Universidad de San Carlos de Guatemala
Facultad de Ingeniería Escuela de Ciencias y Sistemas
Organización de Lenguajes y Compiladores 1
Escuela de Vacaciones, junio 2019
Catedrático:
Ing. Mario Bautista
Tutor académico:
Miguel Ruano

Transact-OLC1

Proyecto de laboratorio FASE 1

Contenido

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 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 1.

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:

3.1.2 Funcionalidades

- **Crear archivos**: El editor deberá ser capaz de crear archivos en blanco.
- **Abrir archivos**: El editor deberá abrir archivos en formato .tr
- **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.

3.1.3 Características

- **Múltiples pestañas:** el editor deberá ser capaz de crear todas las pestañas que el usuario desee.

3.1.4 Herramientas

- **Compilar**: Invocará al intérprete.

3.1.5 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.

3.1.6 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

3.1.7 Área de consola

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

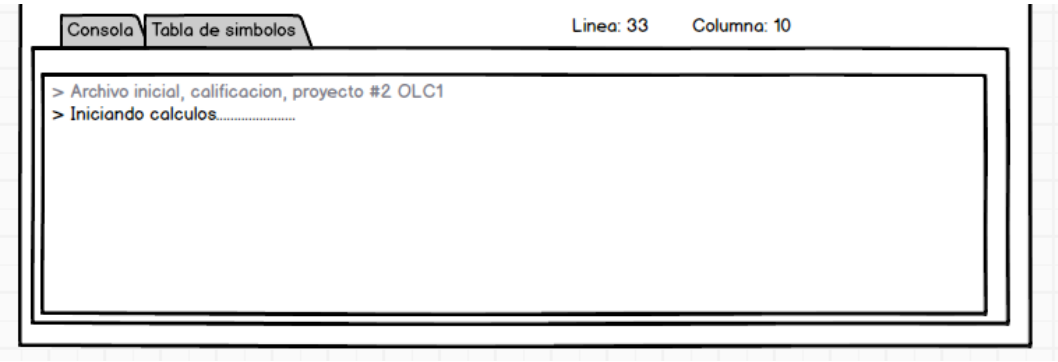

Ilustración 2 Muestra de funcionamiento de consola de salida

4. DESCRIPCIÓN DEL LENGUAJE

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

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 diagonales 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 Tipos de Datos

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

Nombre	Descripción	Ejemplo	Observaciones
Entero	Este tipo de dato acepta solamente números enteros.	1, 50, 100, 25552, etc.	4 bytes
Doble	Es un entero con decimal.	1.2, 50.23, 00.34, etc.	Se manejará como regla general el manejo de 4 decimales
Booleano	Admite valores de verdadero o falso, y variantes de estos.	Verdadero, falso, true, false.	Si se asigna el booleano a un entero este debe aceptar 1 como verdadero y 0 como false.
Caracter	Solamente admite un carácter por lo que no será posible ingresar cadenas enteras. Viene encerrado en comilla simple **recta**.	'a', 'b', 'c', 'E', 'Z', '1', '2', '^', '%', ') ', '=', '!', '&', '/', '\\', '\n', etc.	En el caso de querer escribir comilla simple escribir se escribirá \ y después comilla simple \', si se quiere escribir \ se escribirá dos veces \\, existirá también \n, \t, \r, \''.
Cadena	Este es un conjunto de caracteres encerrados en comilla doble.	"cadena1", "-- ** 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: - Iguales= **True** - No iguales= **False**	1 == 1 ''hola'' == ''hola'' 25.5933 == 90.8883
!=	Diferenciación: Compara ambos lados y verifica si son distintos. - Iguales= **False** - No iguales= **True**	1 != 2, var1 != var2
<	Menor que: Compara ambos lados y verifica si el derecho es mayor que el izquierdo. - Derecho mayor= **True** - Izquierdo mayor= **False**	(5/(5+5))<(8*8)
<=	Menor o igual que: Compara ambos lados y verifica si el derecho es mayor o igual que el izquierdo. - Derecho mayor o igual= **True** - Izquierdo mayor= **False**	55+66<=44
>	Mayor que: Compara ambos lados y verifica si el izquierdo es mayor que el derecho. - Derecho mayor= **False** - Izquierdo mayor= **True**	(5+5.5)>8.98
>=	Mayor o igual que: Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho. - Derecho menor o igual= **True** - Izquierdo menor= **False**	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
!&&	NAND: Compara expresiones lógicas y si son ambas verdaderas entonces ***devuelve = falso*** en otro caso ***retorna = verdadero***	(flag2) !&& (“hola”==”hola”) ***Devuelve falso***	flag2 es true
!	NOT: Devuelve el valor inverso de una expresión lógica si esta es	!var1	var1 es true

