

Universidad de San Carlos de Guatemala  
 Facultad de Ingeniería  
 Escuela de Ingeniería en Ciencias y Sistemas  
 Organización de Lenguajes y Compiladores 2  
 Segundo Semestre 2021



**Catedráticos:** Ing. Edgar Sabán, Ing. Bayron López, Ing. Erick Navarro e Ing. Luis Espino  
**Tutores académicos:** Ronald Romero, Manuel Miranda, Jhonatan López, César Sazo, Pablo Roca

# JOLC

## Primer proyecto de laboratorio

<b>Competencias</b>	<b>3</b>
Competencia general	3
Competencias específicas	3
<b>Descripción</b>	<b>4</b>
Descripción General	4
Flujo específico de la aplicación	4
Ingreso de código fuente	5
Ejecución	5
Generación de reportes	5
<b>Componentes de la aplicación</b>	<b>5</b>
<b>Sintaxis de JOLC</b>	<b>7</b>
Generalidades	7
Tipos de dato válidos	8
Expresiones	9
Aritméticas	9
Multiplicación	10
División	11
Potencia	11
Módulo	12
Nativas	12
Relacionales	13
Lógicas	14
Impresión	15
Asignaciones	15

Funciones	17
Creación de funciones	17
Funciones Nativas	17
Llamada a funciones	17
Paso por valor o por referencia	18
Condicionales	18
Loops	19
Ciclo while	19
Ciclo for	19
Sentencias de transferencia	20
Arreglos	21
Structs	22
Tabla de Resumen de instrucciones	23
<b>Reportes generales</b>	<b>24</b>
Tabla de Símbolos	24
Árbol de análisis sintáctico	25
Tabla de errores	25
<b>Entregables y Calificación</b>	<b>26</b>
Entregables	26
Restricciones	27
Consideraciones	27
Calificación	28
Entrega de proyecto	28

# 1. Competencias

## 1.1. Competencia general

Que los estudiantes apliquen los conocimientos adquiridos en el curso para la construcción de un intérprete utilizando las herramientas establecidas.

## 1.2. Competencias específicas

- Que los estudiantes utilicen herramientas para la generación de analizadores léxicos y sintácticos.
- Que los estudiantes apliquen los conocimientos adquiridos durante la carrera y el curso para el desarrollo de la solución.
- Que los estudiantes realicen análisis semántico e interpretación del lenguaje JOLC

## 2. Descripción

### 2.1. Descripción General

Julia es un lenguaje de programación reciente que le interesa a científicos de datos, estadísticos y analistas financieros. Este cuenta con distintas características atractivas para los programadores. Aun así, Julia al ser un lenguaje de programación reciente, hay pocos lugares donde se pueda probar su sintaxis y ejecución de manera sencilla. Es por eso por lo que se le solicita el desarrollo de JOLC, un lenguaje de programación basado en Julia que se podrá programar y ejecutar desde cualquier navegador.

