Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Organización de Lenguajes y Compiladores 1
2do Semestre 2021
Catedráticos
Ing. Mario Bautista
Ing. Manuel Castillo
Ing. Kevin Lajpop
Tutores Académicos
Emely García, José Morán
Erick Lemus, René Corona
Sandra Jiménez

SYSCOMPILER

Proyecto 2

Contenido

1. Objetivos

1.1 Objetivos Generales

Aplicar los conocimientos sobre la fase de análisis léxico y sintáctico de un compilador para la realización de un intérprete sencillo, con las funcionalidades principales para que sea funcional.

1.2 Objetivos Específicos

- Reforzar los conocimientos de análisis léxico y sintáctico para la creación de un lenguaje de programación.
- 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.
- Aplicar conceptos de contenedores para generar aplicaciones livianas.
- Conocer más acerca de Docker y Docker-Compose
- Generar aplicaciones utilizando arquitecturas Cliente-Servidor.

2. Descripción General

El curso de Organización de Lenguajes y Compiladores 1, ha puesto en marcha un nuevo proyecto, requerido por la Escuela de Ciencias y Sistemas de la Facultad de Ingeniería, que consiste en crear un lenguaje de programación 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. Cabe destacar, que este lenguaje será utilizado para generar sus primeras prácticas de laboratorio del curso antes mencionado.

Por lo tanto, a usted, que es estudiante del curso de Compiladores 1, se le encomienda realizar el proyecto llamado SysCompiler, dado sus altos conocimientos en temas de análisis léxico, sintáctico y semántico.

3. Arquitectura General del proyecto

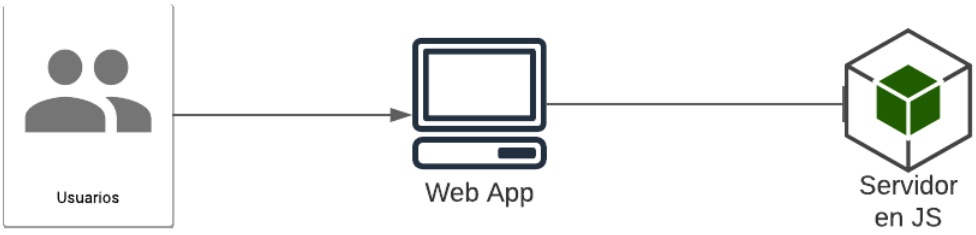
Hoy en día, se ha dado gran importancia al uso de tecnologías de contenedores, lo que otorga como ventajas; rapidez en el despliegue y facilidad de mantenimiento en un servidor.

Para el presente proyecto, se le solicita manejar 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.

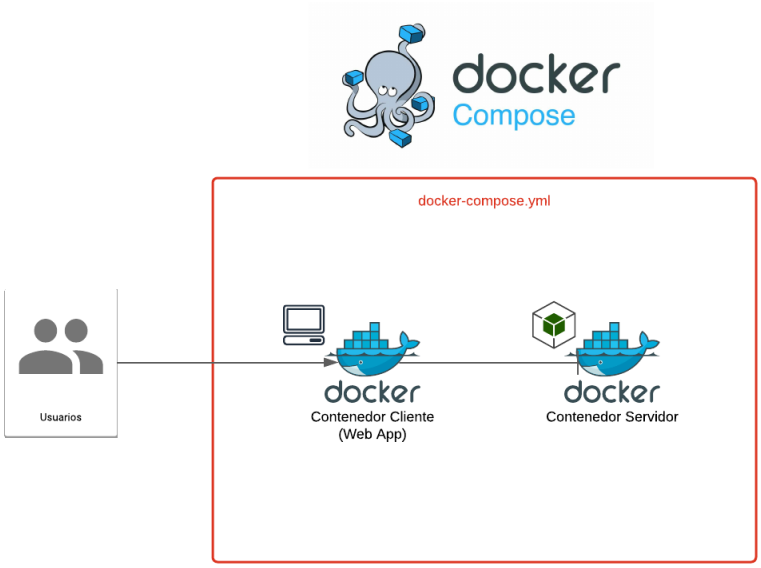
De la misma manera, se solicita el uso de contenedores de Docker (de manera no obligatoria), además del uso de docker-compose para administrar y desplegar fácilmente los contenedores a través de un archivo de especificaciones .YAML.

A continuación, se encuentra el diagrama de la arquitectura a implementar en un entorno de Docker.