	verdadera entonces ***devolverá falso***, de lo contrario ***retorna=verdadero***.	***Devuelve falso***	
! ||	NOR: Compara expresiones lógicas y si al menos una es verdadera entonces ***devuelve=falso*** en otro caso ***retorna=verdadero***	(band1)!||(!¡bandera) ***Devuelve verdadero***	band1=falso bandera= no nulo

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 (entero, doble) el resultado será numérico.
- Al sumar dos datos de tipo carácter (carácter, cadena) 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 (booleano) 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.

+	**Entero**	**Cadena**	**Doble**	**Caracter**	**Booleano**
Entero	Entero	Cadena	Doble	Entero	Entero
Cadena	Cadena	Cadena	Cadena	Cadena	Error
Doble	Doble	Cadena	Doble	Doble	Doble
Caracter	Entero	Cadena	Doble	Entero	Entero
Booleano	Entero	Error	Doble	Entero	Booleano

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 (entero, doble) 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 (cadena)
- No es posible restar tipos de datos lógicos (booleano) entre sí.
- Todas las demás especificaciones se encuentran en la siguiente tabla.

-	**Entero**	**Cadena**	**Doble**	**Carácter**	**Booleano**
Entero	Entero	Error	Doble	Entero	Entero
Cadena	Error	Error	Error	Error	Error
Doble	Doble	Error	Doble	Doble	Doble
Carácter	Entero	Error	Doble	Int	Error
Booleano	Entero	Error	Doble	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 (Entero, doble) el resultado será numérico.
- No es posible multiplicar datos numéricos con tipos de datos carácter (cadena)
- No es posible multiplicar tipos de datos Cadena entre sí.
- Al multiplicar dos datos de tipo lógico (booleano) 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.

*	**Entero**	**Cadena**	**Doble**	**Carácter**	**Booleano**
Entero	Entero	Error	Doble	Entero	Entero
Cadena	Error	Error	Error	Error	Error
Doble	Doble	Error	Doble	Doble	Doble
Carácter	Entero	Error	Doble	Entero	Entero
Booleano	Entero	Error	Doble	Entero	Booleano

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 (entero, doble) el resultado será numérico.
- No es posible dividir datos numéricos con tipos de datos de tipo carácter (cadena)
- No es posible dividir tipos de datos lógicos (booleano) 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.

/	**Entero**	**Cadena**	**Doble**	**Caracter**	**Booleano**
Entero	Doble	Error	Doble	Doble	Entero
Cadena	Error	Error	Error	Error	Error
Doble	Doble	Error	Doble	Doble	Doble
Carácter	Doble	Error	Doble	Doble	Entero
Booleano	Doble	Error	Doble	Doble	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 (entero, doble) 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 (booleano) entre sí.

*	Entero	Cadena	Doble	Carácter	Booleano
Entero	Entero	Error	Doble	Entero	Entero
Cadena	Error	Error	Error	Error	Error
Doble	Doble	Error	Doble	Doble	Doble
Carácter	Entero	Error	Doble	Entero	Entero
Booleano	Entero	Error	Doble	Entero	Booleano

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

4.7 Precedencia de análisis y operación de las expresiones lógicas y aritméticas

Nivel	Operador	Asociatividad
0	^	No asociativa
1	/ , *	Izquierda
2	+, -	Izquierda
3	==, !=, <, <=, >, >=	Izquierda
4	!	Derecha
5	&&, !&&	Izquierda
6	||, !||	Izquierda

Nota: 0 es el nivel de mayor importancia

4.8 Carácter de finalización y encapsulamiento de sentencias

Toda sentencia terminará siempre con el símbolo de dólar $ y el cuerpo de toda clase deberá de encapsularse por signos de agrupación {}. Al igual que en los ciclos esta encapsulación debe de mantenerse, si esto no se cumple se tomará como un error sintáctico.