### 2.2. Flujo específico de la aplicación

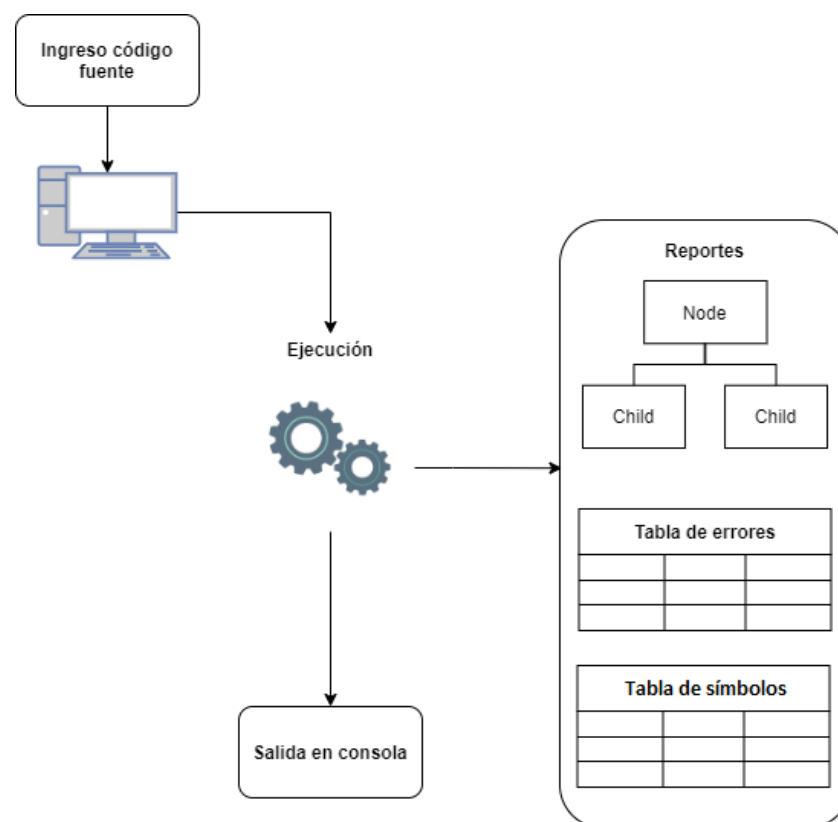


Ilustración 1: Flujo específico de la aplicación

## 2.3. Ingreso de código fuente

El lenguaje JOLC está basado en Julia con instrucciones limitadas. Se deberá de contar con un lugar en la página donde los desarrolladores puedan programar y ejecutar sus archivos de JOLC para luego ejecutarlos.

## 2.4. Ejecución

Si el código se ejecuta y encuentra errores en el mismo entonces se deberá contar con recuperación de errores, los **únicos errores de recuperación será tipo semántico**, esto con la finalidad de que se pueda seguir ejecutando el resto del código de entrada y finalizando mostrar dichos errores en los reportes respectivos, de lo contrario se generará la salida en consola satisfactoriamente.

## 2.5. Generación de reportes

El código fuente se colocará en la página web, ya sea escribiendo desde la página o copiándolo y pegándole de un archivo .jl hacia la página. El programa se ejecutará y mostrará los resultados en la consola. Luego de eso, el desarrollador podrá ver distintos reportes que se generaron con el código que ingresó. Entre los cuales se encuentran el **AST, Tabla de errores y la Tabla de símbolos**. Estos reportes serán para que se verifique cómo el estudiante usa las estructuras internas para la interpretación del lenguaje. En la sección de reportes se detallará más al respecto de estos reportes.

# 3. Componentes de la aplicación

La aplicación deberá contar con las siguientes vistas, con la libertad de diseñar a su creatividad.

- **Página de bienvenida:** En esta vista deberá mostrar sus datos personales.
- **Página de análisis:** Contiene el editor de entrada y consola de salida. No hace falta abrir archivos, basta con copiar y pegar la entrada.
- **Página de Reportes:** En esta vista se podrán consultar los reportes. Los reportes de errores y tabla de símbolos se deben mostrar en la aplicación. El reporte de AST se debe descargar en un formato en el que se vea claramente el contenido, entre los formatos considerados para mostrar los reportes son **png o svg**.

