Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Organización de Lenguajes y Compiladores
Segundo Semestre 2022
Catedráticos
Ing. Mario Bautista, Ing. Manuel Castillo, Ing. Kevin Lajpop
Tutores Académicos
Moisés González, Fernando Guerra

MFMScript
Proyecto 2

Objetivos

Objetivos Generales

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

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.
- Generar aplicaciones utilizando arquitecturas Cliente-Servidor.

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.

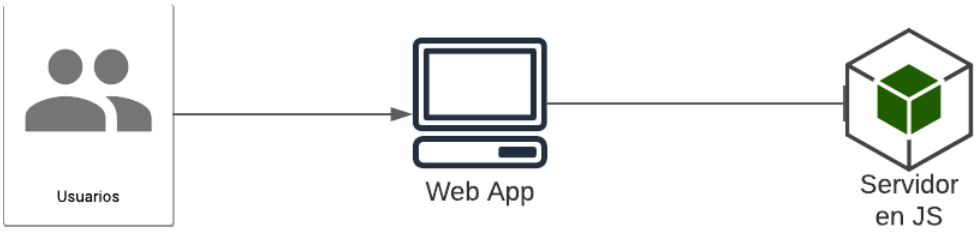
Por lo tanto, a usted, que es estudiante del curso de Compiladores 1, se le encomienda realizar el proyecto llamado MFMScript, dado sus altos conocimientos en temas de análisis léxico, sintáctico y semántico.

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 propone 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.

Arquitectura del proyecto a implementar:



Entorno de Trabajo

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.

Funcionalidades
- Crear archivos: El editor deberá ser capaz de crear archivos en blanco.
- Abrir archivos: El editor deberá abrir archivos [.olc]
- Guardar el archivo: El editor deberá guardar el estado del archivo en el que se estará trabajando.

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.

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

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

Descripción del Lenguaje

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.

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:

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

Comentarios de una línea
Estos comentarios deberán comenzar con /* y terminar con */.

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

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

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

TIPO	DEFINICIÓN	DESCRIPCIÓN	EJEMPLO	OBSERVACIONES	DEFAULT
Entero	Int	Este tipo de datos aceptará solamente números enteros.	1, 50, 100, 25552, etc.	Del -2147483648 al 2147483647	0
Decimal	Double	Admite valores numéricos con decimales.	1.2, 50.23, 00.34, etc.	Se maneja cualquier cantidad de decimales.	0.0

Logico	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
Carácter	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, \".	'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 línea \r retorno de carro \t tabulación	"" (string vacío)

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	DESCRIPCIÓN	EJEMPLO
\n	Salto de línea	"Hola\nMundo"
\\	Barra invertida	"C:\\usuario\\Personal"
\"	Comilla doble	"\"Esto es una cadena doble\""
\t	Tabulación	"\tAqui hay una tabulacion"
\'	Comilla simple	"\'Son comillas simples\'"

Operadores Aritméticos

Suma
Es la operación aritmética que consiste en realizar la suma entre dos o más valores. El símbolo para 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	Decimal	Boolean	Caracter	Cadena
Entero	Entero	Decimal	Entero	Entero	Cadena
Decimal	Decimal	Decimal	Decimal	Decimal	Cadena
Boolean	Entero	Decimal			Cadena
Caracter	Entero	Decimal		Cadena	Cadena
Cadena	Cadena	Cadena	Cadena	Cadena	Cadena

Nota:

- *Convertir el tipo de dato booleano en entero. Verdadero en 1 y Falso en 0.*
- *Convertir el tipo de dato carácter en entero, usando su código ASCII.*
- *Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico.*

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	Decimal	Boolean	Caracter	Cadena
Entero	Entero	Decimal	Entero	Entero	
Decimal	Decimal	Decimal	Decimal	Decimal	
Boolean	Entero	Decimal			
Caracter	Entero	Decimal		Entero	
Cadena					

Nota:

- *Convertir el tipo de dato booleano en entero. Verdadero en 1 y Falso en 0.*
- *Convertir el tipo de dato carácter en entero, usando su código ASCII.*
- *Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico.*

Multiplicación

Operación aritmética que consiste en multiplicar dos números. El símbolo para representar la operación es el asterisco (*).

