Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Organización de Lenguajes y Compiladores 1
Primer Semestre 2024
Catedráticos:
Ing. Mario Bautista
Ing. Manuel Castillo
Ing. Kevin Lajpop
Tutores académicos:
Walter Guerra
Fabian Reyna
Carlos Acabal

CompiScript+
Proyecto 2

Tabla de Contenido

1. Objetivo General

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.

2. Objetivos específicos

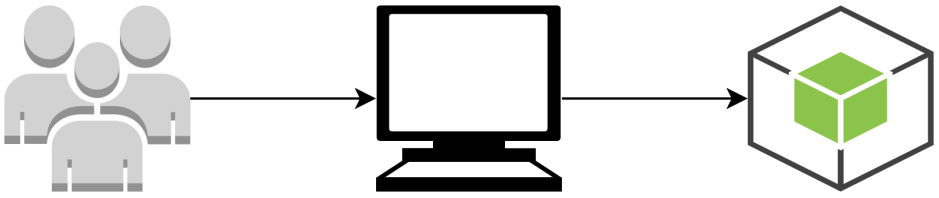
- Reforzar los conocimientos de análisis léxico, sintáctico y semántico para la creación de un lenguaje de programación.
- Aplicar los conceptos de compiladores para implementar un proceso de interpretación de código de alto nivel.
- Aplicar los conceptos de compiladores para analizar un lenguaje de programación y producir las salidas esperadas.
- Aplicar la teoría de compiladores para la creación de soluciones de software.
- Generar aplicaciones utilizando la arquitectura cliente-servidor

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

Arquitectura Cliente-Servidor



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 de código fuente que será analizado. En este se podrán abrir diferentes archivos al mismo tiempo. El editor de texto se tendrá que mostrar en el navegador. Queda a discreción del estudiante el diseño.

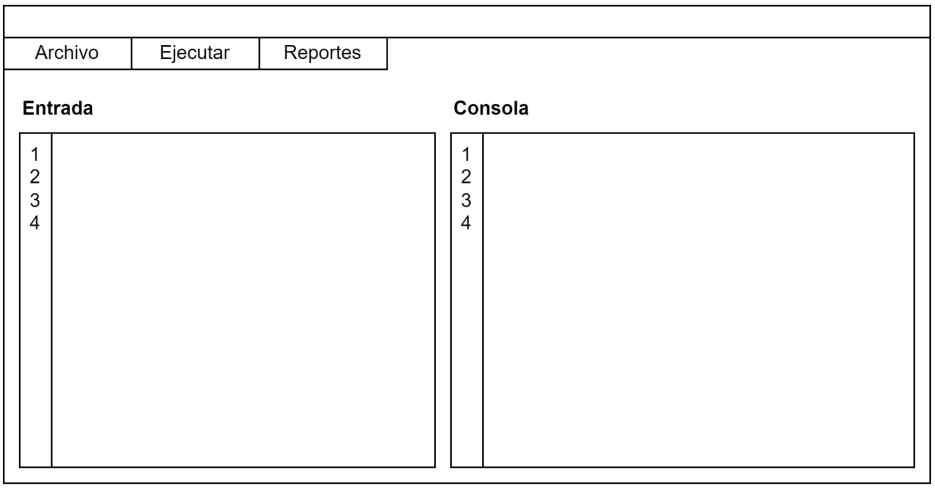


Figura 1: Propuesta Interfaz

4.2 Funcionalidades

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

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:** se debe generar una imagen del árbol de análisis sintáctico que 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 es case insensitive por lo que no reconoce entre mayúsculas y minúsculas. Ejemplo:

```
int a = 0;

iNt A = 0;
```

Nota: Ambos casos son lo mismo

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 un mensaje en específico. El lenguaje deberá soportar dos tipos de comentarios que son los siguientes;

5.2.1 Comentarios de una línea

Este comentario comenzará con // y deberá terminar con un salto de línea.