Ejemplo:

```
NombreClase {
        //Sentencia 1
        $ //Sentencia 2 $ //Ciclo 1 {
                //Sentencia  1$
                //Sentencia      2
                $ //Ciclo 1_1 {
                                        //Sentencia 1 $
                        //Sentencia 2 ….. $
                }
                //Sentencia 3 …. $
        }
        //Sentencia 3 $
}
```

4.8 Declaraciones y asignaciones de variables

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

4.8.1 Declaración

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

Estructura:

TIPO nombre [nombre1, nombre2 ...., nombre n] [= Asignación] $

Ejemplo:

```
entero contador$
booleano estado = true$
cadena cad1, cad2, cad3, ejex, ejey = "hola mundo"$
```

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:

```
doble contador, contador2 = 30.55$
booleano f2, f1 = true$
cadena palabra2, palabra3 = "esto es un ejemplo :D "+"v"$
booleano bandera2, bandera3 = getFlag() $
booleano flag2, flag3 = !false$
booleano flag4 = flag2$
caracter letra = 'A'$
booleano estado = verdadero && falso$
entero contador = 0 * 9 – getValor() + 2$
doble nota = getNota()$
```

4.8.2 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() $
```

Con la asignación el casteo correspondiente para que la integridad del tipo de dato de la variable se mantenga, se debe realizar de acuerdo con la siguiente tabla:

Tipo Variable	Tipo Valor	Resultado de casteo
entero	entero	entero
entero	cadena	error
entero	Booleano	entero
entero	doble	entero
entero	carácter	entero ascii
cadena	<cualquier tipo>	cadena
Booleano	<cualquier tipo>	Error
doble	entero	doble
doble	doble	doble
doble	cadena	error
doble	boolean	error
carácter	entero	ascii
carácter	carácter	carácter
carácter	<cualquier tipo>	error

4.8.3 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 (entero, doble) el resultado será numérico.
- No es posible aumentar el tipo de dato cadena.
- No es posible aumentar tipos de datos lógicos (booleanos).
- El aumento podrá realizarse sobre números o sobre identificadores de tipo numérico.

Operando	Tipo de dato resultante	Ejemplos
doble++	Doble	4.56++ = 5.56
entero++ carácter++	entero	15++ = 16 'a'++ = 98 = 'b'

4.8.4 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 (entero, doble, carácter) el resultado será numérico.
- No es posible decrementar tipos de datos cadena.
- No es posible decrementar tipos de datos lógicos (booleano).
- El decremento podrá realizarse sobre números o sobre identificadores de tipo numérico.

Operandos	Tipo de dato resultante	Ejemplos
doble--	doble	4.56-- = 3.56
entero-- caracter--	entero	15-- = 14 'b'-- = 97 = 'a'

4.9 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:

Arreglo TIPO <nombre> [, nombre2, 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:

```
arreglo entero arr1 [a+b+(i+(5+1)*id1)] $            //con 1 dimensión
arreglo doble arr3, arr4 [contador][5*num+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.

Una dimensión: {<expresion>,<expresion>, ……,<expresion>}
Dos dimensiones: { {<expresion>,…,<expresion>}, {<expresion>,…,<expresion>}}
Tres dimensiones:
{{{<expresion>,…,<expresion>},…,{<expresion>,…,<expresion>}},{{<expresion>,…,<expresion>},…,{<expresion>,…,<expresion>}}}

Ejemplo:

```
Arreglo entero 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.

Arreglo entero arr10, arr20, arr30 [2] [1+1+1] = {
                                        {id1, id2/2, 15*23},
                                        {20, var1^2, 30}
                                       }$

Arreglo doble 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.

4.9.1 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:

```
VarEnt1 = arr0[2+3] * 3 $  // acceso al arreglo 0 en la posición 5
Var2 = arr5[i *1]+5*8+b $  // acceso al arreglo5 en la posición i * 1
```

4.9.2 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+x] = 5*8+b $ // reasignación al arreglo5 en la posición 3+x
```

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

4.10. Declaración de clases

4.10.1 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> [extender <ID>, <ID>…. <ID>] {
        [ LISTA DE VARIABLES ] $
        [ LISTA FUNCIONES | MÉTODOS ]
}
```

4.10.2 Declaración de instancias de clase(Objetos)

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

<Nombre Clase> <id> [= nuevo <Nombre Clase> ()] $

Ejemplo:

```
Clase1 c1 = nuevo Clase1();
Clase1 c2;
Clase2 c2 = nuevo Clase2();
c2.numero = 3+4;
imprimir(c2.numero );
```

4.10.3 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. Estructura:

<Nombre Clase> **.** <ID> $

<Nombre Clase> **.** <ID> ( ) $

<Nombre Clase> **.** <ID> (<L PARAMETROS>) $

Ejemplo:

```
entero numero1 =  c1.num1 $
entero num2 =  c1.funcion1(2*var1, 4.5*var3, 45) $
Clase2 c2 = c1.funcionRetornarClase() $
Cadena cad1 = c2.concatenar("hola", "mundo") $
Arreglo Cadena arreglo1[2] = funcionRetornarArreglo() $
C2.metodoPrintEnConsola() $
```

4.10.4 Reasignación de variables globales de la clase

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

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

Ejemplo:

```
c2.variableentera1 = 34 $
c2.VarCadena = "hola", "mundo" $
```

4.11 Sentencia extender (Herencia)

La sentencia extender será un id que representará la clase desde la que se quiera extender. 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 principal 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 principal de ejecución y la clase que importa puede no tener este método. Al momento de compilar dicha clase dará error por no tener método principal 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.
- Se puede definir la extensión de más de un lienzo, aunque esto puede traer el problema de que estos lienzos posean métodos o funciones con el mismo nombre y que al extender no se pueda saber cuál tomar, la solución a esto es que la última función, método o variable declarada será la que permanezca al final si y solo si esta tiene la propiedad “Sobrescribir”, por lo que si el lienzo el cual se está definiendo tiene un método el cual fue heredado previamente de otro lienzo, este sustituirá al método heredado solo si tiene la propiedad “Conservar”, sino quedará el método, función o variable del último lienzo extendido.

Ejemplo:

```
clase Clase4 extender clase1, clase2, clase3{
    <L SENTENCIAS>
}
```

4.12 Sobrescribir

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

```
Sobrescribir <Visibilidad> <TIPO> <Id> (LParametros )
{
        LSENTENCIAS $
}
```

Ejemplo:

```
Sobrescribir publico entero funcion1 (entero ent1) {
    <L SENTENCIAS> $
}
```

4.13 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.

4.13.1. Publico

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

Ejemplo:

```
clase Clase2 extender clase1, clase2, clase3{
    Publico entero entero2 $
    cadena cadena2 $
    Publico void metodo1(){
        imprimir("hola") $
    }
}
```

4.13.2. Privado

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.

Ejemplo:

```
clase Clase2 extender clase1, clase2, clase3{
    Privado entero entero2$
    Privado entero funcion1 (entero var1, entero var2){
        retornar var1 + var2$
    }
}
```

4.13.3. Protegido

Este tipo de visibilidad permitirá el acceso a los componentes pertenecientes a un objeto protegido únicamente desde el mismo objeto u objetos que hereden de él.

Ejemplo:

```
clase Clase2 extender clase1, clase2, clase3{
    Protegido entero entero2$
    Protegido entero funcion1 (entero var1, entero var2){
        retornar var1 + var2$
    }
}
```

4.14 Método Principal

El método principal 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 principal, por lo que se debe hacer una pasada para buscar el método principal y empezar con la ejecución desde ahí. Tendrá la siguiente sintaxis:

```
Principal(){
    <LSENTENCIAS>
}
```

Ejemplo:

```
Clase clase1 {
        …..
    Principal()
     {
            entera contador = 100 $
            cadena saludo = “iniciando ejecución” $
            iniciar(contador, saludo) $
     }
}
```

4.15 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:

[Sobrescribir] [Visibilidad] void <Id> (<LParametros>)
{
LSENTENCIAS $
}

Ejemplo:

```
Publico void Inicio (entero n, cadena m) {
        repetir(n)
        {
            imprimir(m) $
        }
}
```

4.16 Sentencia retornar

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:

Retornar EXP $

Ejemplo:

```
Retornar 10 + id $
Retornar count()$
Retornar flag && flag2 $
```

4.17 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 RETORNAR
La función puede retornar los siguientes tipos:
entero, doble, booleano, carácter, cadena, ARREGLOS, OBJETOS
La sintaxis para la declaración de una función es la siguiente.

Retornar tipo primitivo:

<Sobrescribir><VISIBILIDAD> <TIPO> <**ID**> (<**L_PARAMETROS**>) {
    LSENTENCIAS $
}

Retornar tipo arreglo:

<Sobrescribir> <VISIBILIDAD> **arreglo <TIPO>**<**L_DIMENSION**><**ID**> (<**L_PARAMETROS**>) {
        LSENTENCIAS $
}

Retornar Clase:

<Sobrescribir><VISIBILIDAD><**NOMBRE_CLASE**><**NOMBRE_FUNCION**>(<**L_PARAM.**>) {
        LSENTENCIAS $
}

Ejemplo:

```
doble operacion(entero opcion, doble x )
{
        SI(opcion == 0)     // Seno
        {
            retornar sin(x)$
        }
        SINO
        {
            retornar cos(x)$
        }
}

publico clase1 funcionObjeto(clase1  c1) {   // retornar un objeto
    clase1 c2 = nuevo clase1() $
    retornar c2 $
}

Privado arreglo entero [2][3] funcionArreglo( ){  // retornar un  arreglo de cadenas.
    Arreglo cadena arregloRetorno[2][2+1] = {{1, 2, 3},{3, 4, 5}} $
    Retornar arregloRetorno $
}
```

4.18 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, OBJETO, ARREGLOS**. Cumpliendo la siguiente sintaxis:

<IDFuncion>(<LParametros>) $

Ejemplo de llamada de funciones:

```
 IniciarMensajes() $
C2.funcionSuma(2, var2+5*4) $
entero var1 = 3 $
arreglo entero arreglo1[var1] = {1, 2, 3} $
funcion1(arreglo1) $
Imprimir(“Hola mundo”) $
MostrarVentana(“Bienvenido”, “Calificación compiladores 1”) $
```

Cabe resaltar que, en cuanto a las funciones con retorno, se deberá agregar a la tabla de símbolos con su valor de retorno. Las funciones se invocarán como si fueran una variable. Se debe agregar el NO TERMINAL::= LLAMADA (recomendación de nombre) como una producción del NO TERMINAL EXP (expresiones aritméticas).

Ejemplo de llamada de una función con retorno:

```
Entero resultado = suma(10, var2) $
Resultado = factorial(resultado) $
```

4.19 Función Nativa Imprimir

Esta función será llamada a través de la palabra reservada “imprimir” 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:

Imprimir (E) $

Ejemplo:

```
Imprimir (verdadero);
Imprimir (“Resultado “ + resultado());
Imprimir (“Resultado “ + verdadero);
```

4.20 Función Nativa MostrarNotificacion

Esta función será llamada a través de la palabra reservada “MostrarNotificacion”, 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:

mostrarNotificacion(E, E) $

Ejemplo:

```
mostrarNotificacion(titulo(),“Saludo”)$
mostrarNotificacion(“Resultado”, “Mi nota es : “ + getNota())$
```

Ejemplo Completo:

```
Clase clase1 extender Clase2{
        publico entero variableGlobal = 34 $
        Privado cadena cadenaGlobal = "Titulo 1 @"$
        Clase2 c2 = new Clase2() $
        Entero tamanio = 2 $
        Publico arreglo entero arr[tamanio] = {1, 2} $
        Publico void metodo(int contador, Clase2 cls2) {
                variableGlobal = contador $
                entero arreglo arr1[3] = {1, 2, 3} $

                Clase2 c2 = nuevo Clase2() $
                entero suma1 = c2.retornarSUma(2, 3) $
                entero suma2 = c2.retornarSUma(4, 8) $

                imprimir(suma1 + suma2) $
                entero acceso1 = 1 $
                imprimir(2+arr1[ acceso1*2 ] ) $
                imprimir(c2.retornarsuma(3*6+7, 4)) $

                entero sumaLocal = funcionSuma(funcionSuma(1, 2), 4) $
                imprimir(sumaLocal) $
                entero funcionExtendida = retornarSUMa(2, 3) $
                imprimir(funcionExtendida) $
        }
        privado doble funcionSuma(doble d1, doble d2){
                retornar d1 + d2 $
        }
}
clase Clase2{
        entero variable $
        publico doble retornarSUma(doble num1, doble num2){
                RETORNAR num1 + num2 $
        }
}
```

4.21. Sentencias y ciclos

4.21.1 Instrucción SI, SINO, SINO-SI

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:

```
SI (condición) {
        LISTA INSTRUCCIONES $
    } SINO SI (Condición) {
        LISTA INSTRUCCIONES $
    } SINO SI (Condición) {
        LISTA INSTRUCCIONES $
    } SINO {
        LISTA INSTRUCCIONES $
    }

    // Otra forma de utilizar la instrucción Si
SI (condición) {
        LISTA INSTRUCCIONES $
    }

    // Otra forma de utilizar la instrucción Si
SI (condición){
        LISTA INSTRUCCIONES $
    } SINO {
        LISTA INSTRUCCIONES $
    }
```

4.21.2 Ciclo Para

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.
Tendrá la siguiente sintaxis.

```
Para ((ASIGNACIÓN | DECLARACIÓN) $ CONDICIÓN$ ACTUALIZACIÓN)
{
    LINSTRUCCIONES $
}
```

Ejemplo:

```
Para ( entero a = 0 $ a < 10 $ a ++)
 {
    Imprimir(“Iteracion #”+a) $
 }
       // Otro ejemplo
Entero a $
Para( a = 0 $ a < 10 $ a++){
     Imprimir(“iteración : “ + a) $
}
```

Nota: La variable de asignación puede ser declarada antes o inicializada directamente en la misma asignación, así como la acción posterior debe de haber cualquier acción posterior desde una asignación con incremento o disminución u otra acción

4.21.3 Ciclo Repetir

Este ciclo ejecutará el número de veces que se le indique según el valor entero que se le envié. Se podrá usar la palabra reservada “**romper”** para terminar el ciclo.

```
Repetir (<NUMERO>) {
  L_INSTRUCCIONES $
 }
```

Ejemplo:

```
Repetir((5 - var1) * var 2)
{   Imprimir (“Iteracion #”+contador) $
   contador++ $
 }
```

.4.21.4 Ciclo Mientras

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. Se podrá usar la palabra reservada “**romper”** para terminar el ciclo. La estructura de un ciclo “mientras” es la siguiente:

```
Mientras (<CONDICION>)
{
    LINSTRUCCIONES $
}
```

Ejemplo:

```
Mientras (true)  {
      Imprimir(“ciclo…”) $
 }
```

4.21.5 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 $
}
```

Ejemplo:

```
comprobar(var1) {
        caso valor1:
                cadena cad = “ ” $
                entero ent = 2;
                Salir $
        caso valor 2:
                imprimir (“Valor 2”)$
                salir$
        caso valor 3:
                Imprimir (“valor 3”)$
                Salir$
        defecto:
                Imprimir (“Defecto”) $
}
```

4.21.6 Hacer-Mientras

Este ciclo ejecutará al menos 1 vez su contenido, luego comprobará la condición para determinar si debe o no ejecutarse nuevamente. Se podrá usar la palabra reservada **romper** para terminar el ciclo. 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)) $
```

4.21.7 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))$
```

Por lo que en este se ejecutará las sentencias 1 y 2, pero siempre se obviaba la sentencia 3 por lo que nunca se ejecutaría, en todas las iteraciones del ciclo Hacer-Mientras.
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.

4.21.7 Sentencia Incluir

Esta sentencia servirá para extender la funcionalidad del archivo al copiar todas las clases contenidas en los archivos indicados, funciona como un include de C. Tomará como parámetro un string que contenga el nombre del archivo a incluir, si no se le indica la ruta, se asumirá la misma ruta del archivo donde se llama a la función:

Incluir("<NOMBRE_ARCHIVO>") $

Ejemplo:

```
incluir("archivo1.tr")$
incluir("carpeta/archivo2.tr")$
```

4.22 Funciones para crear el archivo .der

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

Las funciones generarán un archivo como el de la práctica 1, para ello existen objetos de tipo ***Archivo*** que tendrá propiedades como "Conjunto", las cuales generarán las acciones correspondientes al ejecutar el método ***escribir ().*** Todas las operaciones para el tipo de dato ***Archivo*** se detallan a continuación.