Especificaciones de la operación multiplicación

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

*	Entero	Decimal	Boolean	Caracter	Cadena
Entero	Entero	Decimal	Entero	Entero	
Decimal	Decimal	Decimal	Decimal	Decimal	
Boolean	Entero	Decimal			
Caracter	Entero	Decimal		Entero	
Cadena					

Nota:

- *Convertir el tipo de dato booleano en entero. Verdadero en 1 y Falso en 0.*
- *Convertir el tipo de dato carácter en entero, usando su código ASCII.*
- *Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico.*

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 el símbolo (/).

Observaciones:

- Al dividir un numero en 0, se considera como un error semántico.

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

/	Entero	Decimal	Boolean	Caracter	Cadena
Entero	Decimal	Decimal	Decimal	Decimal	
Decimal	Decimal	Decimal	Decimal	Decimal	
Boolean	Decimal	Decimal			
Caracter	Decimal	Decimal		Decimal	
Cadena					

Nota:

- *Convertir el tipo de dato booleano en entero. Verdadero en 1 y Falso en 0.*
- *Convertir el tipo de dato carácter en entero, usando su código ASCII.*
- *Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico.*

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 (^).

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

^	Entero	Decimal	Boolean	Caracter	Cadena
Entero	Entero	Decimal	Entero		
Decimal	Decimal	Decimal	Decimal		
Boolean	Entero	Decimal	Entero		
Caracter					
Cadena					

Nota:

- *Convertir el tipo de dato booleano en entero. Verdadero en 1 y Falso en 0.*
- *Cualquier otra combinación no especificada en esta tabla es inválida y será considerado un error de tipo semántico.*

Módulo

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

Especificaciones de la operación potencia

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

%	Entero	Decimal	Boolean	Caracter	Cadena
Entero	Decimal	Decimal			
Decimal	Decimal	Decimal			
Boolean					
Caracter					
Cadena					

Negación Unaria

Es una operación que niega el valor de un número, es decir que devuelve el contrario del valor original. Se utiliza el símbolo menos (-).

Observaciones:

- Únicamente intercambia el valor original y no cambia el tipo de dato.
- Los únicos tipos de datos validos es el Entero y Decimal.

Nota:

- Cualquier otra combinación es inválida y será considerado un error de tipo semántico.

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:

Mayor, Menor, Mayor o igual, Menor o igual
A continuación, se especifica con una tabla, los resultados posibles al usar los operadores relacionales:

- Mayor: Para representar esta operación se usará el carácter >.
- Menor: Para representar esta operación se usará el carácter <.
- Mayor o igual: Para representar esta operación se usara el carácter >=.
- Menor o igual: Para representar esta operación se usara el carácter <=.

>,<,>=,<=	Entero	Decimal	Boolean	Caracter	Cadena
Entero	Booleano	Booleano		Booleano	
Decimal	Booleano	Booleano		Booleano	
Boolean					
Caracter	Booleano	Booleano		Booleano	
Cadena					

Nota:
- Cualquier otra combinación es inválida y será considerado un error de tipo semántico.
- El tipo de carácter se deberá convertir a ASCII.

Igual, diferente
A continuación, se especifica con una tabla, los resultados posibles al usar los operadores relacionales:

- Igual: Compara ambos valores y verifica si son iguales
- Diferente: Compara ambos lados y verifica si son distintos

==,!=	Entero	Decimal	Boolean	Caracter	Cadena
Entero	Booleano	Booleano			
Decimal	Booleano	Booleano			
Boolean			Booleano		
Caracter				Booleano	Booleano
Cadena				Booleano	Booleano

Nota:
- Cualquier otra combinación es inválida y será considerado un error de tipo semántico.

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 dicha instrucción.

El primer operando ternario corresponde a la condición que debe de cumplir una expresión booleana, de lo contrario se considera como un error semántico. El segundo y tercer operando están separados por el carácter dos puntos (:), y son de cualquier tipo.

```
//Estructura
<EXPRESION_BOOLEANA> ? <EXPRESION> : <EXPRESION>

//Ejemplo del uso del operador ternario int edad = 18;
Boolean mayoria_edad = false;
mayoria_edadd = edad > 21 ? true : false;
```

Nota: Este operador ternario puede ser usado como una instrucción y como una expresión.