5.2.2 Comentarios multilínea

Este comentario comenzará con /* y terminará con */.

```
// Esto es un comentario de una sola línea
/* Esto es un comentario
Multilínea */
```

5.3 Tipos de Dato

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 dato aceptará solamente números enteros	1, 50, 100, -120, etc	Del -2147483648 al 2147483647	0
Doble	double	Admite valores numéricos con decimales	1.2, 50.23, 00.34. etc	Se maneja cualquier cantidad de decimales	0.0
Booleano	bool	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', '1', '&', '\', '\n', etc	En caso de querer escribir comilla simple, se escribirá \ y después la comilla simple \'. Si se quiere escribir \ se escribirá \\. Existirá también \n, \t, \r, \".	'\u0000' (carácter 0)
Cadena	std::string	Es un grupo o conjunto de caracteres que pueden tener cualquier carácter, y este se encontrará delimitado por comillas dobles. " "	"cadena", "- cad"	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)

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	Descripción	Ejemplo
\n	Salto de línea	"Hola\nMundo"
\\	Barra invertida	"C:\\miCarpeta"
\"	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. Para esta se utiliza el signo más **(+)**.

Especificaciones de la operación suma

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

+	Entero	Doble	Boolean	Carácter	Cadena
Entero	Entero	Doble	Entero	Entero	Cadena
Doble	Doble	Doble	Doble	Doble	Cadena
Boolean	Entero	Doble			Cadena
Carácter	Entero	Doble		Cadena	Cadena
Cadena	Cadena	Cadena	Cadena	Cadena	Cadena

Nota: Cualquier otra combinación no especificada en esta tabla es invalida 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. Para esta se utiliza el signo menos **(-)**.

Especificaciones de la operación resta

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

-	Entero	Doble	Boolean	Carácter	Cadena
Entero	Entero	Doble	Entero	Entero	
Doble	Doble	Doble	Doble	Doble	
Boolean	Entero	Doble			
Carácter	Entero	Doble			
Cadena					

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

5.5.3 Multiplicación

Es la operación aritmética que consiste en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). El signo 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	Doble	Boolean	Carácter	Cadena
Entero	Entero	Doble		Entero	
Doble	Doble	Doble		Doble	
Boolean					
Carácter	Entero	Doble			
Cadena					

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

5.5.4 División

Es la 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 de la operación división

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

/	Entero	Doble	Boolean	Carácter	Cadena
Entero	Doble	Doble		Doble	
Doble	Doble	Doble		Doble	
Boolean					
Carácter	Doble	Doble			
Cadena					

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

5.5.5 Potencia

Es una operación aritmética de la forma **pow(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 pow(5, 3), a=5 y b=3 tendríamos que multiplicar 3 veces 5 para obtener el resultado final; 5x5x5 que da como resultado 125.

Especificaciones de la operación potencia

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

pow	Entero	Doble	Boolean	Carácter	Cadena
Entero	Entero	Doble			
Doble	Doble	Doble			
Boolean					
Carácter					
Cadena					

Nota: Cualquier otra combinación no especificada en esta tabla es invalida 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 número entre otro. Para realizar la operación se utilizará el signo **(%)**.

Especificaciones de la operación módulo

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

%	Entero	Doble	Boolean	Carácter	Cadena
Entero	Doble	Doble			
Doble	Doble	Doble			
Boolean					
Carácter					
Cadena					

Nota: Cualquier otra combinación no especificada en esta tabla es invalida 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. Se utiliza el símbolo menos **(-)**.

Especificaciones de la operación negación

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

-exp	Resultado
Entero	Entero
Doble	Doble
Boolean	
Carácter	
Cadena	

Nota: Cualquier otra combinación no especificada en esta tabla es invalida 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-carácter, doble-entero, doble-carácter, carácter-entero, carácter-doble, carácter-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 válida.

Operador	Descripción	Ejemplo
==	Igualación: compara ambos valores y verifica si son iguales - Iguales → True - No iguales → False	1==1 "hola" == "hola" 25.654 == 54.34
!=	Diferenciación: compara ambos lados y verifica si son distintos - Iguales → False - No iguales → True	1 != 2 var1 != var2 50 != 30
<	Menor que: compara ambos lados y verifica si el derecho es mayor que el izquierdo. - Derecho mayor → True - Izquierdo mayor → False	25.5 < 30 54 < 25 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	25.5 <= 30 54 <= 25 50 <= 'F'
>	Mayor que: compara ambos lados y verifica si el izquierdo es mayor que el derecho. - Derecho mayor → False - Izquierdo mayor → True	25.5 > 30 54 > 25 50 > 'F'
>=	Mayor o igual que: Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho. - Derecho mayor → False - Izquierdo mayor o igual → True	25.5 >= 30 54 >= 25 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.