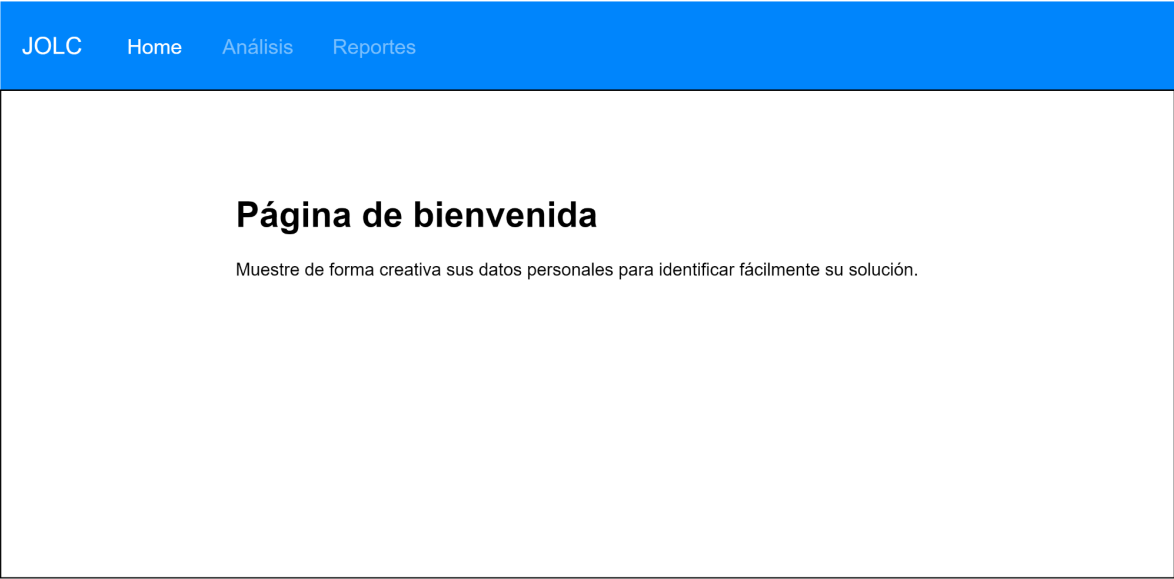


Ilustración 2: Página de bienvenida

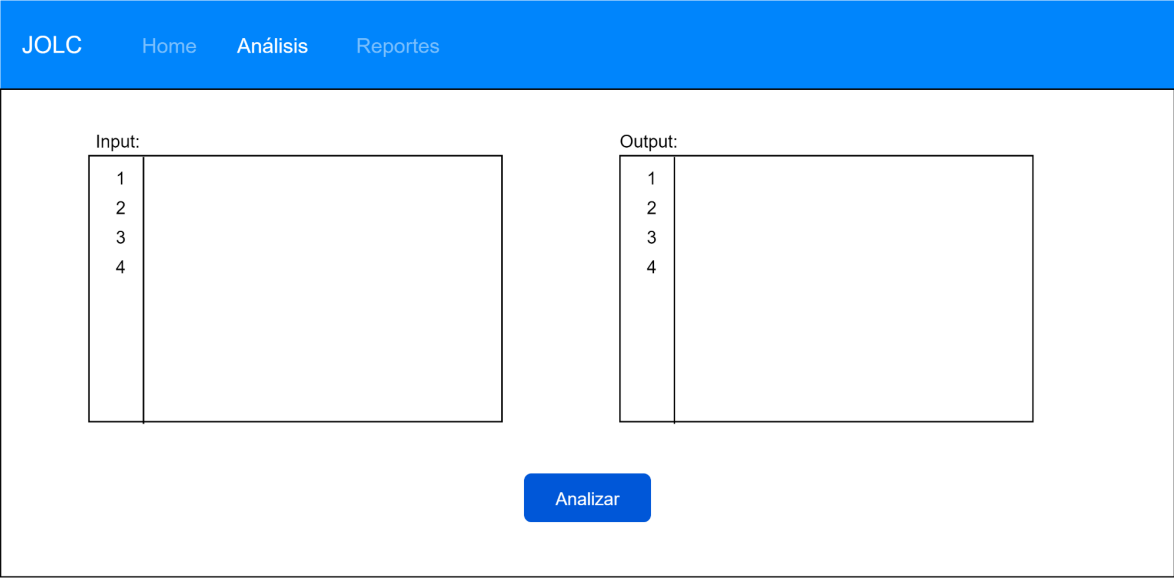


Ilustración 3: Vista de análisis

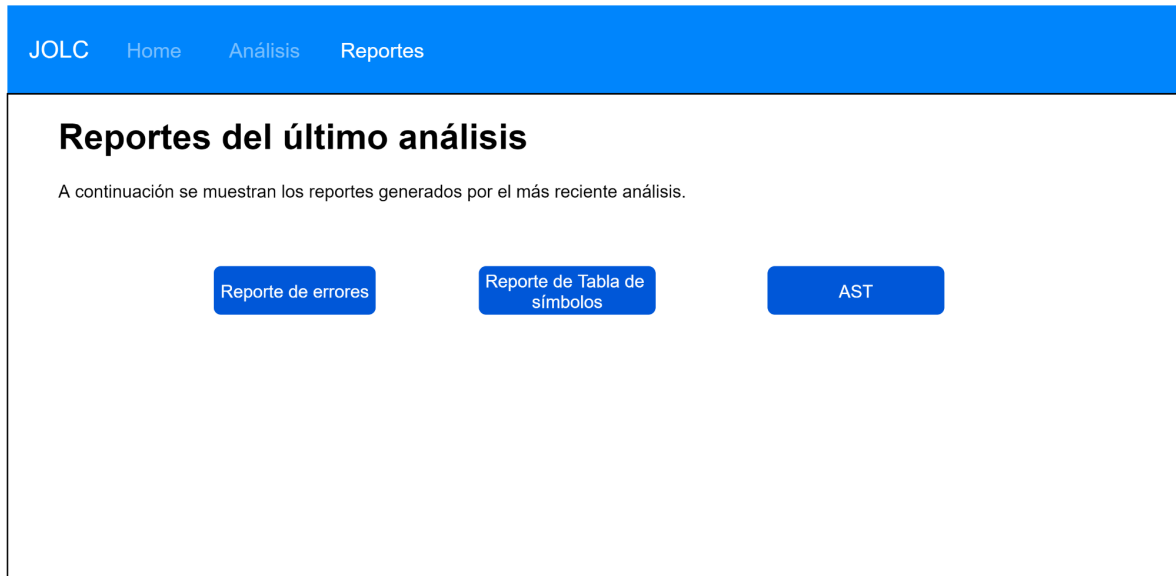


Ilustración 4: Página de consulta de reportes

## 4. Sintaxis de JOLC


JOLC provee distintas funcionalidades de Julia, aun así, al tener funciones limitadas de este lenguaje se debe de definir bien la sintaxis de este al ser Julia un lenguaje bastante amplio. En el siguiente enlace se encontrará con más detalle la sintaxis de JOLC y algunos archivos de entrada aceptados por JOLC: <https://github.com/ManuelMiranda99/JOLC>. También, para conocer más sobre la sintaxis de JOLC, puede revisar en la documentación de lenguaje de programación Julia, pero con las limitaciones que en el proyecto se describe, visite los siguientes enlaces donde encontrará documentación oficial <https://introajulia.org>, <https://docs.julialang.org/en/v1/>.

### 4.1. Generalidades

- Comentarios. Un comentario es un componente léxico del lenguaje que no es tomado en cuenta en el analizador sintáctico. Pueden ser de una línea (#) o de múltiples líneas(#= ... =#).
- Case Sensitive. Esto quiere decir que distinguirá entre mayúsculas y minúsculas.
- Identificadores. Un identificador de JOLC debe comenzar por una letra [A-Za-z] o guión bajo [\_] seguido de una secuencia de letras, dígitos o guión bajo.
- Fin de instrucción. Se hace uso del símbolo ";" para establecer el fin de una instrucción.

## 4.2. Tipos de dato válidos

JOLC únicamente aceptará los siguientes tipos de datos, cualquier otro no se deberá tomar en cuenta:

- **Nulo:** se representa con la palabra reservada *nothing*. Indica que no existe ningún valor.
- **Int64:** valores numéricos enteros. Por ejemplo: 3,2,-1.
- **Float64:** valores numéricos con punto flotante. Por ejemplo: 3.2,45.6,11.2. 
- **Bool:** valores booleanos, true o false.
- **Char:** Literales de caracteres, se definen con comillas simples. Por ejemplo: 'a'.