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<55.5)|| bandera==true	bandera es true
&&	AND: Compara expresiones lógicas y si son ambas verdaderas entonces devuelve verdadero en otro caso retorna falso.	(flag1==flag2) && ("hola" == "hola")	flag1 y flag2 es true
!	NOT: Devuelve el valor inverso de una expresión lógica si esta es verdadera entonces devolverá falso, de lo contrario retorna verdadero.	!var1	var1 es true

Signos de Agrupación
Los signos de agrupación serán utilizados para agrupar operaciones aritméticas, lógicas o relacionales. Los símbolos de agrupación están dados por el carácter ‘(‘ y el carácter ‘)’.

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 instrucción se utilizará el signo dos puntos (;).
- Encapsular sentencias: para encapsular sentencias dadas por los ciclos, métodos, funciones, etc, se utilizará los signos { y }.

```
//Ejemplo de encapsulamiento de instrucción
Int año = 2022;

//Ejemplo de encapsulamiento de sentencias
```

```
{
    int semestre= 2;
}
```

Precedencia de Operadores

NIVEL	OPERADOR
0	-
1	^
2	/, *
3	+, -
4	==, !=, <, <=, >, >=
5	!
6	&&
7	||

Declaración y asignación de variables
El lenguaje será fuertemente tipado (el tipo de dato de una variable no puede cambiar, siempre será el mismo tipo). Cada una de las variables deberá de ser declarada antes de poder ser utilizada, de lo contrario será un error semántico. Las variables cuentan con un identificador y no podrán repetirse en el mismo entorno, para los identificadores es válido usar caracteres del abecedario [A-Z], números y guiones bajos. 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.

```
//Estructura de declaracion
<TIPO> <IDENTIFICADOR> ;
<TIPO> <IDENTIFICADOR> = <EXPRESION>;

//Estructura de asignacion
<IDENTIFICADOR> = <EXPRESION>;


//Ejemplos
<TIPO> mi_identificador;
<TIPO> id1_2, id2_, id_3, id_4;
<TIPO> mi_identificador2_ = <EXPRESION>;
<TIPO> id1, id2, id3, id4_2 = <EXPRESION>;

string curso_ = “organización de compiladores 1 2022”;
char var_111 = ‘a’;
```

```
celular = true;
int var1, var2, var3;
boolean flag_1, flag2_, flag33 = false;
char ch_1, ch_2, ch3, qwert123 = 'M';
universidad= "usac";
boolean flag_personalizada;
```

Nota:

- Al momento de declarar una variable o asignar una variable, comprobar que el tipo de dato de la expresión sea compatible con el tipo de dato declarado.
- Cuando una asignación no tiene una expresión, se tomara los datos por defectos.

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:

>,<,>=,<=	Entero	Decimal	Boolean	Caracter	Cadena
Entero	X	X	X	X	X
Decimal	X	X			X
Boolean	X	X	X	X	X
Caracter	X	X	X	X	X
Cadena	X	X	X	X	X

```
'('<TIPO>')' <EXPRESION>
//Ejemplos
Int edad = (Int) 18.6; //toma el valor entero de 18
Int intBool = (Int) false; //false = 0, true = 1
Int intDecimal = (Int) '0'; // = 0 caracteres numerales de longitud 1
Int intCadena = (Int) "100"; // = 100 cadena de caracteres de longitud variable

Double doubleInt = (Double) 16; //16.0
Double doubleString = (Double) "16.0"; //16.0 Cadenas numericas

Boolean boolInt = (Boolean) 0; //false
Boolean boolDouble = (Boolean) 1.0; //true
Boolean boolString = (Boolean) "0"; //false
Boolean boolChar = (Boolean) '1'; //true

Char charInt = (Char) 16; //'1' solo toma el primer carácter
Char charDouble = (Char) 16.1; //'1' solo toma el primer carácter
```

```
Char charBoolean = (Char) false; //'0' false = '0' true='1'
Char charString = (Char) "asdf"; //'a' solo toma el primer carácter

String stringInt = (String) 16; //'16"
String stringDouble = (String) 16.1; //"16.1"
String stringBoolean = (String) false; //"0" false = "0" true="1"
String stringChar (String) "a"; //""a"
```