Arquitectura Cliente-Servidor sin Docker



Arquitectura Cliente-Servidor con Docker

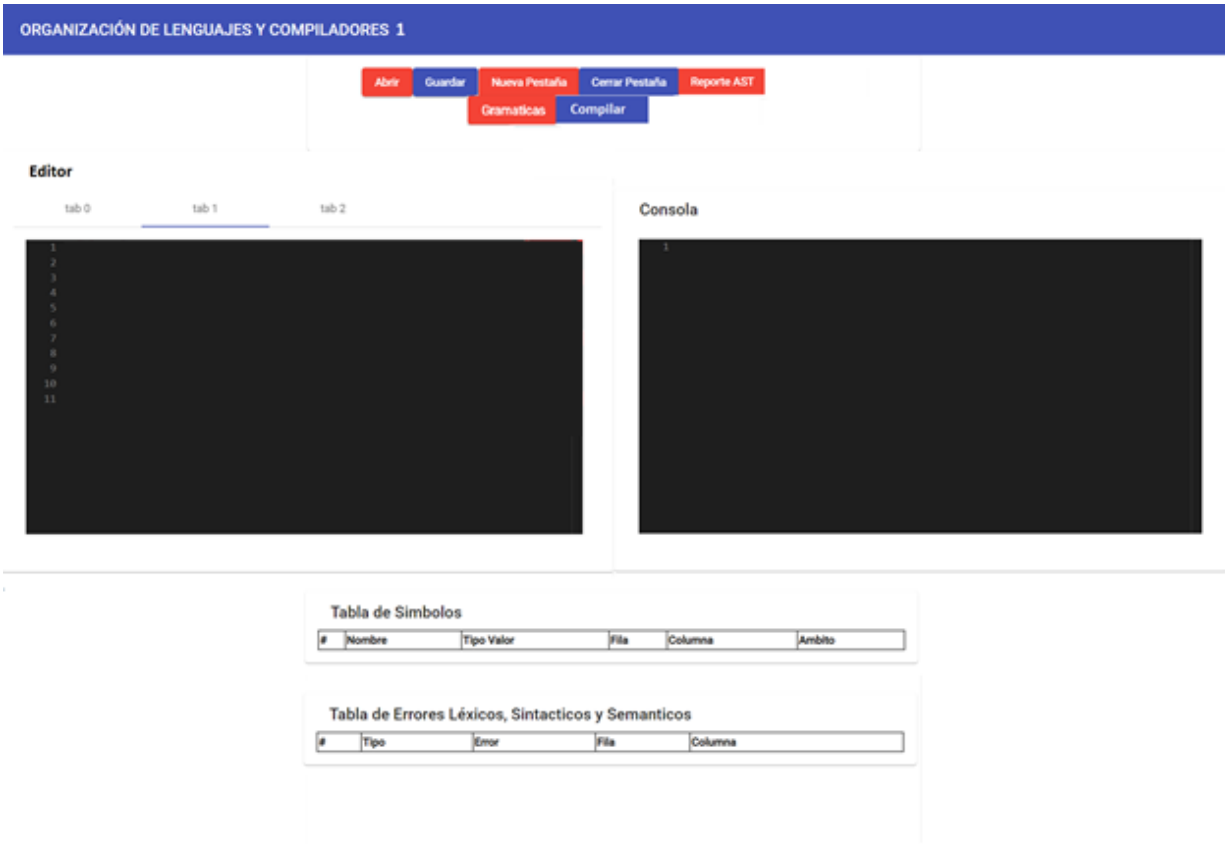


4. Entorno de Trabajo

4.1 Editor

El editor será parte del entorno de trabajo, cuya finalidad será proporcionar ciertas funcionalidades, características y herramientas que serán de utilidad al usuario. La función principal del editor será el ingreso del código fuente que será analizado. En este se podrán abrir diferentes archivos al mismo tiempo y deberá mostrar la línea actual. El editor de texto se tendrá que mostrar en el navegador. Queda a discreción del estudiante el diseño.

PROPUESTA DE INTERFAZ GRAFICA DE LA WEB APP



4.2 Funcionalidades

- **Crear archivos:** El editor deberá ser capaz de crear archivos en blanco.
- **Abrir archivos:** El editor deberá abrir archivos .sc
- **Guardar el archivo**: El editor deberá guardar el estado del archivo en el que se estará trabajando.
- **Eliminar pestaña:** permitirá cerrar la pestaña actual.

4.3 Características

- **Múltiples Pestañas:** se podrán crear nuevas pestañas con la finalidad de ver y abrir los archivos de prueba en la aplicación.

4.4 Herramientas

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

4.5 Reportes

- **Reporte de Errores:** Se mostrarán todos los errores encontrados al realizar el análisis léxico, sintáctico y semántico.
- **Generar Árbol AST (Árbol de Análisis Sintáctico):** se debe generar una imagen del árbol de análisis sintáctico que se genera al realizar los análisis.
- **Reporte de Tabla de Símbolos:** Se mostrarán todas las variables, métodos y funciones que han sido declarados dentro del flujo del programa.

4.6 Área de consola

En esta área se mostrarán los resultados, mensajes y todo lo que sea indicado dentro del lenguaje.

5. Descripción del Lenguaje

5.1 Case Insensitive

El lenguaje no distinguirá entre mayúsculas o minúsculas.

```
// Ejemplo

int a=0;
INt A=0;

//Debe dar error la declaración de “A” ya que la variable “a” ya existe previamente
//int es lo mismo que INt
```

Nota: al guardar variables, funciones, etc. en la tabla de símbolos, se recomienda almacenarlos todos en minúscula o mayúscula.

5.2 Comentarios

Los comentarios son una forma elegante de indicar que función tiene cierta sección del código que se ha escrito simplemente para dejar algún mensaje en específico. El lenguaje deberá soportar dos tipos de comentarios que son los siguientes:

4.4.1. Comentarios de una línea

Estos comentarios deberán comenzar con // y terminar con un salto de línea.

4.4.1. Comentarios de una línea

Estos comentarios deberán comenzar con /* y terminar con */.

EJEMPLOS DE COMENTARIOS

```
// Este es un comentario de una línea

/*
        Este es un comentario
        Multilínea
        Para este lenguaje
*/
```

5.3 Tipos de Datos

Los tipos de dato que soportará el lenguaje en concepto de un tipo de variable se definen a continuación:

TIPO	DEFINICION	DESCRIPCION	EJEMPLO	OBSERVACIONES	DEFAULT
Entero	Int	Este tipo de datos aceptará solamente números enteros.	1, 50, 100, 25552, etc.	Del -2147483648 al 2147483647	0
Doble	Double	Admite valores numéricos con decimales.	1.2, 50.23, 00.34, etc.	Se manejará cualquier cantidad de decimales	0.0
Booleano	Boolean	Admite valores que indican verdadero o falso.	True, false	Si se asigna un valor booleano a un entero se tomará como 1 o 0 respectivamente.	True
Caracter	Char	Tipo de dato que únicamente aceptará un único carácter, y estará delimitado por comillas simples. **' '**	'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, \".	‘\u0000’ (carácter 0)
Cadena	String	Es un grupo o conjunto de caracteres que pueden tener cualquier carácter, y este se encontrará delimitado por comillas dobles. **" "**	"cadena1", "-- ** cadena 1"	Se permitirá cualquier carácter entre las comillas dobles, incluyendo las secuencias de escape: \” comilla doble \\ barra invertida \n salto de linea \r retorno de carro \t tabulación	“” (string vacío)

5.4 Secuencias de Escape

Las secuencias de escape se utilizan para definir ciertos caracteres especiales dentro de cadenas de texto. Las secuencias de escape disponibles son las siguientes:

SECUENCIA	DESCRIPCION	EJEMPLO
\n	Salto de línea	“Hola\nMundo”
\\	Barra invertida	“C:\\miCarpeta\\Personal”
\“	Comilla doble	“\”Esto es una cadena\””
\t	Tabulación	“\tEsto es una tabulación”
\’	Comilla Simple	“\’Estas son comillas simples\’”

5.5 Operadores Aritméticos

5.5.1. Suma

Es la operación aritmética que consiste en realizar la suma entre dos o más valores. El símbolo a utilizar es el signo más **+**.

Observaciones:

- Al sumar dos datos numéricos (entero o doble), el resultado debes ser numérico.

Especificaciones de la operación suma

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

+	Entero	Doble	Boolean	Caracter	Cadena
Entero	Entero	Doble	Entero	Entero	Cadena
Doble	Doble	Doble	Doble	Double	Cadena
Boolean	Entero	Doble			Cadena
Caracter	Entero	Doble		Cadena	Cadena
Cadena	Cadena	Cadena	Cadena	Cadena	Cadena

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico.

5.5.2. Resta

Es la operación aritmética que consiste en realizar la resta entre dos o más valores. El símbolo por utilizar es el signo menos **-**.

Observaciones:

- Al restar dos datos numéricos (entero o doble), el resultado debes ser numérico.