- **String:** Cadenas de texto definidas con comillas dobles.
- **Arreglos:** Conjunto de valores indexados entre 1 hasta n. Puede almacenar diferentes tipos. Para más información, consulte la sección 4.9.

```
[10, 20, 30, 40];  
["Hola", "Mundo"];  
['a', 2.0, 5, ["Hola", "Mundo"]];
```

- **Struct:** Estos son tipos compuestos definidos por el programador. Existen 2 tipos de Struct, aquellos que son **mutables y los inmutables**. Para mayor detalle, consulte la sección 4.10.

```
struct Rectangulo  
  base :: Int64;  
  altura;  
end;
```



## 4.3. Expresiones

### 4.3.1. Aritméticas

Una operación aritmética está compuesta por un conjunto de reglas que permiten obtener resultados con base en expresiones que poseen datos específicos durante la ejecución. A continuación se definen las operaciones aritméticas soportadas por el lenguaje.

#### Suma

La operación suma se produce mediante la suma de número o strings concatenados.

Operandos	Tipo resultante	Ejemplos
<code>Int64 + Float64</code> <code>Float64 + Int64</code> <code>Float64 + Float64</code>	<b>Float64</b>	$2 + 3.3 = 5.3$ $2.3 + 8 = 10.3$ $1.2 + 5.4 = 6.6$
<code>Int64 + Int64</code>	<b>Int64</b>	$2 + 3 = 5$
<code>String * String</code> <i>Nota: Int64 y Float64 pueden ser convertidos a string con la función nativa "parse" para ser utilizados en esta operación.</i>	<b>String</b>	<code>"hola"*"mundo"="holamundo"</code> <code>"Hola" * parse(string,8) = "Hola8"</code>
<code>String ^3</code>	<b>String</b>	<code>"CadenaCadenaCadena"</code>
<code>uppercase(String)</code>	<b>String</b>	<code>animal = "Tigre";</code> <code>println(uppercase(animal));</code> <code>#TIGRE</code>

lowercase(String)	<b>String</b>	<pre>animal = "Tigre"; println(lowercase(animal)); #tigre</pre>
-------------------	---------------	---

## Resta

La resta se produce cuando se sustraen el resultado de los operadores, produciendo su diferencia.