<CONDICIÓN> '?' <EXPRESION> ':' <EXPRESION>

```
int edad = 18;
bool 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 y en otro caso retorna falso.	bandera || 5<2 Devuelve true	bandera es true
&&	AND: compara expresiones lógicas y si ambas son verdaderas, entonces devuelve verdadero y en otro caso retorna falso.	flag && "hola" == "hola" Devuelve true	flag es true
!	NOT: devuelve el valor inverso de una expresión lógica si esta es verdadera entonces devolverá false, de lo contrario retorna verdadero	!var Devuelve false	var es true

5.9 Signos de Agrupación

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

5.10 Precedencia de Operaciones

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

Nivel	Operador	Asociatividad
0	-	Derecha
1	pow	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 instrucción se utilizará el signo **;**
- **Encapsular sentencias:** para encapsular sentencias dadas por ciclos, métodos, funciones, etc, se utilizará los signos **{** y **}**.

```
// Ejemplo de finalización de instrucciones
int edad = 18;
// Ejemplo de encapsulamiento de sentencias
if(1==1){
   int a = 10;
   int b = 20;
}
```

5.12 Caracteres de finalización y encapsulamiento de sentencias

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>  id1. id2, id3, id4 = <EXPRESION>
// Ejemplos
int numero;
int var1, var2, var3;
std::string cadena = "hola";
char var4 = 'a';
bool flag = true;
double a, b, c = 5.5;
```

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.

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 entre 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

```
// Ejemplos
int edad = (int) 18.6; // toma el valor entero de 18
char letra = (char) 70; // toma el valor 'F' ya que el 70 en ascii es F
double numero = (double) 16; // toma el valor de 16.0
```

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 de 18
```

5.15 Estructuras de Datos

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, bool, char o string. **El lenguaje permitirá únicamente el uso de arreglos de una o dos dimensiones.**

Observaciones:

- La posición de cada vector será N-1. Por ejemplo, si se quiere acceder al primer valor de un vector se accede como vector[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.

```
// Declaración de tipo 1
<TIPO> <ID> '[' ']' = new <TIPO> '[' <EXPRESION> ']' ';'
<TIPO> <ID> '[' ']' '[' ']' = new <TIPO> '[' <EXPRESION> ']' '[' <EXPRESION> ']' ';'
```

```
// Declaración de tipo 2
<TIPO> <ID> '[' ']' ='[' <LISTAVALORES> ']' ';'

// Ejemplo
int vector1[] = new int[4];
char vector2[][] = new char[4][4];

std::string vector3[] = ["Hola", "Mundo"];
int vector4 [][] = [ [1, 2], [3, 4] ];
```

5.15.1.2 Acceso a vectores

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

```
<ID> '[' <EXPRESION> ']'
// Ejemplos
std::string vector3[] = ["Hola", "Mundo"];
std::string valor3 = vector3[0]; // Almacena el valor "hola"

int vector4 [][] = [ [1, 2], [3, 4] ];
int valor4 = vector4[0][0]; // Almacena el valor 1
```

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

```
// Ejemplos
std::string vector3[] = ["Hola", "Mundo"];
vector3[0] = "OLC1";
vector3[1] = "1er Semestre";
```

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 if 's 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
if(x<50){
 //sentencias
}
```

5.16.1.2 if else

```
// Ejemplo
if(x<50){
 //sentencias
} else{
 //sentencias
}
```

5.16.1.3 else

```
// Ejemplo
if(x>50){
 //sentencias
} else if(x<50 && x>0){
 //sentencias
}else {
 //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> ]
```

5.16.2.2 Default

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

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

//ejemplos
int edad = 18;
switch( edad ) {
  Case 10:
     // sentencias
    // mas sentencias Break;
  Case 18:
    // sentencias
    // mas sentencias Break;
  Case 25:
    // sentencias
    // mas sentencias Break;
  Default:
    // sentencias
    // mas sentencias Break;
}
```

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
while(x<100){
   //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, etc, es decir, cualquier tipo de asignación
- Dentro pueden venir N instrucciones

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

```
//Ejemplo
for(int i = 0; i<3; i++){
  //sentencias
}
```

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.

Observaciones:

- Dentro pueden venir N instrucciones

