práctica número dos. Por lo cual, la base será la misma, ampliando su funcionamiento.

3. ENTORNO DE TRABAJO

El entorno de trabajo estará compuesto por un editor y un área de consola y será el principal medio de comunicación entre la aplicación y usuario. En el editor se ingresará el código fuente, y en el área de consola, se mostrará el resultado de la ejecución del código fuente.

3.1.1 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 y columna actual. El editor deberá contar con lo siguiente:

3.1.1 Funcionalidades

- **Crear**: El editor deberá ser capaz de crear archivos en blanco.
- **Abrir archivos**: El editor deberá abrir archivos en formato .fi
- **Guardar el archivo**: El editor deberá guardar el estado del archivo en el que se estará trabajando.
- **Guardar el archivo como:** el editor deberá guardar el archivo en el que se estará

trabajando con la extensión y ruta que el usuario desee.

- **Eliminar pestaña**: permitirá cerrar la pestaña actual.

3.1.2 Herramientas

- **Compilar**: Invocará al intérprete.
- **Iniciar conexión**: Servirá como interruptor para iniciar la conexión con el servidor.

3.1.3 Reportes

- **Reporte de errores:** La aplicación deberá poder generar reporte de errores, será una herramienta que permitirá la identificación de los errores léxicos, sintácticos y semánticos, en el momento de interpretar el lenguaje de alto nivel. Estos errores deben ser almacenados en un archivo en formato HTML con estilos CSS.
- **Generar AST:** se debe de generar una imagen con el AST que se genera después del análisis sintáctico.

3.1.4 Tabla de símbolos

Se requiere un área donde se vea cada una de las variables y métodos declarados en el sistema y su valor final.

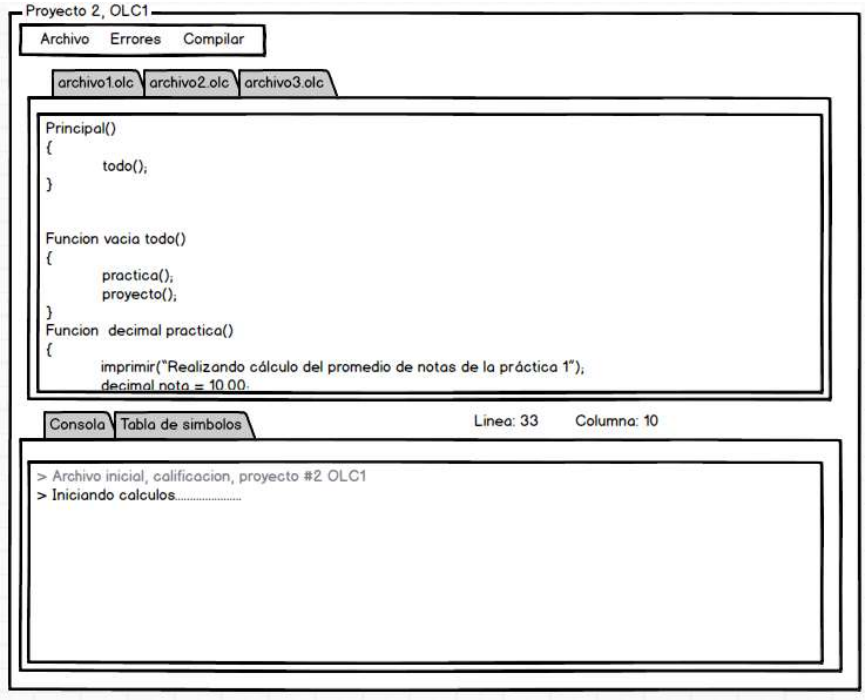

Ilustración 1 Propuesta interfaz gráfica

Área de consola

- El área de consola será la parte donde se mostrarán los mensajes indicados en el lenguaje de alto nivel.

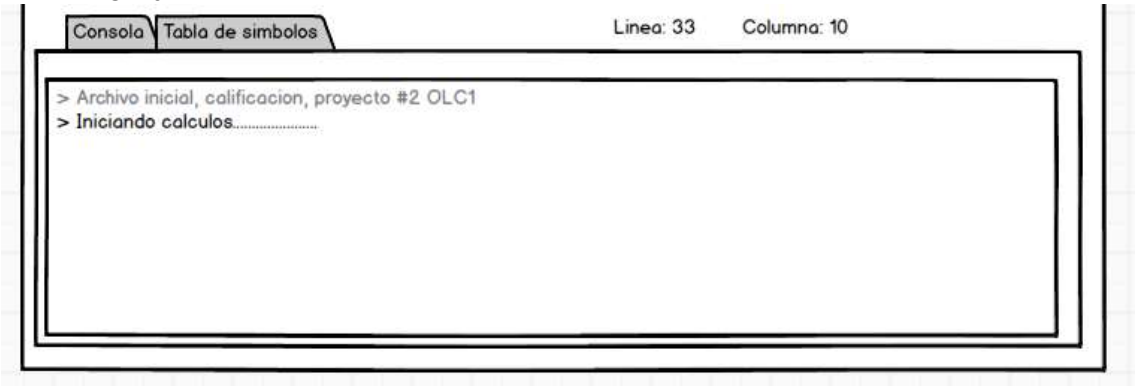

Ilustración 2 Muestra de funcionamiento de consola de salida

4. DESCRIPCIÓN DEL LENGUAJE

A continuación, se describe el lenguaje de programación utilizado en la práctica.

4.1 Case insensitive

El lenguaje no distinguirá la diferencia entre mayúsculas y minúsculas para todas las palabras que acepta el lenguaje por lo que si se declara un atributo de nombre “Contador” este será igual que declarar “contador”.