4.22.1 Crear Archivo .der

Esta sentencia creará una estructura de tipo ***archivo***, la cual se podrá guardar en una variable. La sentencia recibirá como parámetro una cadena con el nombre del archivo. La sintaxis es la siguiente:

crearArchivo (<NOMBRE_ARCHIVO>)$

Ejemplo:

Archivo file1 = **crearArchivoDer**("Arhivo1.der") $

4.23 Funciones para manipular el archivo .der

4.23.1 Función Crear Conjunto

Esta función creará una sección de tipo ***conjunto***, la cual recibirá como parámetro una cadena con el nombre del conjunto, seguido de una cadena con la sintaxis correspondiente para la creación de conjuntos de la **Practica 1**. Esta función insertará el texto correspondiente en el archivo de salida de tipo ***.der***. La sintaxis es la siguiente:

crearConjunto (<NOMBRE_CONJ>, <SINTAXIS_DEL_CONJUNTO>)$

Ejemplo:

file1**.crearConjunto**("letra", "a~z") $
file2**.crearConjunto**(idLetra1, "a, b, c") $

Salida esperada (Se deberá escribir debajo del símbolo "**{**" y arriba de "**%%**"):

CONJ: letra -> a~z **;**

4.23.2 Función Crear Expresión Regular

Esta función creará una sección de tipo ***Expresión Regular***, la cual recibirá como parámetro una cadena con el nombre de la expresión regular, seguido de una cadena con la sintaxis correspondiente para la creación de Expresiones regulares de la **Practica 1**. Esta función insertará el texto correspondiente en el archivo de salida de tipo ***.der***. Se pueden usar los conjuntos definidos con anterioridad. Para poder usar las dobles comillas se usará el signo de escape \ **(\"). No se podrá utilizar** el carácter especial de comilla doble. La sintaxis para la función es la siguiente:

crearRegex (<NOMBRE_REGEX>, <SINTAXIS_DE_LA_REGEX>)$

Ejemplo:

file1**.crearRegex**("ExpReg1", "**.** {letra} * | **\"_\"** | {letra} **\'** ") $

Salida esperada (Se deberá escribir debajo del símbolo "**{**" y arriba de "**%%**"):

ExpReg1 -> **.** {letra} * | "_" | {letra} **\'** **;**

4.23.3 Función Crear Entrada

Esta función creará una sección de tipo **lexema**, la cual recibirá como parámetro una cadena con el nombre de la expresión regular a usar, seguido de una cadena con la sintaxis correspondiente para la creación de lexemas de entrada de la **Practica 1**. Esta función insertará el texto correspondiente en el archivo de salida de tipo ***.der***. La sintaxis es la siguiente:

crearEntrada (<NOMBRE_REGEX>, <LEXEMA_ENTRADA>)$

Ejemplo:

file1**.crearEntrada**("ExpReg1", "34.44") $

Salida esperada (Se deberá escribir debajo del símbolo "**%%**" y arriba de "**}**"):

ExpReg1 : "34.44"

4.23.4 Función Guardar Archivo

Esta función guardará el archivo en la ruta especificada por la función crearArchivo, el archivo deberá contener todas las secciones creadas con anterioridad para esa variable de tipo archivo.

guardarArchivo ( )$

Ejemplo:

file1**.guardarArchivo**() $

Salida esperada

```
{
  ////// CONJUNTOS
 CONJ: letra -> a~z;
 /////// EXPRESIONES REGULARES
 ExpReg1 -> . {letra} * | "_" | {letra} \' ;

 %%
 %%

 ExpReg1 : "34.4" ;
}
```

4.24 Funciones para crear cuentas

4.24.1 Crear Cuenta

La función crear cuentas creará un objeto de tipo ***cuenta*** el cual tendrá métodos y propiedades nativas. Este objeto se podrá guardar en una variable. Recibirá como primer parámetro el nombre del archivo XML que se consultará y como segundo parámetro el nombre de una **expresión regular** que se encuentre en ese archivo XML. La sintaxis para creación de cuentas en la siguiente.

crearCuenta(<NOMBRE_ARCHIVO_XML>, <NOMBRE_REGEX>) $

Ejemplo:

```
Cuenta c1 = CrearCuenta("Archivo1.XML", "ExpReg1") $
Cuenta c2 = CrearCuenta(idArchivo1, IdRegex) $
```

4.24.2 Verificar Cuenta

Esta función leerá el archivo XML descrito en la función CrearCuenta(), seguidamente verificará si existe el identificador en el archivo y por último verificará si dentro de la propiedad <Resultado> de esa expresión regular se indica que es una **Cadena Valida.** Si la cadena es válida la función VerificarCuenta() retornará TRUE, si es inválida retornará FALSE. La sintaxis es la siguiente.