Operandos	Tipo resultante	Ejemplos
<div>Int64 - Float64</div> <div>Float64 - Int64</div> <div>Float64 - Float64</div>	<b>Float64</b>	<pre>2 + 3.3 = -1.3 2.3 + 8 = -5.7 1.2 + 5.4 = -4.2</pre>
<div>Int64 - Int64</div>	<b>Int64</b>	<pre>2 + 3 = -1</pre>

## Multiplicación

El operador multiplicación produce el producto de la multiplicación de los operandos.

Operandos	Tipo resultante	Ejemplos
<code>Int64 * Float64</code> <code>Float64 * Int64</code> <code>Float64 * Float64</code>	<b>Float64</b>	$2 * 3.3 = 6.6$ $2.3 * 8 = 18.4$ $1.2 * 5.4 = 6.48$
<code>Int64 * Int64</code>	<b>Int64</b>	$2 * 3 = 6$

## División

El operador división se produce el cociente de la operación donde el operando izquierdo es el dividendo y el operando derecho es el divisor.

Operandos	Tipo resultante	Ejemplos
<code>Int64 / Float64</code> <code>Float64 / Int64</code> <code>Float64 / Float64</code>	<b>Float64</b>	$2 / 3.3 = 0.60$ $2.3 / 8 = 0.2875$ $1.2 / 5.4 = 0.222$
<code>Int64 / Int64</code>	<b>Float64</b>	$6 / 4 = 1.5$

## Potencia

El operador de potenciación devuelve el resultado de elevar el primer operando al segundo operando de potencia.

Operandos	Tipo resultante	Ejemplos
<code>Int64 ^ Float64</code> <code>Float64 ^ Int64</code> <code>Float64 ^ Float64</code>	<b>Float64</b>	$2 ^ 3.5 = 11.31$ $2.3 ^ 8 = 783.10$ $1.2 ^ 5.4 = 2.67$
<code>Int64 ^ Int64</code>	<b>Int64</b>	$6 ^ 2 = 36$
<code>String ^ Int64</code>	<b>String</b>	$\text{"Hola"}^3 = \text{"HolaHolaHola"}$

## Módulo

El operador módulo devuelve el resto que queda cuando un operando se divide por un segundo operando.

Operandos	Tipo resultante	Ejemplos
<code>Int64 % Float64</code> <code>Float64 % Int64</code> <code>Float64 % Float64</code>	<b>Float64</b>	$2 \% 3.5 = 2.0$ $2.3 \% 8 = 2.3$ $1.0 \% 5.0 = 1.0$
<code>Int64 % Int64</code>	<b>Int64</b>	$6 \% 3 = 0$

## Nativas

Julia cuenta con una gran variedad de funciones nativas, sin embargo, JOLC únicamente contará con las siguientes funciones nativas:

Nombre	Símbolo o función	Descripción	Ejemplo
<b>Logaritmo común (base 10)</b>	<code>log10()</code>	Devuelve el logaritmo común de cada elemento de la matriz X. La función acepta entradas tanto reales como complejas	<code>log10(100)</code>
<b>Logaritmo con diferente base</b>	<code>log()</code>	Devuelve el logaritmo de un número con una base especificada (base, valor)	<code>log(2,4)</code>

<b>Seno</b>	sin()	Devuelve el seno de los elementos de X. La Sin función opera por elementos en matrices.	sin(45)
<b>Coseno</b>	cos()	Devuelve el coseno de los elementos de X. La Cos función opera por elementos en matrices.	cos(45)
<b>Tangente</b>	tan()	Devuelve el tangente de los elementos de X. La Tan función opera por elementos en matrices.	tan(45)
<b>Raíz Cuadrada</b>	sqrt()	Devuelve la raíz cuadrada de un número x es aquel número y que al ser multiplicado por sí mismo da como resultado el valor x	sqrt(25)

#### 4.3.2. Relacionales

Operador	Descripción
>	Mayor que: Compara ambos lados y verifica si el izquierdo es mayor que el derecho
<	Menor que: Compara ambos lados y verifica si el derecho es mayor que el izquierdo
>=	Mayor o igual que: Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho
<=	Menor o igual que: Compara ambos lados y verifica si el derecho es mayor o igual que el izquierdo
==	Igualación: Compara ambos valores y verifica si son iguales
!=	Distinto: Compara ambos lados y verifica si son distintos

## EJEMPLOS:

Operandos	Tipo resultante	Ejemplos
Int64 [>, <, >=, <=] Float64 Float64 [>, <, >=, <=] Int64 Float64 [>, <, >=, <=] Float64 Int64 [>, <, >=, <=] Int64 String [>, <, >=, <=] String	<b>Bool</b>	4 < 4.3 = true 4.3 > 4 = true 4.3 <= 4.3 = true 4 >= 4 = true "hola" > "hola" = false

### 4.3.3. Lógicas

Los siguientes operadores booleanos son soportados en JOLC. No se aceptan valores missing values ni operadores bitwise.