4.2 Comentarios

4.2.1 Comentario de una línea:

Estos deben empezar con dos signos de “mayor que” y terminar con un salto de línea:

```
>>comentario de una sola línea
```

4.2.2 Comentario multilíneas:

Estos deben empezar con un signo de “menor que” y un guion y termina con un guion y un signo de “mayor que”:

```
<- comentario de varias líneas
Segunda línea
Tercera línea…. ->
```

4.3 Tipo de datos

Los tipos de dato que el lenguaje deberá soportar en el valor de una variable son los siguientes:

Nombre	Descripción	Ejemplo	Observaciones
Int	Este tipo de dato acepta solamente números enteros.	1, 50, 100, 25552, etc.	4 bytes
double	Es un entero con decimal.	1.2, 50.23, 00.34, etc	Se manejará como regla general el manejo de 4 decimales
bool	Admite valores de verdadero o falso, y variantes de los mismos.	Verdadero, falso, true, false.	Si se asigna el booleano a un entero este debe aceptar 1 como verdadero y 0 como false.
char	Solamente admite un carácter por lo que no será posible ingresar cadenas enteras. Viene encerrado en comilla simple.	'a', 'b', 'c', 'E', 'Z', '1', '2', '^', '%', ')', '=', '!', '&', '/', '\', etc	
string	Este es un conjunto de caracteres encerrados en comilla doble.	"cadena 1", "-- ** cadena 1"	Se permitirá cualquier carácter entre las comillas dobles, incluyendo saltos de línea.

4.4 Operadores relacionales

Son los símbolos utilizados en las expresiones, su finalidad es comparar expresiones entre sí dando como resultado, booleanos. En el lenguaje serán soportados los siguientes:

Operador	Descripción	Ejemplo
==	Igualación: Compara ambos valores y verifica si son iguales:	1 == 1 "hola" == "hola" 25.5933 == 90.8883
!=	Diferenciación: Compara ambos lados y verifica si son distintos.	1 != 2, var1 != var2
<	Menor que: Compara ambos lados y verifica si el derecho es mayor que el izquierdo.	(5/(5+5))<(8*8)
<=	Menor o igual que: Compara ambos lados y verifica si el derecho es mayor o igual que el izquierdo.	55+66<=44
>	Mayor que: Compara ambos lados y verifica si el izquierdo es mayor que el derecho.	(5+5.5)>8.98
>=	Mayor o igual que: Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho.	5-6>=4+6

4.5 Operadores Lógicos

Símbolos para poder realizar comparaciones a nivel lógico de tipo falso y verdadero, sobre expresiones.

Operador	Descripción	Ejemplo	Observaciones
||	OR: Compara expresiones lógicas y si al menos una es verdadera entonces devuelve verdadero en otro caso retorna=falso	(55.5<4) || bandera==true Devuelve true	Bandera es true
&&	AND: Compara expresiones lógicas y si son ambas verdaderas entonces devuelve verdadero en otro caso retorna=falso	(flag1) && (“hola” ==”hola”) Devuelve true	flag1 es true
!	NOT: Devuelve el valor inverso de una expresión lógica si esta es verdadera entonces devolverá falso, de lo contrario retorna verdadero.	!var1 Devuelve falso	var1 es true

4.6 Operadores Aritméticos

Símbolos para poder realizar operaciones de tipo aritmética sobre las expresiones en las que se incluya estos mismos.

4.6.1 Suma

Operación aritmética que consiste en reunir varias cantidades (sumandos) en una sola (la suma). El operador de la suma es el signo más +.

Especificaciones sobre la suma:

- Al sumar dos datos numéricos (int, double) el resultado será numérico.
- Al sumar dos datos de tipo carácter (char, string) el resultado será la concatenación de ambos datos.
- Al sumar un dato numérico con un dato de tipo carácter el resultado será la suma del dato numérico con la conversión a ASCII del dato de tipo carácter.
- Al sumar dos datos de tipo lógico (bool) el resultado será un dato lógico, en este caso utilizaremos la suma como la operación lógica or.
- Todas las demás especificaciones se encuentran en la siguiente tabla.

+	int	String	Double	Char	Bool
int	Int	String	Double	Int	Int
String	String	String	String	String	Error
Double	Double	String	Double	Double	Double
Char	Int	String	Double	Int	Int
Bool	int	Error	Double	Int	Bool

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

4.6.2 Resta

Operación aritmética que consiste en quitar una cantidad (sustraendo) de otra (minuendo) para averiguar la diferencia entre las dos. El operador de la resta es el signo menos –

Especificaciones sobre la resta:

- Al restar dos datos numéricos (int, double) el resultado será numérico.
- En las operaciones entre número y carácter, se deberá convertir el carácter a código ASCII.
- No es posible restar datos numéricos con tipos de datos carácter (string)
- No es posible restar tipos de datos lógicos (bool) entre sí.
- Todas las demás especificaciones se encuentran en la siguiente tabla.

-	int	String	Double	Char	Bool
int	Int	Error	Double	Int	Int
String	Error	Error	Error	Error	Error
Double	Double	Error	Double	Double	Double
Char	Int	Error	Double	Int	Error
Bool	int	Error	Double	Error	Error

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

4.6.3 Multiplicación

Operación aritmética que consiste en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). Multiplicación es el asterisco ***.** Las operaciones se podrán realizar solo entre valores o variables del mismo tipo en base a la siguiente tabla cualquier otra combinación es inválida y deberá arrojar un error de semántica.