```
'do' '{'
   [ <INSTRUCCIONES> ]
'}' 'while' '(' <EXPRESION> ')' ';'
//Ejemplo
do{
  //sentencias
}while(x<100)
```

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
for(int i = 0; i<3; i++){
   if (i==2){
      break; // me salgo en i = 2
   }
}
```

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
for(int i = 0; i<5; i++){
    if (i==2){
        continue; // me salte 'mas sentencias' en i = 2
    }
    // mas sentencias
}
```

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
for(int i = 0; i<5; i++){
    if (i==2){
        return;
    }
}

for(int i = 0; i<5; i++){
    if (i==2){
        return i;
    }
}
```

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 único, 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
int conversion (double size, std::string tipo){
   if(tipo=="metro"){
      return size/3*3.281;
   } else{
     return -1;
   }
}
```

Cabe 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 tipo, 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 de la palabra reservada 'void', seguido de un 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
void hola_mundo (){
   cout << "hola mundo" ;
}
```

Cabe 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 asignándole a una variable del tipo adecuado, bien integrándose en una expresión.

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

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

PARAMETROS_LLAMADA → PARAMETROS_LLAMADA ',' <TIPO> <ID>
                   | <TIPO> <ID>

//Ejemplo
void hola_mundo (){
    cout << "hola mundo" ;
}
hola_mundo();

int conversion (double size, std::string tipo){
    if(tipo=="metro"){
        return size/3*3.281;
    } else{
       return -1;
    }
}
int resultado = conversion(58.5, "metro");
```

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 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 execute(5.26).
- Para la llamada a métodos como una instrucción se deberá de agregar el punto y coma (;) como una instrucción.

5.21 Función cout

Esta función nos permite imprimir expresiones con valores únicamente de tipo entero, doble, booleano, cadena y carácter. Al utilizar la palabra reservada 'endl' se permitirá realizar un salto de línea al final del contenido

```
'cout' '<<' <EXPRESION> ';'
'cout' '<<' <EXPRESION> '<<' 'endl' ';'

//Ejemplo
cout << "hola mundo!";
cout << "sale compi1 \n"
cout << "primer semestre" <<endl;
cout << 2024;
/*Salida esperada:
hola mundo!sale compi1
primer semestre
2024
*/
```

5.22 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
std::string cadena1 = tolower("HOLa MundO");
//almacena 'hola mundo'
```

5.23 Función toupper

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

```
'toupper' '('  <EXPRESION> ')' ';'

//Ejemplo
std::string cadena1 = toupper("HOLa MundO");
```

```
//almacena 'HOLA MUNDO'
```

5.24 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 a 0.5, se aproxima al entero superior.
- Si el decimal es menor que 0.5, se aproxima al número inferior.

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

//Ejemplo
double valor = round(15.51); //almacena 16
double valor2 = round(9.40); //almacena  9
```

5.25 Funciones nativas

5.25.1 Length

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

Observación:

- Si se usa para otro parámetro de tipo de dato no especificado, se considera un error semántico.

```
<EXPRESION> '.' 'length' '('  ')' ';'
//EXPRESION puede ser cadenas o vectores

//Ejemplos
std::string vector[] = ["Hola", "Mundo"];
std::string cadena = "compi1";

int sizeVector = vector.length(); //Almacena 2
int sizeCadena = cadena.length(); //Almacena 6
```

5.25.2 Typeof

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

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

```
//Ejemplos
std::string vector[] = ["Hola", "Mundo"];
std::string cadena = "compi1";

cout << typeof(cadena) << endl;
cout<<typeof(vector)<< endl;
/*Salida esperada
cadena
vector
*/
```

5.25.3 ToString

Esta función permite convertir un valor de tipo numérico o bool a texto.

Observación:

- Si se usa para otro parámetro de tipo de dato no especificado, se considera un error semántico.