Operación lógica	Operador
OR	
AND	&&
NOT	!

A	B	A && B	A    B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

## 4.4. Impresión

Para mostrar información en la consola o en los reportes, JOLC cuenta con 2 distintas instrucciones para imprimir dependiendo de lo que deseamos realizar.

- Imprimir en una misma línea. Para eso se utiliza la función para imprimir `print(expresión)`.
- Imprimir con salto de línea. Para eso se utiliza la función para imprimir `println(expresión)`.

```
println("+", "-");           # Imprime + -
print("El resultado de 2 + 2 es $(2 + 2)"); # Imprime el resultado de 2 + 2 es 4
println("$a $(b[1])");       # Imprime el valor de a y el valor de b[1]
```

## 4.5. Asignaciones

Una variable, en JOLC, es un nombre asociado a un valor. Una variable puede cambiar su tipo en cualquier momento.

La asignación se puede realizar de la siguiente forma:

```
ID = Expresión :: TIPO;
ó
ID = Expresión;
```

El sufijo `::TIPO` es opcional. Su función es asegurar que la expresión sea del tipo deseado. En caso la expresión sea distinta al tipo debe marcar un error.

```
x = (3*5)::Int64;      # Correcto
str = "Saludo"::Int64; # ERROR: expected Int64, got String
var1 = true::String;   # ERROR: expected String, got Bool
var = 1234;            # Correcto
```

La palabra reservada 'local' en JOLC se usa para crear una variable de alcance limitado cuyo valor es local al alcance del bloque en el que está definida.

```
# Ejemplo 1: Entornos. Variables globales y locales.
x = (3*5)::Int64;      # 15
str = "Saludo";

function ejemplo()
  global str="Ejemplo"; # JOLC hace referencia a la variable global str
  x = 0;                # JOLC crea una nueva variable local
```

```

for i in 1:5
    local x;    # Creando variable local.
    x = i * 2;  # Gracias a 'local' no hace referencia a la variable de la línea 6
    println(x); # imprime: 2 4 6 8 10
end;
println(x);    # 0 --> la variable nunca fue modificada
end;

ejemplo();

println(x);    # 15
println(str);  # Ejemplo --> Modificada dentro de ejemplo()

```

```

# Ejemplo 2: Aclaraciones de scope
x = 15;
y = 44;

function ejemplo2()
    global y;  #JOLC en la asignación no toma la variable global, crea una nueva.
    y = 5;
    println(x); #JOLC en la llamada a una variable SI busca en el entorno local y luego
en el global.
end;

ejemplo2();

println(x);
println(y);

```

```

# Ejemplo 3: not defined.
x=3;
function ejemplo3()
    for i in 1:5
        local x;
        println(x); # ERROR: no se ha definido un valor para x. Deberán llevar control de
las variables a las que no se les ha asignado un valor. Como en este caso
    end;
end;

ejemplo3();

```



## 4.6. Funciones

### 4.6.1. Creación de funciones

Las funciones en JOLC se crean con la palabra clave *function* seguida del nombre de la función y, entre paréntesis, los parámetros de entrada de la función. El lenguaje original, si permite retornar valores en funciones sin utilizar la instrucción `return`. En JOLC es obligatorio utilizar la instrucción `return` para retornar un valor. En caso no se utilice o se utilice `return` sin valor, la función devolverá nada (el dato `nothing`).

```
function NOMBRE_FUNCION (LISTA_PARAMETROS)
  LISTA_INSTRUCCIONES
end;
```

### 4.6.2. Funciones Nativas

Julia cuenta con una gran variedad de funciones nativas. Sin embargo, JOLC contará con solo unas cuantas de las disponibles en Julia para el manejo de datos, las cuales se detallan a continuación:

- `parse`: recibe una cadena y la convierte al tipo numérico que se le indique.
- `trunc`: convierte un número flotante a un número entero sin redondearlo.
- `float`: convierte un número entero a un número flotante.
- `string`: recibe cualquier tipo y lo convierte en una cadena de caracteres.
- `typeof`: obtiene el tipo del argumento.