Especificaciones sobre la multiplicación:

- Al multiplicar dos datos numéricos (int, double) el resultado será numérico.
- No es posible multiplicar datos numéricos con tipos de datos carácter (string)
- No es posible multiplicar tipos de datos string entre sí.
- Al multiplicar dos datos de tipo lógico (bool) el resultado será un dato lógico, en este caso usaremos la multiplicación como la operación AND entre ambos datos.
- Todas las demás especificaciones se encuentran en la siguiente tabla.

*	int	String	Double	Char	Bool
int	Int	Error	Double	Int	Int
String	Error	Error	Error	Error	Error
Double	Double	Error	Double	Double	Double
Char	Int	Error	Double	Int	Int
Bool	int	Error	Double	Int	Bool

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

4.6.4 División

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

Especificaciones sobre la división:

- Al dividir dos datos numéricos (int, double) el resultado será numérico.
- No es posible dividir datos numéricos con tipos de datos de tipo carácter (string)
- No es posible dividir tipos de datos lógicos (bool) entre sí.
- Al dividir un dato numérico entre 0 deberá arrojarse un error de ejecución.
- Todas las demás especificaciones se encuentran en la siguiente tabla.

/	int	String	Double	Char	Bool
int	Double	Error	Double	Double	Int
String	Error	Error	Error	Error	Error
Double	Double	Error	Double	Double	Double
Char	Double	Error	Double	Double	Int
Bool	Double	Error	Double	Double	Error

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

4.6.5 Potencia

Operación aritmética que consiste en multiplicar varias veces un mismo factor. El operador de la potencia es el acento circunflejo **^**

Especificaciones sobre la potencia:

- Al potenciar dos datos numéricos (int, double) el resultado será numérico.
- En las operaciones entre número y carácter, se deberá convertir el carácter a código ASCII.
- No es posible potenciar tipos de datos lógicos (bool) entre sí.

*	int	String	Double	Char	Bool
int	Int	Error	Double	Int	Int
String	Error	Error	Error	Error	Error
Double	Double	Error	Double	Double	Double
Char	Int	Error	Double	Int	Int
Bool	int	Error	Double	Int	Bool

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

4.6.6 Precedencia de análisis y operación de las expresiones lógicas y aritméticas:

Nivel	Operador
0	^
1	/ , *
2	+, -
3	==, !=, <, <=, >, >=
4	!
5	&&
6	| |

Nota: 0 es el nivel de mayor importancia

4.7. Declaraciones y asignaciones de variables

La declaración puede hacerse desde cualquier parte del código ingresado, pudiendo declararlas desde ciclos, métodos y afuera de éstos. La forma normal en que se declaran las variables es la siguiente:

4.7.1 Declaración

Lo que está encerrado en corchetes es opcional puede no venir.
Estructura:

TIPO nombre [nombre2, nombre3…. , nombre" n"] [= Asignación] ;

Ejemplo:

```
int contador;
bool estado;
string cad1,cad2,cad3,ejex,ejey;
```

En este caso las variables no tienen valor alguno pues se declaró, pero no se le ha asignado valor, además, cad1, cad2, cad3, ejex y ejey son de tipo cadena.

Para las variables declaradas en un mismo lugar separadas por coma y que se les asigna un valor, a todas se les asigna el mismo valor. Ejemplo:

```
double contador, contador2 =30.55;
bool f2, f1=true ;
string palabra2, palabra3 = "esto es un ejemplo :D "+"v";
bool bandera2,bandera3=getFlag();
bool flag2, flag3=!false;
bool flag4 = flag2 ;
char letra = 'A' ;
 bool estado = verdadero && falso;
 int contador  = 0 * 9 – getValor() + 2;
 double nota = getNota();
```

4.7.2 Asignación

Esta instrucción nos permite asignarle un valor a una variable ya existente. Debe verificarse que el valor a asignar sea del mismo tipo con el que la variable se ha asignado y que la variable esté ya declarada.

Ejemplo:

```
estado = verdadero && falso;
contador = 0 * 9 – getValor() + 2;
nota = getNota();
```

4.7.3 Aumento

Operación aritmética que consiste en añadir una unidad a un dato numérico. El aumento es una operación de un solo operando. El aumento sólo podrá venir del lado derecho de un dato. El operador del aumento es el doble signo más ++.

Especificaciones sobre el aumento:

- Al aumentar un tipo de dato numérico (int, double, char) el resultado será numérico.
- No es posible aumentar el tipo de dato string.
- No es posible aumentar tipos de datos lógicos (boolean).
- El aumento podrá realizarse sobre números o sobre identificadores de tipo numérico.

Operando	Tipo de dato resultante	Ejemplos
double++	double	4.56++ = 5.56
int++ char++	Int	15++ = 16 'a'++ = 98 = 'b'

4.7.4 Decremento

Operación aritmética que consiste en quitar una unidad a un dato numérico. El decremento es una operación de un solo operando. El decremento sólo podrá venir del lado derecho de un dato. El operador del decremento es el doble signo menos **-**

Especificaciones sobre el decremento:

- Al decrementar un tipo de dato numérico (int, double, carácter) el resultado será numérico.
- No es posible decrementar tipos de datos cadena.
- No es posible decrementar tipos de datos lógicos (boolean).
- El decremento podrá realizarse sobre números o sobre identificadores de tipo numérico.

Operandos	Tipo de dato resultante	Ejemplos
double--	double	4.56-- = 3.56
int-- char--	Int	15-- = 14 'b'-- = 97 = 'a'

4.7.5 Declaración de arreglos

Se pueden realizar declaraciones de arreglos dentro del lenguaje de la aplicación, por lo que la manera de declaración y asignación es parecida a la de variables y es la siguiente:

TIPO array <nombre> [,nomb2,nomb3…. , nombre" n"] ***<dimensiones> [= Asignación];***

Nota: Lo que está encerrado entre corchetes es opcional, y lo que está dentro de los tags es arbitrario.