Especificaciones de la operación resta

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

-	Entero	Doble	Boolean	Caracter	Cadena
Entero	Entero	Doble	Entero	Entero	
Doble	Doble	Doble	Doble	Doble	
Boolean	Entero	Doble			
Caracter	Entero	Doble			
Cadena					

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico. El carácter tomara su valor en ascii, ejemplo: 'A' es igual a 65.

5.5.3. Multiplicación

Operación aritmética que consiste en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). El sino para representar la operación es el asterisco*****.

Observaciones:

- Al multiplicar dos datos numéricos (entero o doble), el resultado debes ser numérico.

Especificaciones de la operación multiplicación
A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

*	Entero	Doble	Boolean	Caracter	Cadena
Entero	Entero	Doble		Entero	
Doble	Doble	Doble		Doble	
Boolean					
Caracter	Entero	Doble			
Cadena					

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico. El carácter tomara su valor en ascii, ejemplo: 'A' es igual a 65.

5.5.4. 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 ***/***.

Observaciones:

- Al dividir dos datos numéricos (entero o doble), el resultado debes ser numérico.

Especificaciones de la operación división
A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

/	Entero	Doble	Boolean	Caracter	Cadena
Entero	Doble	Doble		Doble	
Doble	Doble	Doble		Doble	
Boolean					
Caracter	Doble	Doble			
Cadena					

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico. El carácter tomara su valor en ascii, ejemplo: 'A' es igual a 65.

5.5.5. Potencia

Es una operación aritmética de la forma a^b donde **a** es el valor de la base y **b** es el valor del exponente que nos indicará cuantas veces queremos multiplicar el mismo número. Por ejemplo 5^3, **a=5** y **b=3** tendríamos que multiplicar 3 veces 5 para obtener el resultado final; 5x5x5 que da como resultado 125. Para realizar la operación se utilizará el signo **^**.

Observaciones:

- Al potenciar dos datos numéricos (entero o doble), el resultado debes ser numérico.

Especificaciones de la operación potencia

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

^	Entero	Doble	Boolean	Caracter	Cadena
Entero	Entero	Doble			
Doble	Doble	Doble			
Boolean					
Caracter					
Cadena					

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico.

5.5.6. Módulo

Es una operación aritmética que obtiene el resto de la división de un numero entre otro. El signo a utilizar es el porcentaje **%.**

Observaciones:

- Al obtener el módulo de un numero el resultado debe ser numérico.

Especificaciones de la operación potencia

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

%	Entero	Doble	Boolean	Caracter	Cadena
Entero	Doble	Doble			
Doble	Doble	Doble			
Boolean					
Caracter					
Cadena					

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico.

5.5.7. Negación Unaria

Es una operación que niega el valor de un número, es decir que devuelve el contrario del valor original, por ejemplo

Observaciones:

- La negación de un tipo entero o decimal debe generar como resultado su mismo tipo.

Especificaciones de la operación negación

A continuación, se especifica en una tabla los resultados que se deberán obtener con esta operación.

-num	Resultado
Entero	Entero
Doble	Doble
Boolean	
Caracter	
Cadena	

Nota: Cualquier otra combinación no especificada en esta tabla es inválida y será considerado

un error de tipo semántico.

5.6 Operadores Relacionales

Son los símbolos que tienen como finalidad comparar expresiones, dando como resultado valores booleanos. A continuación, se definen los símbolos que serán aceptados dentro del lenguaje:

Observaciones:

- Se pueden realizar operaciones relacionales entre: **entero-entero, entero-doble, entero-caracter, doble-entero, doble-caracter, caracter-entero, caracter-doble, caracter-carácter** y cualquier otra operación relacional entre entero, doble y carácter.
- Operaciones como cadena-carácter, es error semántico, a menos que se utilice toString en el carácter.
- Operaciones relacionales entre booleanos es válido.

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 25.5 == 20
!=	Diferenciación: Compara ambos lados y verifica si son distintos. - Iguales= False - No iguales= True	1 != 2, var1 != var2 25.5 != 20 50 != 'F' "hola" != "hola"
<	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) 25.5 < 20 25.5 < 20 50 < 'F'
<=	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 25.5 <= 20 25.5 <= 20 50 <= 'F'
>	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 25.5 > 20 25.5 > 20 50 > 'F'
>=	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 25.5 >= 20 25.5 >= 20 50 >= 'F'

5.7 Operador Ternario

El operador ternario es un operador que hace uso de 3 operandos para simplificar la instrucción 'if' por lo que a menudo este operador se le considera como un atajo para la instrucción 'if'. El primer operando del operador ternario corresponde a la condición que debe de cumplir una expresión para que el operador retorne como valor el resultado de la expresión segundo operando del operador y en caso de no cumplir con la expresión el operador debe de retornar el valor de la expresión del tercer

operando del operador.

```
<CONDICION> '?' <EXPRESION> ':' <EXPRESION>

//Ejemplo del uso del operador ternario

int edad = 18;
boolean banderaedad = false;
banderaedad = edad > 17 ? true : false;
```

5.8 Operadores Lógicos

Son los símbolos que tienen como finalidad comparar expresiones a nivel lógico (verdadero o falso). A continuación, se definen los símbolos que serán aceptados dentro del lenguaje:

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)|| 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

5.9 Signos de Agrupación

Los signos de agrupación serán utilizados para agrupar operaciones aritméticas, lógicas o relacionales. Los símbolos de agrupación están dados por **(** y **)**.

5.10 Precedencia de Operaciones

La precedencia de operadores nos indica la importancia en que una operación debe realizarse por encima del resto. A continuación, se define la misma.

NIVEL	OPERADOR	ASOCIATIVIDAD
0	-	Derecha
1	^	No asociativa
2	/, *	Izquierda
3	+, -	Izquierda
4	==, !=, <, <=, >, >=	Izquierda
5	!	Derecha
6	&&	Izquierda
7	||	Izquierda

NOTA: el nivel 0 es el nivel de mayor importancia.

5.11 Caracteres de finalización y encapsulamiento de sentencias

El lenguaje se verá restringido por dos reglas que ayudan a finalizar una instrucción y encapsular sentencias:

- **Finalización de instrucciones:** para finalizar una instrucciones se utilizara el signo **;**.
- **Encapsular sentencias:** para encapsular sentencias dadas por los ciclos, métodos, funciones, etc, se utilizará los signos **{** y **}**.

```
//Ejemplo de finalización de instrucción
int edad = 18;
//Ejemplo de encapsulamiento de sentencias
If(i==1){
    Int a=15;
    WriteLine("soy el numero "+a)
}
```

5.12 Declaración y asignación de variables

Una variable deberá de ser declarada antes de poder ser utilizada. Todas las variables tendrán un tipo de dato y un nombre de identificador. Las variables podrán ser declaradas global y localmente.