<ID>.VerificarCuenta() $

Ejemplo:

```
Booleano cuentaValida = c1.verificarCuenta() $
SI (c2.VerificarCuenta()){
    Sentencias $
}
```

4.24.3 Depositar en Cuenta

Esta función aumentará de forma interna una variable de tipo doble guardada dentro del objeto cuenta, con el monto especificado en el primer parámetro. La sintaxis es la siguiente.

<ID>.DepositarCuenta(<NUMERO>) $

Ejemplo:

```
C1.depositarCuenta(100.20) $
C2.depositarCuenta(entero2) $
```

4.24.4 Restar en Cuenta

Esta función restará de forma interna una variable de tipo doble guardada dentro del objeto cuenta, con el monto especificado en el primer parámetro. Si la variable interna luego de restar es menor a 0 se deberá cancelar la transacción. La sintaxis es la siguiente.

<ID>.RestarCuenta(<NUMERO>) $

Ejemplo:

```
C2.restarCuenta(1300.20) $
C1.restarCuenta(entero4) $
```

4.24.5 Imprimir transacciones de Cuenta

Esta función retornará una cadena con todas las transacciones realizadas, separadas por punto y

coma, la sintaxis de la salida se especifica más adelante. La sintaxis para imprimir las transacciones es la siguiente.

<ID>.ImprimirTransaccionesCuenta() $

Ejemplo:

```
Cadena valorString = C2.imprimirTransaccionesCuenta() $
```

Salida esperada para la cadena:

Sintaxis:

<TIPO_OPERACION> :: <MONTO>; […. <TIPO_OPERACION> :: <MONTO>;]

Ejemplo:

"DEPOSITAR :: 100; RESTAR:: 50; DEPOSITAR :: 200;"

4.24.6 Consultar Saldo de Cuenta

Esta función retornará el monto actual de la cuenta, el retorno será de tipo doble.

<ID>.ConsultarCuenta() $

Ejemplo:

```
doble monto = c1.consultarCuenta() $
```

4.24.7 Transferir de cuenta a cuenta

Esta función tomará como parámetros dos variables de tipo **cuenta** y transferirá el monto indicado en el tercer parámetro. Esta transacción se realizará de la segunda a la primera cuenta. Si la cuenta a restar no tiene suficiente dinero se deberá mostrar un error.

Transferir (<CUENTA1>, <CUENTA2>, <MONTO>) $

Ejemplo:

```
Transferir( cuenta1, cuenta2, 499 )$
```

Nota: En este ejemplo se restan 499 a cuenta2 y se depositan en cuenta1.

5. Restricciones

- La aplicación deberá desarrollarse en lenguaje C++ utilizando la plataforma de desarrollo QtCreator.
- La herramienta para el análisis léxico y sintáctico deberá usar Flex/Bison.
- 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. Descripción de fases

Fase1:

- Análisis léxico y sintáctico del enunciado
- Reporte de errores
- Operaciones aritméticas, lógicas y relacionales
- Declaración y asignación de variables de todos los tipos
- Instrucción si, sino-si, sino, mientras, hacer-mientras, repetir, para
- Sentencia incluir
- Tabla de símbolos
- Comprobación de tipos para todas las operaciones
- Funciones nativas con archivos **.der**
- Métodos con y sin parámetros.
- Funcion imprimir en consola
- Árbol AST

La entrega de la fase 1 será el miércoles 26 de junio de 2019 antes de las 23:59.

Fase 2:

- Para la fase 2 se debe entregar todo lo restante como la interfaz gráfica completa la ejecución de todas las sentencias de control, el uso de todos métodos nativos etc.

Nota:

Se debe crear un árbol AST para la ejecución del código de alto nivel.

7. Entregables

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

Fecha de Entrega y Calificación: Sábado 13 de Julio de 2019 antes de las 06:00 AM.