Las dimensiones serán limitadas a un máximo de 3 dimensiones, esto con el fin de poder facilitar la programación.

Estructura:

[<expresión>][<expresión>].....[<expresión>]

Ejemplo:

```
int array arr1 [a+b+(a+(5+1)*5)];                    //con 1 dimensión
int array arr2,arr3,arr4 [5][5*5+5-4];               //con 2 dimensiones
```

También permitirá asignar valores al arreglo cuando se declare, si no se asigna un valor todas las casillas comienzan con valor nulo o que no tienen valor asignado. Para poder asignar al momento de inicializar se utilizará las llaves "{""}" para indicar los arreglos para cada dimensión y se separa con coma por cada posición o casilla del arreglo.

- 1 dimensión {<expresion>,<expresion>, ……,<expresion>}
- 2 dimensiones {{<expresion>,...,<expresion>},{<expresion>,…,<expresion>}}
- 3 dimensiones {{{<expresion>,...,<expresion>},...,{<expresion>,...,<expresion>}},{{<expresion>,...,<expresion>},…,{<expresion>,...,<expresion>}}}

Ejemplo:

```
int array arr0 [1+2] = {5, 10, 15}; //En este caso arr0 tiene en la primera posición
5,10 en la segunda y 15 en la tercera. Es un arreglo con 3 posiciones.

int array arr10, arr20, arr30 [2] [1+1+1] = {
                                        {5, 10/2, 15*23},
                                        {20, var1^2, 30}
                                       };

int array arr10, arr20, arr30 [2][3][4] = {
                                              {
                                                 {5+2, 4, var3, 2},
                                                 {8, 3, var15, 9},
                                                 {10, var4, var3, 11}
                                              },
                                              {
                                                 {20, var1, 30, 23},
                                                 {35, 42, 10, 3},
                                                 {89, 6, 34, 52}
                                              }
                                          };
```

Nota: lo que puede ir en cada casilla para definir su valor es una expresión.

4.7.6 Uso de arreglos

Una vez definido el arreglo se puede acceder al valor de cada posición del arreglo.
Estructura:

<Destino> = <Nombre del Arreglo> [<Dimensiones>];

Ejemplo:

```
VarEntera1 = arr0[2+3] * 3; // acceso al arreglo 0 en la posición 5
Var2 = arr5[3*1]+5*8+b;  // acceso al arreglo5 en la posición 3
```

4.7.7 Reasignación de las posiciones de los arreglos

Una vez definido el arreglo se puede volver a definir el valor de cada posición del arreglo.
Estructura:

<Nombre del Arreglo> [<Dimensiones>] = <Reasignación>;

Ejemplo:

```
arr0[1*2] = arr0[1] * 7; // reasignación al arreglo 0 en la posición 2
arr5[3] = 5*8+b; // reasignación al arreglo5 en la posición 3
```

Nota: Si el resultado de la expresión excede el tamaño de la dimensión, deberá mostrarse como un error semántico.

4.8 Declaración de clases, métodos y funciones

4.8.1 Clases

Una clase es un conjunto de variables, funciones y métodos. Las especificaciones para las clases son las siguientes:

- Las listas de variables, funciones y métodos son opcionales.
- Un archivo de entrada puede tener varias clases declaradas.
- La extensión entre clases será opcional
- Se podrá crear objetos con todas las clases declaradas dentro del archivo de entrada.
- Las variables globales pueden ser de tipo clase.
- No se llamará a funciones en el ámbito global

Tendrán la siguiente sintaxis:

```
clase <ID> [importar <ID>, <ID>…. <ID>] {
        [ LISTAVARIABLES ]
        [ LISTA_FUNCIONES ]
}
```

Ejemplo:

```
Clase clase1 importar Clase2{
        publico int variableGlobal = 34;
        Privado String cadenaGlobal = “Titulo 1 @”;
        Clase2 c2 = new Clase2();
        Publico array int[2] = {1, 2};
        Publico metodo1 void (int contador, Clase2 cls2){
                variableGlobal = contador;
                int array arr1[3] = {1, 2, 3};

                Clase2 c2 = new Clase2();
                Int suma1 = c2.retornarSUma(2, 3);
                Int suma2 = c2.retornarSUma(4, 8);

                Print(suma1 + suma2);
                Int acceso1 = 1;
                Print  (2+arr1[ acceso1*2 ] );
                Print (c2.retornarsuma(3*6+7, 4));

                Int sumaLocal = funcionSuma(funcionSuma(1, 2), 4);
                Print(sumaLocal)
                Int funcionExtendida = retornarSUMa(2, 3);
                Print

        }
        privado funcionSuma double(double d1, double d2){
                return d1 + d2;
        }
}
clase Clase2{
        int variable;
        publico retornarSUma(double num1, double num2){
                RETURN num1 + num2;
        }
}
```

4.8.2 Declaración de instancias de clase(Objetos)

Para declarar un nuevo objeto se usará la palabra reservada **new** con la que se creará una instancia del objeto indicado, con sus variables globales y métodos implementados. Tendrá la siguiente sintaxis:

```
<Nombre Clase> <id> [ = new <Nombre Clase> (); ]
```

Ejemplo:

```
Clase1 c1 = new Clase1();
Clase1 c2;
```

4.8.3 Acceso a variables, funciones y métodos de una clase

Para acceder a una variable se usará el nombre de la clase, un punto y después el nombre de la variable, función o método que se quiere acceder.

```
<Nombre Clase> . <ID>
<Nombre Clase> . <ID> ( )
<Nombre Clase> . <ID> ( <L PARAMETROS>)
```