Durante la declaración de variables también se tendrá la opción de poder crear múltiples variables al mismo tiempo, al crear múltiples variables al mismo tiempo se tendrá la opción de crear todas las variables con un mismo valor, para ello se realizará una asignación al final del listado de las variables, en caso de no indicar esta asignación se dejará el valor por defecto para cada variable.

```
<TIPO> identificador;
<TIPO> id1, id2, id3, id4;
<TIPO> identificador = <EXPRESION>;
<TIPO> id1, id2, id3, id4 = <EXPRESION>;
//Ejemplos
int numero;
int var1, var2, var3;
string cadena = "hola";
char var_1 = 'a';
boolean verdadero;
boolean flag1, flag2, flag3 = true;
char ch1, ch2, ch3 = 'R';
```

Las variables no pueden cambiar de tipo de dato, se deben mantener con el tipo declarado inicialmente, por lo que se debe de validar que el tipo de la variable y el valor sean compatibles.

```
identificador = <EXPRESION>;
//Ejemplos
numero = 4;
verdadero = true;
cadena = “valor = “ + valor;
```

5.13 Casteos

Los casteos son una forma de indicar al lenguaje que convierta un tipo de dato en otro, por lo que, si queremos cambiar un valor a otro tipo, es la forma adecuada de hacerlo. Para hacer esto, se colocará la palabra reservada del tipo de dato destino ente paréntesis seguido de una expresión.

‘(’ <TIPO> ‘)’ <EXPRESION>

El lenguaje aceptará los siguientes casteos:

- **Int a double**
- **Double a Int**
- **Int a String**
- **Int a Char**
- **Double a String**
- **Char a int**
- **Char a double**

```
‘(’<TIPO>’)’ <EXPRESION>
 //Ejemplos
int edad = (int) 18.6; //toma el valor entero de 18
char letra = (char) 70; //tomar el valor ‘F’ ya que el 70 en ascii es F
double numero = (double) 16; //toma el valor 16.0
```

Nota: la conversión de un tipo a string se explica en la sección 5.25.5 con la función. toString

5.14 Incremento y Decremento

Los incrementos y decrementos nos ayudan a realizar la suma o resta continua de un valor de uno en uno, es decir si incrementamos una variable, se incrementará de uno en uno, mientras que, si realizamos un decremento, hará la operación contraria.

```
<EXPRESION>’+’’+’’;’
<EXPRESION> ‘-‘‘-’’;’
 //Ejemplos
int edad = 18;
edad++; //tiene el valor de 19
edad--; //tiene el valor 18

int anio=2020;
anio = 1 + anio++; //obtiene el valor de 2022
anio = anio--;   //obtiene el valor de 2021
```

5.15 Estructuras de Datos

Las estructuras de datos nos sirven para almacenar cualquier valor de un solo tipo dentro de la misma estructura, en el lenguaje se tiene dos tipos que son: listas y vectores.
A continuación, se definen las estructuras:

5.15.1. Vectores

Los vectores son una estructura de datos de tamaño fijo que pueden almacenar valores de forma limitada, y los valores que pueden almacenar son de un único tipo; int, double, boolean, char o string. **El lenguaje permitirá únicamente el uso de arreglos de una dimensión.**

Observaciones:

- La posición de cada vector será N-1. Por ejemplo, si deseo acceder al primer valor de un vector debo acceder como miVector[0].

5.15.1.1 Declaración de Vectores

Al momento de declarar un vector, tenemos dos tipos que son:

- **Declaración tipo 1:** En esta declaración, se indica por medio de una expresión numérica del tamaño que se desea el vector, además toma los valores por default para cada tipo.
- **Declaración tipo 2:** En esta declaración, se indica por medio de una lista de valores separados por coma, los valores que tendrá el vector, en este caso el tamaño del vector será el de la misma cantidad de valores de la lista.

```
//DECLARACION TIPO 1
<TIPO> <ID>‘[’ ‘]’ = new <TIPO>’[’ <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

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

5.15.1.2 Acceso a vectores

Para acceder al valor de una posición de un vector, se colocará el nombre del vector seguido de [EXPRESION].

```
<ID> ’[’ EXPRESION ’]’
//Ejemplo de acceso
string vector2[ ] = {"hola", "Mundo"}; //creamos un vector de 2 posiciones de tipo string
string valorPosicion = vector2[0]; //posición 0, valorPosicion = “hola”
```

5.15.1.3 Modificación de Vectores

Para modificar el valor de una posición de un vector, se debe colocar el nombre del vector seguido de ‘[’ EXPRESION ‘]’ = EXPRESION

Observaciones:

- A una posición de un vector se le puede asignar el valor de otra posición de otro vector o del mismo vector.
- A una posición de un vector se le puede asignar el valor de una posición de una lista.

```
<ID> ‘[’ EXPRESION ‘]’ = EXPRESION‘;’

string vector2[ ] = {"hola", "Mundo"}; //vector de 2 posiciones, con "Hola" y "Mundo"
int vectorNumero[ ] = {2020,2021,2022};
vector2[0] = “OLC1 ”;
vector2[1] = “1er Semestre ”+vectorNumero[1];

/*
        RESULTADO
vector2[0]= “OLC1 ”
vector2[1]= “1er Semestre 2021”
*/
```

5.15.2. Listas Dinámicas

Las listas son una estructura de datos que pueden crecer de forma iterativa y pueden almacenar hasta N elementos de un solo tipo; int, double, boolean, char o string.

Observaciones:

- La posición de cada lista será N-1. Por ejemplo si deseo acceder al primer valor de una lista debo acceder como getValue(miLista, 0) **ACCESO A LISTAS EN SECCIÓN 5.15.2.3**.

5.15.2.1 Declaración de Listas

Al momento de declarar una lista, tendremos un único tipo de declaración.

```
//DECLARACION
‘DynamicList’‘<’ <TIPO> ’>’ <ID> = new ‘DynamicList’‘<’ <TIPO> ’>’ ‘;’


//Ejemplo de declaración
DynamicList<int> lista1 = new DynamicList<int>; //se crea una lista vacía de tipo entero
DynamicList<char> lista1 = new DynamicList<char>; //se crea una lista vacía de tipo
char
```

5.15.2.2 Agregar valor a una lista

Para agregar un valor a una lista, se utilizara la palabra reservada **append** seguido de **‘(’ NOMBRE_LISTA ’,’ EXPRESION ‘)’**.

```
        'append' '('<ID>','<EXPRESION>')' ';'
//Ejemplo de agregar valor a una lista
DynamicList <int> lista2 =new DynamicList <int>; //creamos un lista de tipo int SE ENCUENTRA VACÍA
append(lista2,5);
append(lista2,4);
append(lista2,100);
/*
        RESULTADO
getValue(lista2, 0) = 5
getValue(lista2, 1) = 4
getValue(lista2, 2) = 100
*/
```

5.15.2.3 Acceso a Listas

Para acceder al valor de una posición de una lista, se colocará la palabra reservada **GETVALUE** seguido de '(' ID ',' EXPRESION ')'.