También se incluyen las siguientes funciones nativas para arreglos:

- `push`: inserta un nuevo valor al final del arreglo.
- `pop`: elimina y devuelve el último valor de un arreglo.
- `length`: obtiene el tamaño de un arreglo.

```
println(parse(Float64,"3.13159")); # 3.13159
println(trunc(Int64, 3.99999));   # 3
println(float(34));               # 34.0
println(string([1,2,3]));         # "[1,2,3]"
println(typeof(5 * 5));           # Int64
```

### 4.6.3. Llamada a funciones

La llamada a funciones se realiza con el nombre de la función, y entre paréntesis, los parámetros a pasar.

#### 4.6.4. Paso por valor o por referencia

En JOLC, los únicos tipos que son pasados por referencia son los arreglos y struct, por lo que si se modifican dentro de una función también se modificarán fuera. El resto de tipos son pasados por valor.

```
# En este caso el vector [1,2] se transformará en [3,2]
function valores(x)
    x[1] = 3;
end;

x = [1, 2];
valores(x);
print(x);
```

### 4.7. Condicionales

El lenguaje JOLC cuenta con sentencias condicionales, la evaluación condicional permite que porciones de código se evalúen o no se evalúen dependiendo del valor de una expresión booleana. Estos se definen por las instrucciones *if*, *elseif*, *else*.

Consideraciones:

- Las instrucciones *elseif* y *else* son opcionales.
- La instrucción *elseif* se puede utilizar tantas veces como se desee.

```
# Instrucción if
if x == 8
    var1 = (x + 8) :: Int64;
    println(sqrt(var1));
end;

# Instrucción if, elseif, else
if x == 8
    var1 = (x + 8) :: Int64;
    println(sqrt(var1));
elseif x < 8
    var1 = (x / 3) :: Float64;
    println(sin(var1));
else
    println("Error");
end;

# Instrucción if, else
if x == 10
    var2 = (x + 10) :: Int64;
    println(sqrt(var2));
else
    println(sqrt(x+8));
end;
```

## 4.8. Loops

En el lenguaje JOLC existen dos sentencias iterativas, este tipo de sentencias son aquellas que incluyen un bucle sobre una condición, las sentencias iterativas que soporta el lenguaje son las siguientes:

### 4.8.1. Ciclo while

Esta sentencia ejecutará todo el bloque de sentencias solamente si la condición es verdadera, de lo contrario las instrucciones dentro del bloque no se ejecutarán, seguirá su flujo secuencial.

Consideraciones:

- Si la condición es falsa, detendrá la ejecución de las sentencias de la lista de instrucciones.
- Si la condición es verdadera, ejecuta todas las sentencias de su lista de instrucciones.

```
var1 = 0;
while var1 < 10
    println(var1);
    var1 = var1 + 1;
end;
```

### 4.8.2. Ciclo for

Esta sentencia puede iterar sobre un rango de expresiones, cadena de caracteres (*"string"*) o arreglos. Permite iniciar con una variable como variable de control en donde se verifica la condición en cada iteración, luego se deberá actualizar la variable en cada iteración.

Consideraciones:

- Contiene una variable declarativa que se establece como una variable de control, esta variable servirá para contener el valor de la iteración.
- La expresión que evaluará en cada iteración es de tipo rango, string o array. Aunque también se puede especificar mediante una variable.

```
for i in 1:4          # Recorre rango de 1:4
    print(i, " ");    # Únicamente se recorre ascendentemente
end;                 # Imprime 1 2 3 4

for letra in "Hola Mundo!" # Recorre las letras de la cadena
    print(letra, "-");     # Imprime H-o-l-a-M-u-n-d-o-!-
end;

cadena = "OLC2";
for letra in cadena
    print(letra, "-");     # Imprime O-L-C-2
end;
```

```

for animal in ["perro", "gato", "tortuga"]
  println("$animal es mi favorito");
  #= Imprime
    perro es mi favorito
    gato es mi favorito
    tortuga es mi favorito
  =#
end;

arr = [1,2,3,4,5];
for numero in arr[2:4]
  print(numero, " ");    # Imprime 2 3 4
end;

```

### 4.8.3. Sentencias de transferencia

A veces es conveniente terminar un ciclo antes de que la condición sea falsa o detener la iteración de una sentencia loop antes de que se alcance el final del objeto iterable, además también es conveniente saltar unas sentencias de un ciclo en determinadas ocasiones y por para las funciones es necesario el retorno de un valor.