Ejemplo:

```
Int numero1 =  c1.num1;
Int num2 =  c1.funcion1(2*var1, 4.5*var3, 45);
Clase2 c2 = c1.funcionRetornarClase();
String cad1 = c2.concatenar("hola", "mundo");
String array arreglo1[2] = funcionRetornarArreglo();
C2.metodoPrintEnConsola();
```

4.8.4 Reasignación de variables globales de la clase

Se puede reasignar o asignar las variables de una clase previamente declarada.

```
<Nombre Clase> . <ID> = <Expresion>
```

Ejemplo:

```
Clase2 c2 = new Clase2();
c2.numero = 3+4;
Print(c2.numero );
```

4.8.4 Sentencia importar

La sentencia importar será una lista de ids que representarán las clases que se quieren importar. Cada clase tendrá la posibilidad de poder importar atributos, métodos y funciones de otra clase indicando el nombre de la clase de la cual se desea obtener toda su estructura, esto permitirá que se logre la unión de varias clases para formar uno solo. Al momento de importar una clase, se deberá buscar la clase padre en el archivo binario de clases cargadas y compiladas para poder importar la clase hijo. La importación de una clase tiene las siguientes características:

- El método main de ejecución de la clase no será heredado de una clase a otra clase, por lo que la clase a importar puede tener su método main de ejecución y la clase que importa puede no tener este método. Al momento de compilar dicha clase dará error por no tener método main de ejecución dado a que este no existe o no fue definido.
- Importar de otra clase es opcional para cada clase que se vaya a crear.

```
clase Clase2 importar clase1, clase2, clase3{
    <L SENTENCIAS>
}
```

4.8.5 Override

Cuando se tengan dos métodos o funciones con un mismo nombre en dos clases que esten importadas, se usará la palabra reservada **Override** para indicar el método que se quiere conservar. Esta declaración será totalmente opcional. Su sintaxis será la siguiente:

```
<Visibilidad> <Id> void override (LParametros )
{
        LSENTENCIAS;
}
```

4.8.6 Visibilidad en variables globales y funciones.

Las variables globales, funciones y métodos puede tener los siguientes tipos de visibilidad. Si la visibilidad NO ESTÁ DECLARADA se tomará como Publico.

4.8.6.1. Publico

Este tipo de visibilidad es el más permisivo de todos. Cualquier componente perteneciente a un objeto **público** podrá ser accedido desde cualquier objeto.

```
clase Clase2 importar clase1, clase2, clase3{
    Publico int entero2;
    String entero3;
    Publico metodo1 void(){
        Print("hola");
    }
}
```

4.8.6.1. Publico

Este tipo de visibilidad es el más restrictivo de todos. Cualquier componente perteneciente a un objeto **privado** solo podrá ser accedido por ese mismo objeto y nada más

```
clase Clase2 importar clase1, clase2, clase3{
    Privado int entero2;
    Privado funcion1  int (int var1, int var2){
        Return var1 + var2;
    }
}
```

4.8.7 Método Main

El método main será el encargado de indicar el flujo del programa. Es decir, la ejecución del programa comenzará con este método. Dentro de un archivo de entrada con varias clases, sólo una clase tendrá declarado el método main, por lo que se debe hacer una pasada para buscar el método main y empezar con la ejecución desde ahí. Tendrá la siguiente sintaxis:

```
Main() {
        LSENTENCIAS;
}
```

Ejemplo:

```
Clase clase1 {
        …..
        Main()
         {
                int contador = 100;
                string saludo = “iniciando ejecución”;
                iniciar(contador, saludo);
         }

}
```

4.8.8 Funciones vacías (Sin retorno)

Una función vacía es un bloque de instrucciones encapsuladas bajo un id que **pueden o no** recibir parámetros para su ejecución. Al terminar su ejecución, no devuelve ningún valor. La sintaxis para la declaración de una función vacía es la siguiente:

```
[Visibilidad] <Id> void [Override] (LParametros )
{
        LSENTENCIAS;
}
```

Ejemplo:

```
Publico Inicio void (int n, string m)
{
        repetir(n)
        {
                print(m);
        }
}
```

4.8.9 Sentencia return

Esta sentencia se encargará de retornar el valor de la expresión indicada.
Esta sentencia deberá venir únicamente dentro de un método con retorno y podrá estar en **cualquier segmento** de dicho bloque. Al ejecutarse sentencia, se hará el cálculo de la expresión indicada y **ya no se ejecutarán las sentencias siguientes** (cuando haya sentencias después de esta). Su sintaxis es la siguiente:

```
Return EXPL;
```

Ejemplo:

```
Return 10 + id;
Return count();
Return flag && flag2;
```

4.8.10 Funciones con retorno

Una función con retorno es un bloque de instrucciones encapsuladas bajo un id que **pueden o no** recibir parámetros para su ejecución. Al terminar su ejecución, este deberá de retornar un valor del tipo del cual se haya declarado utilizando la sentencia RETURN.
La función puede retornar los siguientes tipos:
Int, double, bool, char, string, ARREGLOS, OBJETOS
La sintaxis para la declaración de una función es la siguiente.

```
<VISIBILIDAD> <Id> <TIPO> [Override] (LParametros )
{
        LSENTENCIAS;
 }

 <VISIBILIDAD> <Id> array <TIPO><L_DIMENSION> (LParametros) {
      LSENTENCIAS;
 }

<VISIBILIDAD> <Id> <NOMBRE CLASE> (LParametros )
{
        LSENTENCIAS;
 }
```

```
operacion double (int opcion, double x )
{
          if(opcion == 0)     >> Seno
          {
                  return sin(x);
          }
          else
          {
                 return cos(x);
           }
}

publico funcionObjeto clase1 (clase1 c1){   >> retornar un objeto
     clase1 c2 = new clase1();
     return c2;
}

Privado funcionArreglo Array String[2][3] (){   >> retornar un  arreglo de string
     String array arregloRetorno [2][2+1] = {{1, 2},{3, 4}};
     Return arregloRetorno;
}
```