```
        'getValue' '(' <ID> ',' EXPRESION ')'
//Ejemplo de acceso
DynamicList <int> lista2 =new DynamicList <int>; //creamos un lista de tipo int
//Agregamos valores a la lista
append (lista2,5);
append (lista2,4);
append (lista2,100);
//accesamos a un valor de la lista
int valor = getValue(lista2,1); // valor = 4
```

5.15.2.4 Modificación de Vectores

Para modificar el valor de una posición de una lista, se colocará la palabra reservada **SETVALUE** seguido de '(' ID ',' EXPRESION ',' EXPRESION ')'.

Observaciones:

- A una posición de una lista se le puede asignar el valor de otra posición de otra lista o de la misma lista.
- A una posición de una lista se le puede asignar el valor de una posición de un vector.
- La primera expresión hace referencia a la posición, mientras que la segunda expresión al nuevo valor en la posición de la lista

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

DynamicList <string> listaS = new DynamicList <string>; //lista de strings
int[ ] vectorNumero = {2020,2021,2022};
```

```
//agregamos valores a la lista
append(listaS ,“Hola ”);
append(listaS ,“Mundo”);
/*
        Actualmente
getValue(listaS, 0) = “Hola ”
getValue(listaS, 1)= “Mundo”
*/
setValue(listaS,0,“OLC1 ”);
setValue(listaS,1,“1er Semestre ”+vectorNumero[1]);

/*
        RESULTADO
getValue(listaS, 0)= “OLC1 ”
getValue(listaS, 1)= “1er Semestre 2021”
*/
```

5.16 Sentencias de control

Estas sentencias modifican el flujo del programa introduciendo condicionales. Las sentencias de control para el programa son el IF y el SWITCH.

OBSERVACIONES:

- También, entre las sentencias pueden tener Ifs anidados.

5.16.1. if

La sentencia if ejecuta las instrucciones sólo si se cumple una condición. Si la condición es falsa, se omiten las sentencias dentro de la sentencia.

5.16.1.1. if

```
‘if’ ‘(’ [<EXPRESION>] ‘)’ ‘{‘
        [<INSTRUCCIONES>]
 ‘}’
| ‘if’ ‘(’ [<EXPRESION>] ‘)’ ‘{‘
        [<INSTRUCCIONES>]
  ‘}’ ‘else’ ‘{‘
        [<INSTRUCCIONES>]
 ‘}’
| ‘if’ ‘(’ [<EXPRESION>]  ‘)’ ‘{‘
        [<INSTRUCCIONES>]
  ‘}’ ‘else’ [<IF>]


//Ejemplo de cómo se implementar un ciclo if

if (x <50)
 {
    WriteLine(“Menor que 50”);
    //Más sentencias
}
```

5.16.1.2. if else

```
//Ejemplo de cómo se implementar un ciclo if-else

if (x <50)
 {
    WriteLine(“Menor que 50”);
    //Más sentencias
}
else
{
    WriteLine(“Mayor que 100”);
    //Más sentencias
}
```

5.16.1.3. else

```
//Ejemplo de cómo se implementar un ciclo else if

if (x > 50)
 {
    WriteLine(“Mayor que 50”);
    //Más sentencias
}
else if (x <= 50 && x > 0)
{
    WriteLine (“Menor que 50”);
    if (x > 25)
    {
        WriteLine(“Número mayor que 25”);
        //Más sentencias
    }
    else
    {
        WriteLine(“Número menor que 25”);
        //Más sentencias
    }
    //Más sentencias
}
else
{
    WriteLine(“Número negativo”);
    //Más sentencias
}
```

5.16.2. Switch Case

Switch case es una estructura utilizada para agilizar la toma de decisiones múltiples, trabaja de la misma manera que lo harían sucesivos if.

5.16.2.1. Switch

Estructura principal del switch, donde se indica la expresión a evaluar.

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

5.16.2.2. Case

Estructura que contiene las diversas opciones a evaluar con la expresión establecida en el switch.

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

5.16.2.3. Default

Estructura que contiene las sentencias si en dado caso no haya salido del switch por medio de una sentencia **break**.

```
'default' ':'
        [<INSTRUCCIONES>]
```

```
// EJEMPLO DE SWITCH
int edad = 18;
switch( edad ) {
    Case 10:
            WriteLine("Tengo 10 anios.");
            // mas sentencias
            Break;
    Case 18:
            WriteLine("Tengo 18 anios.");
            // mas sentencias
    Case 25:
            WriteLine("Tengo 25 anios.");
            // mas sentencias
            Break;
    Default:
            WriteLine("No se que edad tengo. :(");
            // mas sentencias
            Break;
 }
/* Salida esperada
Tengo 18 anios.
No se que edad tengo. :(
*/
```

OBSERVACIONES:

- Si la cláusula “case” no posee ninguna sentencia “break”, al terminar todas las sentencias del case ingresado, el lenguaje seguirá evaluando las demás opciones.

5.17 Sentencias cíclicas

Los ciclos o bucles, son una secuencia de instrucciones de código que se ejecutan una vez tras otra mientras la condición, que se ha asignado para que pueda ejecutarse, sea verdadera. En el lenguaje actual, se podrán realizar 3 sentencias cíclicas que se describen a continuación.

OBSERVACIONES:

- Es importante destacar que pueden tener ciclos anidados entre las sentencias a ejecutar.
- También, entre las sentencias pueden tener ciclos diferentes anidados.

5.17.1. 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>]
‘}’

//Ejemplo de cómo se implementar un ciclo while

while (x<100){
        if (x > 50)
        {
                WriteLine(“Mayor que 50”);
                //Más sentencias
        }
        else
        {
                WriteLine(“Menor que 100”);
                //Más sentencias
        }
        X++;
        //Más sentencias

}
```

5.17.2. 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.

OBSERVACIONES:

- Para la actualización de la variable del ciclo for, se puede utilizar:

- **Incremento | Decremento:** i++ | i--
- **Asignación:** como i=i+1, i=i-1, i=5, i=x, etc, es decir cualquier tipo de asignación.

- Dentro pueden venir N instrucciones

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

                  //Ejemplo 1: declaración dentro del for con incremento
                   for ( int i=0; i<3;i++ ){
                            WriteLine("i="+i)
                            //más sentencias
                   }
                  /*RESULTADO
                     i=0
                     i=1
                     i=2
                   */

//Ejemplo 2: asignación de variable previamente declarada y decremento por asignación
                   for ( i=5; i>2;i=i-1 ){
                            WriteLine("i="+i)
                            //más sentencias
                   }
                  /*RESULTADO
                     i=5
                     i=4
                     i=3
                   */
```