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
```

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 únicamente los vectores.

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 o dos dimensiones.**

- La posición de cada vector será N. Por ejemplo, si deseo acceder al primer valor de un vector debo acceder como miVector[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 (una dimensión)
<TIPO> '[' ']' <ID> = new <TIPO>'[' <EXPRESION> ']' ';'
<TIPO> '[' ']' '[' ']' <ID>= new <TIPO>'[' <EXPRESION> ']' '[' <EXPRESION> ']' ';'

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

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

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

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

Acceso a Vectores

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

Nota: esto cuenta como una expresión valida, para parámetros, operaciones, y cualquier tipo de expresion de retorno etc. etc.

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

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

Modificación de Vectores

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‘;’

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

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

/*
RESULTADO
vector2[1]= “OLC1 ”
vector2[2]= “2do Semestre 2021”
```

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.

Nota:

- También, entre las sentencias pueden tener Ifs anidados.

Sentencia 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. Para anidar las sentencias if se utilizara “elif”

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

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

| ‘if’ ‘(’ [<EXPRESION>] ‘)’ ‘{‘ [<INSTRUCCIONES>]
‘}’ elif ‘(’ [<EXPRESION>] ‘)’ ‘{‘
    [<INSTRUCCIONES>]
 }’

//Ejemplo de cómo se implementar una sentencia if
```

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

//Ejemplo de cómo se implementar una sentencia if else

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


//Ejemplo de cómo se implementar un ciclo else if

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

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.

Switch

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

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

Case

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

Dentro de cada break puede o no venir el break.
De no venir el break debe seguir evaluando las opciones.

```
‘case’ [<EXPRESION>] ‘:’ [<INSTRUCCIONES>]
```

Default

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

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

Ciclos

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.

Notas:

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

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)
{
Print("Mayor que 50");
//Más sentencias
}
else
{
Print("Menor que 100");
//Más sentencias
} X++;
//Más sentencias
}
```

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.

Notas:

- 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++ ){
Println(“i=”+i)
//más sentencias
}

/*RESULTADO
i=0 i=1 i=2
*/

//Ejemplo 2: asignación de variable previamente declarada y decremento por asignación
Int i;
for ( i=5; i>2;i=i-1 ){
Print(“i=”+i+”\n”)
//más sentencias
}
/*RESULTADO
i=5 i=4 i=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){
          Println(true)
    }
    else{
          Println(false)
    }
    a--;
} while (a>0);

/*RESULTADO
false false false true true
```

```
*/
```

DO UNTIL

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

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

 //Ejemplo de cómo se implementar un ciclo do-while Int a=5;
do{
    if (a>=1 && a <3){
          Println(true)
    }
    else{
          Println(false)
    }
    a--;
} until (a==0);

/*RESULTADO
false false false true true
*/
```

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:

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){
        Println(“Me salgo del ciclo en el numero “ + i);
        break;
    }
    Println(i);
```

```
}
```

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){
        Println(“Me salte el numero ” + i); continue;
    }
    Println(i);
}
```

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.

Nota: los metodos no pueden retornar expresiones o valores, las funciones si, se debe validar que las funciones en todo momento retornen un valor.

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

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 identificador único, luego un tipo de dato 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.

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

|     [<TIPO>] [<ID>]


//Ejemplo de declaración de una función de enteros

 conversion (double pies, string tipo): double  {
     if (tipo == “metro”)
     {
          return pies/3.281;
     }else{
         return -1;
     }
}
```

Metodo

Un método también es una subrutina de código que se identifica con un nombre y un conjunto de parámetros, aunque a diferencia de las funciones estas subrutinas no deben de retornar un valor. Para este lenguaje los métodos serán declarados haciendo uso un identificador del método, seguido de la palabra reservada 'void' (que puede o no aparecer), 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 {}.

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


        //Ejemplo con tipo definido de un método

        holamundo(): void {
                Print("Hola mundo");
        }

        //Ejemplo sin tipo definido de un método

        HolaCompi() {
                Print("Hola Compi 1");
        }
```

Tomar en cuenta 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.