4.8.11 Llamada de funciones

La sentencia para poder invocar una función vacía o una función con retorno es la misma y consiste en el nombre de la función y entre paréntesis la lista valores que recibe como parámetro la función, separados con comas. Los valores que puede recibir como parámetros son **tipos primitivos, clases, arreglos**. Cumpliendo la siguiente sintaxis:

```
IDFuncion(LParametros) ;
```

Ejemplo de llamada de una función vacía:

```
IniciarMensajes();
funcionSuma(2, var2+5*4)
int var1 = 3;
int array arreglo1[var1] = {1, 2, 3};
funcion1(arreglo1)
```

```
print("Hola mundo");
show("Bienvenido", "Calificación compiladores 1");
```

Cabe resaltar que, en cuanto a las funciones con retorno, se deberá asignar su valor de retorno, es decir, se invocarán como parte de EXPL para la asignación o declaración de una variable. Se debe agregar el no terminal CALL (recomendación de nombre) como una producción del no terminal EXPA (expresiones aritméticas).

Ejemplo de llamada de una función con retorno:

```
Int resultado = suma(10, var2);
Resultado = factorial(resultado);
```

4.9 Función Nativa Print

Esta función será llamada a través de la palabra reservada "print" y contará con un único parámetro que puede ser de cualquier tipo y cualquier tipo de operación. Se agregará el texto resultante a la consola de salida. Tendrá la siguiente sintaxis:

```
print ( EXPL) ;
```

Ejemplo:

```
print (verdadero);
print ("Resultado " + resultado());
print ("Resultado " + verdadero);
```

4.10 Función Nativa Show

Esta función será llamada a través de la palabra reservada "show", y recibirá dos parámetros que pueden ser cualquier tipo de operación. El primer parámetro será el título y el segundo parámetro será el texto del mensaje. Se desplegará un mensaje emergente.

La sintaxis es la siguiente:

```
Show ( EXPL, EXPL) ;
```

Ejemplo:

```
show(titulo(), “Saludo”);
show(“Resultado”, “Mi nota es : “ + getNota());
```

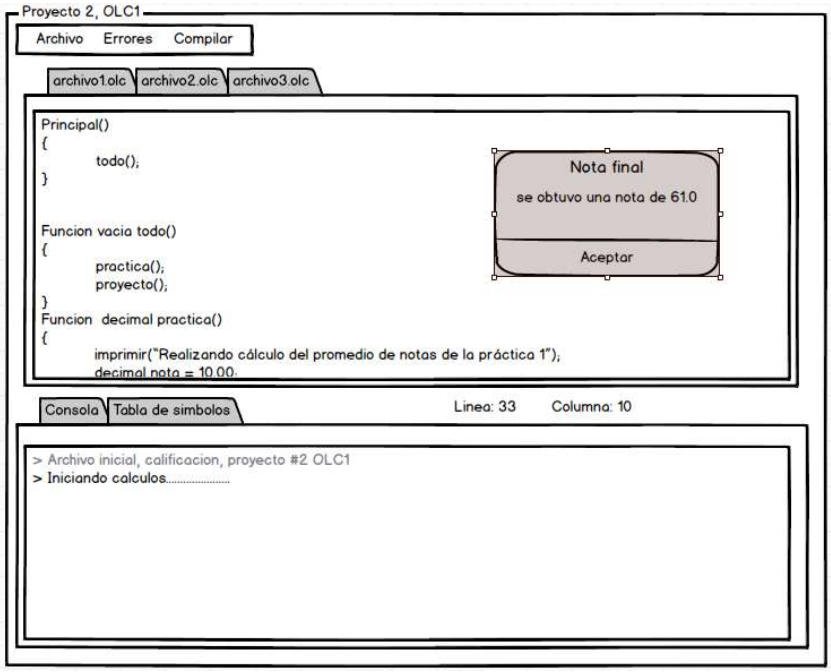

Ilustración 3 Ejemplo de mensaje de salida a través de la función Show

4.11. Sentencias y ciclos

4.11.1 Instrucción IF, ELSE, ELSE-IF

Esta instrucción recibe una condición booleana y si esta se cumple, se ejecutará la lista de instrucciones que traiga consigo. Así mismo pueden venir más condiciones Sino si con sus respectivas condiciones. Si ninguna condición se cumple puede o no que venga la instrucción sino, la cual no tiene condición que evaluar. La sintaxis es la siguiente:

```
If ( condicion ) {
        LISTA_INSTRUCCIONES
    } else if ( Condición ) {
        LISTA_INSTRUCCIONES
    }else if ( Condición ) {
        LISTA_INSTRUCCIONES
    }else  {
        LISTA_INSTRUCCIONES
    }

    >> Otra forma de utilizar la instrucción Si
    if ( condición) {
        LISTA_INSTRUCCIONES
    }

    >> Otra forma de utilizar la instrucción Si
    if ( condición){
        LISTA_INSTRUCCIONES
    }else {
        LISTA_INSTRUCCIONES
    }
```

4.11.2 Ciclo For

Este ciclo ejecutará el número de veces necesarias hasta que la condición se cumpla. Esta tendrá un área de asignación o declaración, una condición y actualización. En la actualización se permiten tanto decrementos como aumentos.

```
For ((ASIGNACION | DECLARACION) ; CONDICION;  ACTUALIZACIÓN)
{
    LINSTRUCCIONES
}
```