5.17.3. Do-While

El ciclo o bucle 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>] ')' ';'

                   //Ejemplo de cómo se implementar un ciclo do-while
                  Int a=5;
                  Do{
                        If (a>=1 && a <3){
                             WriteLine(true)
                        }
                        Else{
                             WriteLine(false)
                        }
```

```
        a--;
    } while (a>0);
    /*RESULTADO
      false
      false
      false
      true
      true
    */
```

NOTA: Dentro pueden venir N instrucciones

5.18 Sentencias de transferencia

Las sentencias de transferencia nos permiten manipular el comportamiento de los bucles, ya sea para detenerlo o para saltarse algunas iteraciones. El lenguaje soporta las siguientes sentencias:

5.18.1. Break

La sentencia break hace que se salga del ciclo inmediatamente, es decir que el código que se encuentre después del break en la misma iteración no se ejecutara y este se saldrá del ciclo.

```
break;
//Ejemplo en un ciclo for
for(int i = 0; i < 9; i++){
    if(i==5){
        WriteLine("Me salgo del ciclo en el numero " + i);
        break;
    }
    WriteLine(i);
}
```

5.18.2. Continue

La sentencia continue puede detener la ejecución de la iteración actual y saltar a la siguiente. La sentencia continue siempre debe de estar dentro de un ciclo, de lo contrario será un error.

```
continue;
//Ejemplo en un ciclo for
for(int i = 0; i < 9; i++){
    if(i==5){
        WriteLine("Me salte el numero " + i);
        continue;
    }
    WriteLine(i);
}
```

5.18.3. Return

La sentencia return finaliza la ejecución de un método o función y puede especificar un valor para ser devuelto a quien llama a la función.

```
return;
return <EXPRESION>;
//Ejemplos
//--> Dentro de un metodo
void mi_metodo(){
    int i;
    for(i = 0; i < 9; i++){
        if(i==5){
            return; //se detiene
        }
        WriteLine(i);
    }
}
//--> Dentro de una función
int sumar(int n1, int n2){
    int n3;
    n3 = n1+n2;
    return n3;      //retorno el valor
}
```

5.19 Funciones

Una función es una subrutina de código que se identifica con un nombre, tipo y un conjunto de parámetros. Para este lenguaje las funciones serán declaradas definiendo primero su tipo, luego un identificador para la función, seguido de una lista de parámetros dentro de paréntesis (esta lista de parámetros puede estar vacía en el caso de que la función no utilice parámetros).

Cada parámetro debe estar compuesto por su tipo seguido de un identificador, para el caso de que sean varios parámetros se debe utilizar comas para separar cada parámetro y en el caso de que no se usen parámetros no se deberá incluir nada dentro de los paréntesis. Luego de definir la función y sus parámetros se declara el cuerpo de la función, el cual es un conjunto de instrucciones delimitadas por llaves {}.

Para las funciones es obligatorio que las mismas posean un valor de retorno que coincida con el tipo con el que se declaró la función, en caso de que no sea el mismo tipo o de que no venga un retorno dentro del cuerpo de la función debería lanzarse un error de tipo semántico.

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

//Ejemplo de declaración de una función de enteros
double conversion(double pies, string tipo){
```

```
        if (tipo == “metro”)
        {
                return pies/3.281;
        }
        else
        {
                return -1;
        }
}
```

Cabe a destacar que **no habrá sobrecarga de funciones y métodos** dentro de este lenguaje por lo que solo puede existir una función o método con el id declarado por lo que si se crea otra función o método con un id previamente utilizado esto debe de generar un error de tipo semántico.

5.20 Métodos

Un método también es una subrutina de código que se identifica con un nombre, tipo y un conjunto de parámetros, aunque a diferencia de las funciones estas subrutinas no deben de retornar un valor. Para este lenguaje los métodos serán declarados haciendo uso de la palabra reservada ‘void’ al inicio, luego se indicará el identificador del método, seguido de una lista de parámetros dentro de paréntesis (esta lista de parámetros puede estar vacía en el caso de que la función no utilice parámetros).

Cada parámetro debe estar compuesto por su tipo seguido de un identificador, para el caso de que sean varios parámetros se debe utilizar comas para separar cada parámetro y en el caso de que no se usen parámetros no se deberá incluir nada dentro de los paréntesis. Luego de definir el método y sus parámetros se declara el cuerpo del método, el cual es un conjunto de instrucciones delimitadas por llaves {}.

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


//Ejemplo de declaración de un método

void holamundo(){
        WriteLine(“Hola mundo”);
}
```

Cabe a destacar que **no habrá sobrecarga de funciones y métodos** dentro de este lenguaje por lo que solo puede existir una función o método con el id declarado por lo que si se crea otra función o método con un id previamente utilizado esto debe de generar un error de tipo semántico.

5.21 Llamadas

La llamada a una función específica la relación entre los parámetros reales y los formales y ejecuta la función. Los parámetros se asocian normalmente por posición, aunque, opcionalmente, también se pueden asociar por nombre. Si la función tiene parámetros formales por omisión, no es necesario asociarles un parámetro real.

La llamada a una función devuelve un resultado que ha de ser recogido, bien asignándolo a una variable del tipo adecuado, bien integrándolo en una expresión.

La sintaxis de las llamadas de los métodos y funciones serán la misma.

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

PARAMETROS_LLAMADA -> [<PARAMETROS_LLAMADA>] ',' [<ID>]
                                  | [<ID>]
                  //Ejemplo de llamada de un método

                  WriteLine("Ejemplo de llamada a método");
                  holamundo();
                  /* Salida esperada
                     Ejemplo de llamada a método
                     Hola Mundo
                  */
                  //Ejemplo de llamada de una función

                  WriteLine("Ejemplo de llamada a función");
                  Int num = suma(6,5); // a = 11
                  WriteLine("El valor de a es: " + a);
                  /* Salida esperada
                     Ejemplo de llamada a función
                     Aquí puede venir cualquier sentencia :D
                     El valor de a es: 11
                  */

                  Int suma(int num1, int num2)
                  {
                      WriteLine("Aquí puede venir cualquier sentencia :D");
                      return num1 + num2;
                      WriteLine("Aquí pueden venir más sentencias, pero no se ejecutarán
por la sentencia RETURN D:"); //WriteLine en una línea
                  }
```

OBSERVACIONES:

- Al momento de ejecutar cualquier llamada, no se diferenciarán entre métodos y funciones, por lo tanto, podrá venir una función que retorne un valor como un método, pero la expresión retornada no se asignará a ninguna variable.