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

Print(“Ejemplo de llamada a método”);
holamundo();
/* Salida esperada
Ejemplo de llamada a método Hola Mundo
*/
//Ejemplo de llamada de una función

Print(“Ejemplo de llamada a función”);
Int num = suma(6,5); // a = 11
Println(“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
*/

suma(int num1, int num2):int {
Println(“Aquí puede venir cualquier sentencia :D”);
return num1 + num2;
Println(“Aquí pueden venir más sentencias, pero no se ejecutarán por la sentencia RETURN D:”); //Print 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 run(5.26).

- Para la llamada a métodos como una instrucción se deberá de agregar el punto y coma (;) como una instrucción.

Función Print

Esta función nos permite imprimir expresiones con valores únicamente de tipo entero, doble, booleano, cadena y carácter. Esta función no concatena un salto de línea al final del contenido que recibe como parámetro.

```
Print ( <EXPRESION> );
//Ejemplo
Print("Hola mundo!!");
Print("Sale compi \n" + valor + "!!");
Print(suma(2,2));
/*
Salida Esperada:
Hola Mundo!!Sale compi
25!!4
*/
// 25 es el valor almacenado en la variable "valor"
```

Función Println

Esta función nos permite imprimir expresiones con valores únicamente de tipo entero, doble, booleano, cadena y carácter. Esta función concatena un salto de línea al finalizar el contenido que recibe como parámetro

```
Println ( <EXPRESION> );
//Ejemplo
Println("Hola mundo!!");
Println("Sale compi \n" + valor + "!!");
Println(suma(2,2));
/*
Salida Esperada:
Hola Mundo!!
Sale compi
25!!
4
*/
// 25 es el valor almacenado en la variable "valor"
```

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"
```

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”
```

Función 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 ( ) ;

// Ejemplo

Double valor = round(5.8); //valor = 6

Double valor2 = round(5.4); //valor2 = 5
```

Funciones nativas

Length
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
string[ ] vector2 = ["hola", "Mundo"];

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.

Typeof

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

```
typeof ( <VALOR> ) ;

//Ejemplo

Int[] lista2 =new int [];
 String tipo = typeof(15); // tipo = “int”
String tipo2 = typeof(15.25) // tipo = “double”
 String tipo3 = typeof(lista2);// tipo3 = “vector”
```

ToString

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.

ToCharArray

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

```
toCharArray ( <VALOR> );

//Ejemplo

Char[] caracteres = toCharArray(“Hola”);
/*
caracteres [1] = “H”
caracteres [2] = “o”
caracteres [3] = “l”
caracteres [4] = “a”
*/
```

Push

Esta instrucción permite ingresar un elemento a un vector de una dimensión, cuando el vector es de más dimensiones se reconoce como un error semántico. Usara la palabra reservada “Push” y dentro de paréntesis el elemento que se

desea ingresar es necesario hacer un reconocimiento de tipo de datos para hacer la inserción correcta. La inserción de este elemento será al final del vector

```
//vector de ejemplo
string[ ] vector2 = ["hola", "Mundo"];

vector2.push(“bonito”);

//contenido del vector
//= ["hola", "Mundo",”bonito”]
```

Pop
Esta instrucción eliminara el ulimo elemento de un vector de una dimensión. Cuando es de mas dimensiones, se reportara como error semántico.

```
//vector de ejemplo
string[ ] vector2 = ["hola", "Mundo",”bonito”];

vector2.pop();

//contenido del vector
//= ["hola", "Mundo"]
```

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