Ejemplo:

```
For ( int a = 0 ; a < 10 ; a ++)
 {
     print("Iteracion #"+a);
 }
      >> Otro ejemplo
     Int a;
      for( a = 0; a < 10; a++){
              print ("iteración : " + a);
      }
```

4.11.3 Ciclo Repeat

Este ciclo ejecutará el número de veces que se le indique según el valor entero que se le envié.

```
Repeat((5 - var1) * var 2) {
LINSTRUCCIONES
}
```

Ejemplo

```
Repeat(5*2)
 {   print("Iteracion #"+contador);
     contador++;

}
```

.4.11.4 Ciclo While

Este ciclo ejecutará su contenido siempre que se cumpla la condición que se le dé por lo que podría no ejecutarse si la condición es falsa desde el inicio. La estructura de un ciclo "mientras" es la siguiente:

```
While (CONDICION)
{
    LINSTRUCCIONES
}
```

Ejemplo:

```
While (true)  {
     Print("ciclo…");
     }
```

4.11.5 Sentencia Comprobar

Esta sentencia de control evaluará una variable y ejecutará un contenido dependiendo del valor de ésta, así como también podrá establecerse un contenido "defecto" (opcional en la sentencia) por si la variable no contiene ningún valor de los establecidos previamente. Al final de cada caso estará la sentencia salir.

NOTA IMPORTANTE: Aquí se introduce una nueva palabra reservada, **"salir",** por medio de la cual se podrá salir de cualquier ciclo o sentencia de control sin ejecutar el código que se encuentre por debajo de esta palabra.

La estructura de una sentencia de control "comprobar" es la siguiente.

```
comprobar(variable) {
        caso valor1:
                // Sentencias caso 1;
                [salir] ;
        caso valor 2:
                 // Sentencias caso 2;
                [salir] ;
        caso valor 3:
                // Sentencias caso 3 ;
                [salir];
       //mas casos
        defecto:
                // Sentencias default;
                [salir] ;

      }
```

```
comprobar(var1) {
        caso valor1:
                String cad = " ";
                Salir;
        caso valor 2:
                Print("Valor 2");
                salir;
        caso valor 3:
                Print("valor 3");
                salir;
        defecto:
                 Print("Defecto");
}
```

4.11.6 Hacer-Mientras

Este ciclo ejecutará al menos 1 vez su contenido, luego comprobará la condición para determinar si debe o no ejecutarse nuevamente. La estructura de un ciclo "Hacer-Mientras" es la siguiente:

```
Hacer{
        // Sentencia 1;
        // Sentencia 2;
        // Sentencia 3;
        …
        // Sentencia n;
} mientras(condición);
```

Ejemplo:

```
Hacer{
        // Sentencia 1;
        // Sentencia 2;
        // Sentencia 3;
        …
        // Sentencia n;
} mientras ((1+2)<=(a+b+c)&&(a==b)) ;
```

4.11.7 Sentencia continuar

Para los ciclos puede venir la palabra reservada “Continuar” que lo que hace es que ignora las demás instrucciones siguientes a partir de donde fue escrita de la iteración en curso y pasa a la siguiente iteración.
Ejemplo:

```
Hacer{
        // Sentencia 1;
        // Sentencia 2;
        Continuar;
        // Sentencia 3;
}mientras((1+2)<=(a+b+c)&&(a==b));
```

Se ejecutarían las sentencias 1 y 2 y se obviaría la sentencia 3, para todas las iteraciones del ciclo.
Nota: Para el uso de las sentencias salir o continuar se debe verificar el ambiente, si el ambiente no es el correcto deberá marcar error.

4.12 Otras Funciones Nativas del lenguaje

Las funciones nativas del lenguaje se podrán llamar desde cualquier parte de código, es decir, dentro de funciones, ciclos, etc.

4.12.1 Función nativa add Figure

La función nativa addfigure tiene como función el agregar una imagen al buffer de figuras. Este buffer es intrínseco al sistema, es decir no será necesario declararlo y este será vaciado cada vez que se mande a ejecutar la función figure.

Existen cuatro figuras predeterminadas que son:

- Circle:
- Triangle
- Square
- Line

Para agregar una figura al búfer será de la siguiente manera:

- **Agregar un círculo, este orden de los parámetros**:
  - Color: Expresión de tipo String que tendrá que ser el nombre de un color en idioma inglés o bien el valor en hexadecimal.
  - Radius: Expresión de tipo numérico que indicará el radio del círculo.
  - Solid: Expresión de tipo booleano que indicará si el círculo será solido o bien sólo el contorno.
  - Posx: Expresión de tipo numérico que indicará la posición en la coordenada x del centro del círculo.
  - PosY: Expresión de tipo numérico que indicará la posición en la coordenada y del centro del círculo.

Sintaxis:

```
addFigure( circle(EColor, ERadius, ESolid, EPosX, EPosy));
```

Ejemplo:

```
Boolean flag = 10<2;
Int varx = 10;
Int vary = 20;
addFigure( circle(“red”, 10+20+3, flag, varX, varY));
```

- **Agregar un triángulo, este orden de los parámetros:**
  - Color: Expresión de tipo String que tendrá que ser el nombre de un color en idioma inglés o bien el valor en hexadecimal.
  - Solid: Expresión de tipo booleano que indicará si la figura será solida o bien sólo el contorno.
  - PointNX, pointNY: Expresión de tipo numérico que indicará la posición de uno de los vértices del triángulo. N significa el número de punto, en total deberá haber 3 pares de puntos.