- Se podrán llamar métodos y funciones antes que se encuentren declaradas, para ello se recomienda realizar 2 pasadas del AST generado: La primera para almacenar todas las funciones, y la segunda para las variables globales y la función start with (5.26).
- Para la llamada a métodos como una instrucción se deberá de agregar el punto y coma (;) como una instrucción.

5.22 Función WriteLine

Esta función nos permite imprimir expresiones con valores únicamente de tipo entero, doble, booleano, cadena y carácter.

```
'WriteLine' '(' <EXPRESION> ')';
//Ejemplo
WriteLine("Hola mundo!!");
WriteLine("Sale compi \n" + valor);
WriteLine(suma(2,2));
```

5.23 Función toLower

Esta función recibe como parámetro una expresión de tipo cadena y retorna una nueva cadena con todas las letras minúsculas.

```
'toLower' '(' <EXPRESION> ')';
//Ejemplo
string cad_1 = toLower("hOla MunDo");  // cad_1 = "hola mundo"
string cad_2 = toLower("RESULTADO = " + 100);  // cad_2 = "resultado = 100"
```

5.24 Función toUpper

Esta función recibe como parámetro una expresión de tipo cadena retorna una nueva cadena con todas las letras mayúsculas.

```
'toUpper' '(' <EXPRESION> ')';
//Ejemplo
string cad_1 = toUpper("hOla MunDo");  // cad_1 = "HOLA MUNDO"
string cad_2 = toUpper("resultado = " + 100);  // cad_2 = "RESULTADO = 100"
```

5.25. Funciones nativas

5.25.1Length

Esta función recibe como parámetro un vector, una lista o una cadena y devuelve el tamaño de este.

```
'length' '(' <VALOR> ')' ';'

// En donde <VALOR> puede ser:
//   - lista
//   -vector
//   -cadena
```

```
//Ejemplo
DynamicList<int> lista2 =new DynamicList <int>;
string[ ] vector2 = {"hola", "Mundo"};
int tam_lista = length(lista2); // tam_lista = 0

int tam_vector = length(vector2); // tam_vector = 2

int tam_hola = length(tam_vector[0]); // tam_hola = 4
```

Nota: Si recibe como parámetro un tipo de dato no especificado, se considera un error semántico.

5.25.2. Truncate

Esta función recibe como parámetro un valor numérico. Permite eliminar los decimales de un número, retornando un entero.

```
'truncate' '('<VALOR>')' ';'

// Ejemplo

int nuevoValor = truncate(3.53); // nuevoValor = 3
int otroValor = truncate(10); // otroValor = 10

double decimal = 15.4654849;
int entero = truncate(decimal); // entero = 15
```

Nota: Si la función recibe un valor no numérico, se considera un error semántico.

5.25.3. Round

Esta función recibe como parámetro un valor numérico. Permite redondear los números decimales según las siguientes reglas:
Si el decimal es mayor o igual que 0.5, se aproxima al número superior
Si el decimal es menor que 0.5, se aproxima al número inferior

```
'round' '(' <VALOR>')' ';'

// Ejemplo

Double valor = round(5.8); //valor = 6
Double valor2 = round(5.4); //valor2 = 5
```

Nota: La función retorna un valor entero.

5.25.4. Typeof

Esta función retorna una cadena con el nombre del tipo de dato evaluado.

```
'typeof' '(' <VALOR> ')' ';'

//Ejemplo

DynamicList<int> lista2 =new DynamicList <int>;
String tipo = typeof(15); // tipo = "int"
String tipo2 = typeof(15.25); // tipo = "double"
String tipo3 = typeof(lista2); // tipo3 = "lista"
```

5.25.5. To String

Esta función permite convertir un valor de tipo numérico o booleano en texto.

```
'toString' '(' <VALOR> ')' ';'

//Ejemplo

String valor = toString(14); // valor = "14"
String valor2 = toString(true); // valor = "true"
```

Nota: Si recibe como parámetro un tipo de dato no especificado, se considera un error semántico.

5.25.6. toCharArray

Esta función permite convertir una cadena en un arreglo de caracteres.

```
'toCharArray' '(' <VALOR> ')' ';'

//Ejemplo

DynamicList<char> caracteres = toCharArray("Hola");
/*
caracteres [[0]] = "H"
caracteres [[1]] = "o"
caracteres [[2]] = "l"
caracteres [[3]] = "a"
*/
```

5.26. Start With

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

```
    'start' 'with' <ID> '(' ')' ';'
    'start' 'with' <ID> '(' <LISTAVALORES> ')' ';'
LISTAVALORES->LISTAVALORES ',' EXPRESION
                | EXPRESION
    //Ejemplo 1
    void funcion1(){
        WriteLine("hola");
    }

    start with funcion1();
    /*RESULTADO
        hola
    */

    //Ejemplo 2
    void funcion2(string mensaje){
        WriteLine(mensaje);
    }

    start with funcion2("hola soy un mensaje");
    /*RESULTADO
        Hola soy un mensaje
    */
```

6. Reportes

Los reportes son una parte fundamental de SysCompiler, ya que muestra de forma visual las herramientas utilizadas para realizar la ejecución del código.

A continuación, se muestran ejemplos de estos reportes. (Queda a discreción del estudiante el diseño de estos, solo se pide que sean totalmente legibles).

6.1 Tabla de Símbolos

Este reporte mostrará la tabla de símbolos después de la ejecución. Se deberán mostrar todas las variables, funciones y métodos declarados, así como su tipo y toda la información que se considere necesaria.

Identificador	Tipo	Tipo	Entorno	Línea	Columna
Factor1	Variable	Entero	Función multiplicar	15	4
Factor2	Variable	Decimal	Función multiplicar	16	7
Resultado	Variable	Decimal	Función multiplicar	17	7
MostrarMensaje	Método	Void	-	50	6
Multiplicar	Función	Decimal	-	14	10