```
'std' ':' ':' 'toString' '(' <EXPRESION> ')' ';'

//Ejemplos

std::string var = std::toString(1+20+30) ; //almacena 51
std::string var2 = std::toString(true); //almacena true
```

5.25.4 Función c_str

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

-

```
<EXPRESION> '.' c_str'('  ')' ';'

//Ejemplos
std::string var1 = "compi1";
char[] caracteres = var.c_str();
//Almacena ['c', 'o', 'm', 'p', 'i', '1,']
```

5.26 Función Execute

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

```
'execute' <ID> '(' ')' ';'
'execute' <ID> '(' [<PARAMETROS>] ')' ';'

//Ejemplos
void funcion1(){
    cout << "Hola Mundo" << endl;
    cout << funcion2(10) << endl;
}
int funcion2(int numero){
return numero;
}

execute funcion1();
/* Resultado
Hola Mundo
10
*/
```

6. Reportes

Los reportes son parte fundamental de compiScript+, 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 errores

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

#	Tipo	Descripción	Línea	Columna
1	Léxico	El carácter "$" no pertenece al lenguaje	5	3
2	Sintáctico	Se encontró Identificador y se esperaba Expresión.	6	3

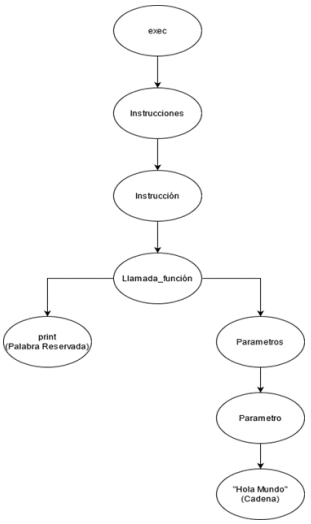
6.2 Tabla de Símbolos

Este reporte mostrará la tabla de símbolos después de la ejecución. Deberá mostrar las variables y arreglos declarados, así como su tipo, valor y toda la información que considere necesaria.

#	ID	Tipo	Tipo	Entorno	Línea	Columna
1	var	Variable	Entero	Funcion1	15	20
2	var2	Variable	Bool	Funcion2	20	13

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 instrucción "cout", 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:

- Interfaz Gráfica funcional
- Operaciones aritméticas, lógicas y relacionales
- Declaración y asignación de variables
- Sentencias de control
- Sentencias cíclicas
- Métodos (con o sin parámetros)
- Llamadas a métodos (con o sin parámetros)
- Instrucción cout
- Sentencia execute
- Reporte AST
- Consola
- Documentación completa (manual de usuario, manual técnico y archivo de gramáticas)

8. Entregables

- **Código fuente del proyecto**

- **Manual de Usuario en un archivo pdf**
    - Capturas de pantalla detallando cómo funciona su entorno de trabajo y los reportes que se generan.

- **Manual Técnico en un archivo pdf**
    - Información importante del proyecto para que se pueda realizar el mantenimiento en el futuro. Especificar el lenguaje, herramientas utilizadas, métodos y funciones más importantes.

- **Archivo de Gramática en un archivo txt**
    - El archivo debe contener su gramática y debe de ser limpio, entendible y no debe ser una copia del archivo de Jison.
    - La gramática debe estar escrita en formato **BNF(Backus-Naur form).**
    - Solamente se debe escribir la gramática independiente del contexto.

9. Restricciones

- La entrega debe ser realizada mediante UEDI enviando el enlace del **repositorio privado** de Github en donde se encuentra su proyecto.
- El nombre del repositorio de Github debe ser **OLC1_Proyecto2_#Carnet**
- Se debe agregar al auxiliar encargado como colaborador al repositorio de Github.
- Lenguaje de Programación a utilizar: **Javascript/TypeScript**
- Herramientas para el análisis léxico y sintáctico: **Jison**
- **El proyecto debe ser realizado de forma individual.**
- Para graficar se puede utilizar cualquier librería (Se recomienda graphviz)
- Puede utilizar un framework como Angular, React, Vuejs, etc, para generar su entorno gráfico. Queda a discreción del estudiante cuál utilizar.
- **Copias completas/parciales** de: código, gramática, 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.
- La calificación tendrá una duración de 30 minutos, acorde al programa del laboratorio.
- Se debe visualizar todo desde el navegador

10. Fecha de Entrega

Domingo, 21 de abril de 2024 a las 23:59. La entrega será por medio de la plataforma UEDI. Entregas fuera de la fecha indicada, no se calificarán.

SE LE CALIFICARA DEL ÚLTIMO COMMIT REALIZADO ANTERIOR A ESTA FECHA.