Sintaxis:

```
addFigure( triangle(EColor, ESolid, EP1X, EP1Y, EP2X, EP2Y, EP3X, EP3Y));
```

Ejemplo:

```
Boolean flag = 10<2;
Int varx = 10;
Int vary = 20;
addFigure( triangle(“red”, flag, varX, varY, varX+10, varY+10, varX+100, varY ));
```

- **Agregar un rectángulo, este orden de los parámetros:**
  - Color: Expresión de tipo String que tendrá que ser el nombre de un color en idioma inglés o bien el valor en hexadecimal.
  - Solid: Expresión de tipo booleano que indicará si la figura será solida o bien sólo el contorno.
  - CenterX: Expresión de valor numérico que representa la posición en la coordenada X donde estará el centro de la figura.
  - CenterY: Expresión de valor numérico que representa la posición en la coordenada Y donde estará el centro de la figura.
  - Weight: Expresión de valor numérico que representa la altura de la figura.
  - Height: Expresión de valor numérico que representa la anchura de la figura.

Sintaxis:

```
addFigure( square(EColor, ESolid, EX, EY, EWeigth, Eheight));
```

Ejemplo:

```
Boolean flag = 10<2 || true;
Int varx = 10;
Int vary = 20;
addFigure( square(“#008000”, flag, varX, varY, 100, 20*2));
```

- **Agregar una línea, este orden de los parámetros:**
  - Color: Expresión de tipo String que tendrá que ser el nombre de un color en idioma inglés o bien el valor en hexadecimal.
  - InicioX: Expresión de valor numérico que representa la posición en la coordenada X donde iniciará la figura.
  - InicioY: Expresión de valor numérico que representa la posición en la coordenada Y donde iniciará la figura.
  - FinX: Expresión de valor numérico que representa la posición en la coordenada X donde finalizará la figura.
  - FinY: Expresión de valor numérico que representa la posición en la coordenada Y donde finalizará la figura.
  - Thickness: Expresión numérica que representará el grosor de la línea.

Sintaxis:

```
addFigure( square(EColor, InX, InY, FinX, FinY, thickness));
```

Ejemplo:

```
Int varx = 10;
Int vary = 20;
Int grosor = getGrosor()   +5 ;
addFigure( square(“#008000”, varX, varY,varx+100, vary+100, grosor);
```

4.12.2 Función nativa Figure

La función Figure será la encargada de generar de generar imágenes. Esta función recibirá una Expresión de tipo String que será el título de la figura a generar.

Esta función generará una imagen en donde se encuentren todas las figuras agregadas al búfer. Además, limpiará el búfer.

Sintaxis

```
Figure (EXPL);
Figure (“Flor “ + “de” + “ loto”);
```

Nos generará, un nuevo frame con una imagen.

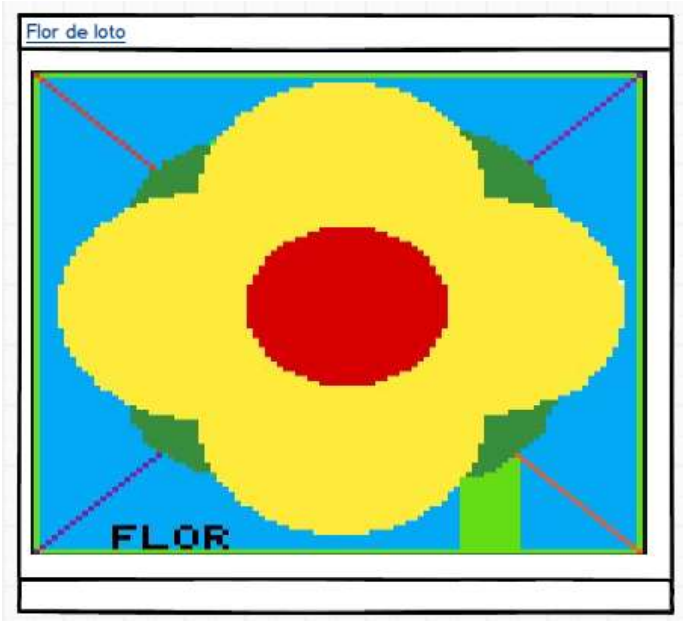

Ilustración 4 Ejemplo de Frame de salida generada por la función figure

5. Restricciones

- La aplicación deberá desarrollarse en lenguaje C# utilizando la plataforma de desarrollo Visual Studio.
- La herramienta para el análisis léxico y sintáctico deberá usar Irony.
- Deben construir su propio árbol AST para la interpretación y ejecución de todas las sentencias.
- El proyecto se realizará de manera individual si se detecta algún tipo de copia el laboratorio quedará anulado.

6. Requisitos mínimos

Los requerimientos mínimos son funcionalidades que garantizan una ejecución básica del sistema, para tener derecho de calificación se debe de cumplir con los siguientes requerimientos:

- Entorno de trabajo
  - Crear archivos
  - Abrir archivos
  - Ejecutar
  - Reporte de errores
- Funcionalidades OLC
  - Clases
  - Extensión
  - Manejo de Procedimientos y Funciones
  - Funciones nativas
  - Declaración y Asignación de variables
  - Operadores Aritméticos, Relacionales y Lógicos
  - Ciclos y Bifurcaciones
    - Si/Sino
    - Para
    - Mientras
    - Método Principal
    - Sentencia de “retorno” (todos los tipos de datos y arreglos)
    - Sentencia salir
  - Arreglos

7. Entregables

- Aplicación funcional.
- Código fuente.
- Gramática escrita en Irony

Fecha de Entrega: miércoles 8 de mayo de 2019 antes de las 23:59