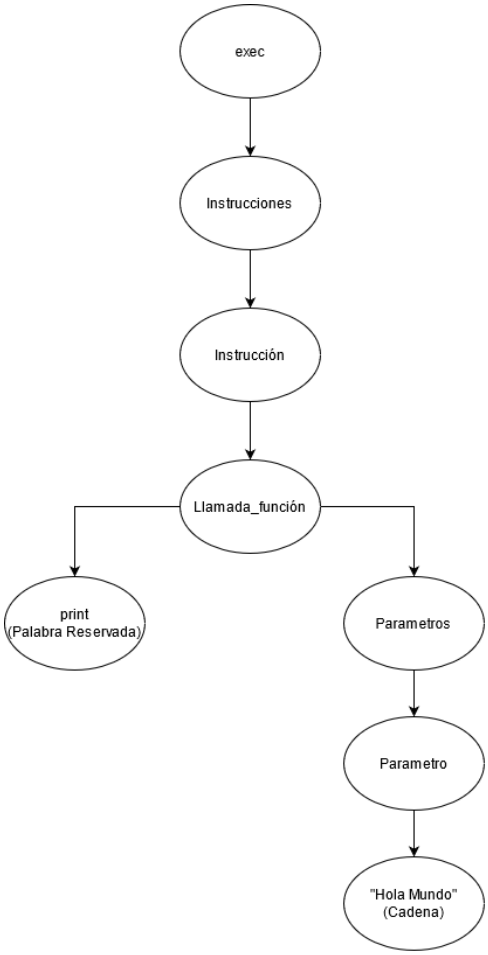
6.2 Tabla de Errores

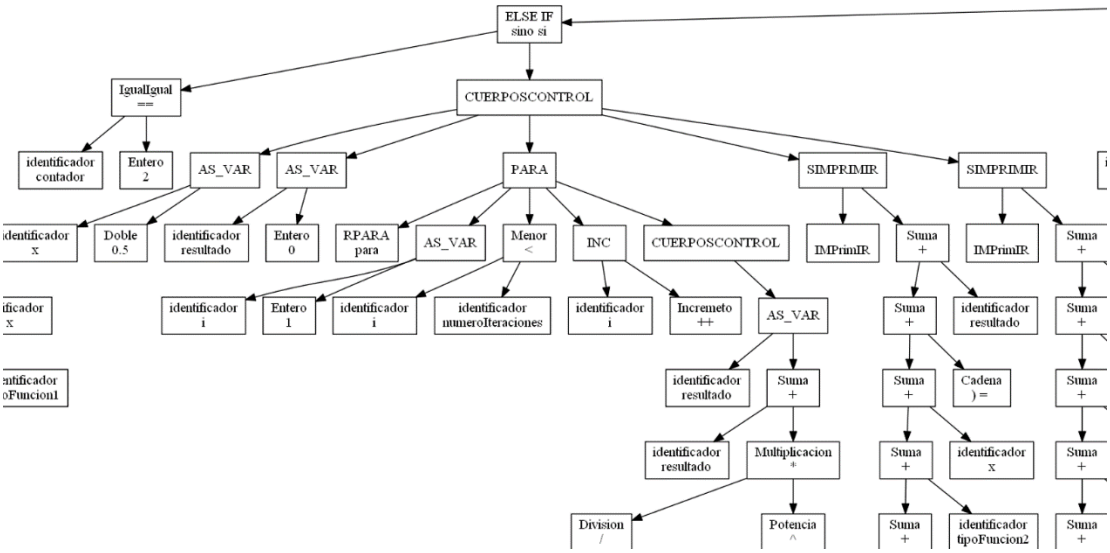
El reporte de errores debe contener la información suficiente para detectar y corregir errores en el código fuente.

#	Tipo de Error	Descripción	Línea	Columna
1	Léxico	El carácter “$” no pertenece al lenguaje.	7	32
2	Sintáctico	Encontrado Identificador “Ejemplo”, se esperaba Palabra Reservada “Valor”	150	12

6.3 AST

Este reporte muestra el árbol de sintaxis producido al analizar los archivos de entrada. Este debe de representarse como un grafo. Se deben mostrar los nodos que el estudiante considere necesarios para describir el flujo realizado para analizar e interpretar sus archivos de entrada.





Nota: Se sugiere a los estudiantes utilizar la herramienta Graphviz para graficar su AST.

6.4 Salidas en Consola

La consola es el área de salida del intérprete. Por medio de esta herramienta se podrán visualizar las salidas generadas por la función nativa “WriteLine”, así como los errores léxicos, sintácticos y semánticos.

```
> Este es un mensaje desde mi interprete de compi 1.
>
>
---> Error léxico: Símbolo "#" no reconocido en línea 10 y columna 7
---> Error Semántico: Se ha intentado asignar un entero a una variable booleana
```

7. Requerimientos Mínimos

Para que el estudiante tenga derecho a calificación, deberá cumplir con lo siguiente:

1. Poseer al menos 5 commits realizados en el repositorio donde se estuvo trabajando el proyecto.
2. Interfaz gráfica funcional.
3. Operaciones aritméticas, lógicas y relacionales
4. Declaración y asignación de variables (int, double, boolean, char, string)
5. Sentencias de control
6. Sentencias cíclicas
7. Métodos (con y sin parámetros)
8. Llamadas a métodos (con y sin parametros)
9. Función WriteLine
10. Start With
11. Reporte de AST
12. Reporte de Tabla de símbolos
13. Consola

Por último, se deberá entregar un documento explicando la gramática utilizada, así como las acciones semánticas que permiten la creación del árbol de análisis sintáctico; esto con el fin de verificar que el estudiante trabajó de forma individual.
Este documento debe contener las especificaciones de un lenguaje formal, tales como:

- Expresiones regulares
- Terminales
- No terminales
- Inicio de la gramática
- Descripción de las producciones

SI EL ESTUDIANTE NO CUMPLE CON ESTOS REQUERIMIENTOS, NO TENDRÁ DERECHO A CALIFICACIÓN.

8. Entregables

- Código Fuente del proyecto.
- Manuales de Usuario.
- Manual Técnico.
- Archivo de Gramática para la solución (El archivo debe de ser limpio, entendible y no debe ser una copia del archivo de jison).
- Link al repositorio privado de Github en donde se encuentra su proyecto (Se debe de agregar como colaborador al auxiliar que le califique).

9. Restricciones

1. Lenguajes de programación a usar: **Javascript/Typescript.**
2. Puede utilizar frameworks como Angular, React, Vuejs, etc para generar su entorno gráfico. Queda a discreción del estudiante.
3. Puede utilizar herramientas como Nodejs para Javascript.
4. Herramientas de análisis léxico y sintáctico: **Jison**
5. **El Proyecto es Individual.**
6. Para graficar se puede utilizar cualquier librería (Se recomienda Graphviz)
7. Copias completas/parciales de: código, gramáticas, etc. serán merecedoras de una nota de 0 puntos, los responsables serán reportados al catedrático de la sección y a la Escuela de Ciencias y Sistemas.
8. La calificación tendrá una duración de 30 minutos, acorde al programa del laboratorio.
9. Se debe visualizar todo lo acontecido desde el navegador.

10. Fecha de Entrega

Domingo 31 de octubre de 2021.

La entrega será por medio de la plataforma UEDI, si existieran inconvenientes con la plataforma se utilizará Classroom PREVIAMENTE INDICADO CON SU RESPECTIVO AUXILIAR.

Entregas fuera de la fecha indicada, no se calificarán.

SE LE CALIFICARA DEL COMMIT REALIZADO HASTA ESTA FECHA.