```
run <ID> ( ) ;
run <ID> ( <LISTAVALORES> ) ; LISTAVALORES->LISTAVALORES ,
EXPRESION
| EXPRESION
//Ejemplo 1 funcion1():void{
Print(“hola”);
}

run funcion1();
/*RESULTADO
hola
*/
```

```
//Ejemplo 2
funcion2(string mensaje):void{
Print(mensaje);
}

Run funcion2(“hola soy un mensaje”);
/*RESULTADO
Hola soy un mensaje
*/
```

Reportes

Los reportes son una parte fundamental de compscript, 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).

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**

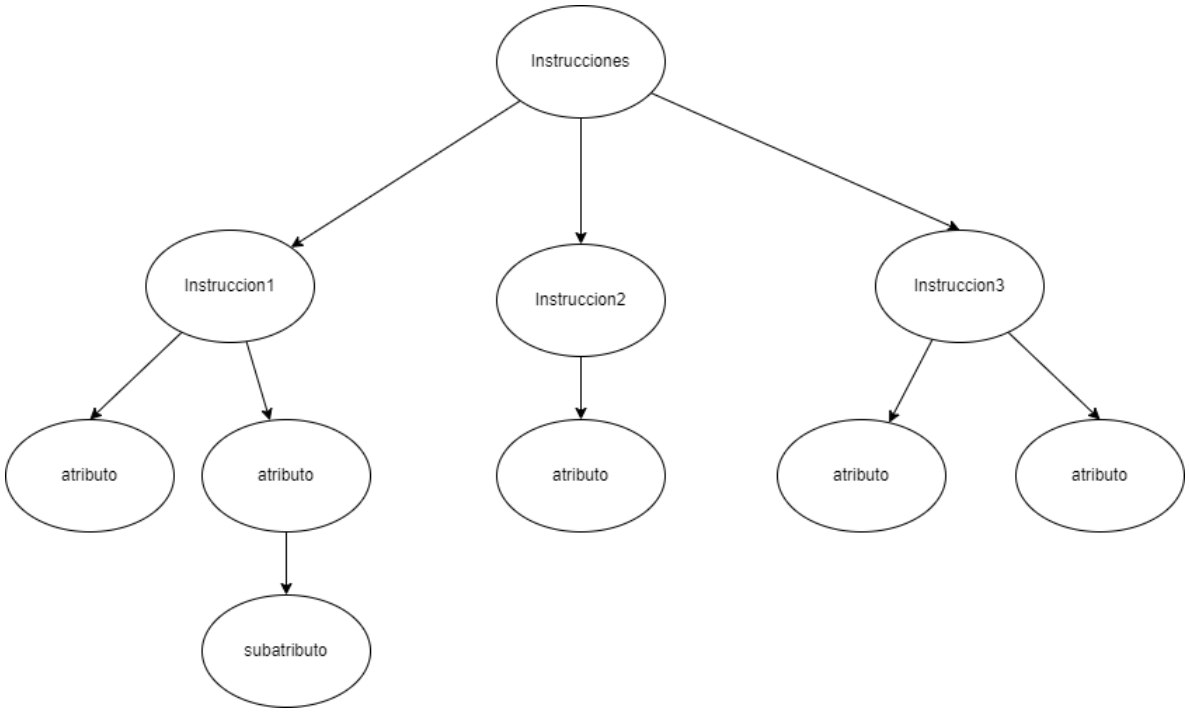
Tabla de errores

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 perteneceal lenguaje.	7	**32**
2	**Sintáctico**	**Encontrado Identificador “Ejemplo”, se esperaba Palabra Reservada “Valor”**	**150**	**12**

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.

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 “Print”, 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
```

Entregables a Calificar

- 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 o Gitlab en donde se encuentra su proyecto (Se usara el mismo repositorio del proyecto1).

Restricciones

1. Lenguajes de programación a usar: **Javascript/Typescript.**
2. Debe utilizar un framework a su elección: Angular, React, Vuejs, etc para generar su entorno gráfico. Queda a discreción del estudiante.
3. Debe utilizar la herramienta Nodejs para el Servidor.
4. Herramientas de análisis léxico y sintáctico a utilizar: 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. **Para poder calificarse es obligatorio mostrar el reporte del AST.**

Aclaraciones:
Todo el análisis semántico tendrá un ponderado extra sobre la hoja de calificación.

NO HAY PRORROGAS.

Fecha de Entrega

3 de noviembre de 2022.

La entrega será por medio de la plataforma UEDI (el entregable será el enlace de su repositorio), si existieran inconvenientes con la plataforma se utilizarán medios alternativos definidos por sus respectivos auxiliares.

Entregas fuera de la fecha indicada, no se calificarán.

SE LE CALIFICARA DEL COMMIT REALIZADO HASTA ESTA FECHA.