- Break
- Continue
- Return

Consideraciones:

- Se debe validar que la sentencia *break* y *continue* se encuentre únicamente dentro de una sentencia loop.
- Es necesario validar que la sentencia *break* detenga las sentencias asociadas a una sentencia loop.
- Es necesario validar que la sentencia *continue* salte a la siguiente iteración asociada a su sentencia loop.
- Es requerido validar que la sentencia *return* esté contenida únicamente en una función.

#### # Ejemplo break

```

while true
  print(true);    # Imprime solamente una vez true
  break;
end;

```

#### # Ejemplo continue

```

num = 0;
while num < 10
  num = num + 1;
  if num == 5
    continue;
  end;
  print(num);    # Imprime 1234678910
end;

```

```
end;
```

#### **# Ejemplo de return**

```
function funcion()  
    num = 0;  
    while num < 10  
        num = num + 1;  
        if num == 5  
            return 5;  
        end;  
        print(num);  
    end;  
    return 0;  
end;
```

## 4.9. Arreglos

En JOLC, se cuenta este tipo de dato compuesto y mutable. Puede contener cualquier tipo de dato.

Además, toma en cuenta que al tratarse de un tipo mutable, maneja referencias, por ejemplo:

#### **# Prueba de referencias**

```
function valores(x)  
    x[1] = 3;  
    x[3][2]=55;  
end;  
  
arr = [1,2,3,4,5,6];  
x = [1, 2,arr];  
valores(x);  
  
println(x);           # [3, 2, [1, 55, 3, 4, 5, 6]]  
println(arr);         # [1, 55, 3, 4, 5, 6]
```

## 4.10. Structs

Los *structs* son tipos compuestos que se denominan registros, los tipos compuestos se introducen con la palabra clave *struct* seguida de un bloque de nombres de campos, opcionalmente con tipos usando el operador "::". Los *struct* existen de tipos mutables e inmutables que se especifican en las consideraciones.

Consideraciones:

- Los atributos tienen opcionalmente tipos de datos.
- Los atributos sin especificar el tipo, en consecuencia pueden contener cualquier tipo de valor.
- Los objetos *structs* declarados como inmutables no pueden modificar sus atributos después de la construcción.
- Los objetos *structs* declarados como mutables si pueden modificar sus atributos después de la construcción.
- Los *structs* también se pueden utilizar como retorno de una función.
- Las declaraciones de los *structs* se pueden utilizar como expresiones.
- Los atributos se pueden acceder por medio de la notación ".".

### # Struct Inmutable

```
struct Personaje
    nombre;
    edad::Int64;
    descripcion::String;
end;
```

### # Struct Mutable

```
mutable struct Carro
    placa;
    color::String;
    tipo;
end;
```

### # Construcción Struct

```
p1 = Personaje("Fer", 18, "No hace nada");
p2 = Personaje("Fer", 18, "Maneja un carro");
c1 = Carro("090PLO", "gris", "mecanico");
c2 = Carro("P0S921", "verde", "automatico");
```

### # Asignación Atributos

```
p1.edad = 10;           # Error, Struct Inmutable
p2.edad = 20;           # Error, Struct Inmutable
c1.color = "cafe";      # Cambio aceptado
c2.color = "rojo";      # Cambio aceptado
```

### # Acceso Atributo

```
println(p1.edad);       # Imprime 18
println(c1.color);      # Imprime cafe
```

## 4.11. Tabla de Resumen de instrucciones

Instrucción	Descripción
Print	Imprime el resultado de la expresión en una sola línea.
Println	Imprime el resultado de la expresión agregando el salto de línea.
Declaración y asignación	Asigna el resultado de la expresión a una variable existente, si no existe la variable será creada en esta instrucción.
Llamada a funciones	Llama a una función para que sea ejecutada, luego regresa al flujo normal donde se realizó la llamada.
Function	Son secuencias de instrucciones definidas en un bloque para ser ejecutadas en una llamada de la función.
If	La instrucción condicional ejecuta el bloque de instrucciones si la condición es verdadera.
Else If	Si la condición en if es falsa, esta instrucción evaluará una nueva condición si es verdadera ejecuta las instrucciones del bloque.
Else	Si las condiciones en if y elseif son falsas se ejecuta las instrucciones que estén en else.
While	Esta instrucción ejecuta el bloque de instrucciones si la condición es verdadera.
For	La instrucción for puede iterar sobre tipos iterables como lo son rangos, arreglos y cadenas.
Break	Esta instrucción termina la ejecución de una instrucción cíclica antes de que se alcance el final del objeto iterable.
Continue	Esta instrucción salta a la siguiente iteración sin ejecutar las instrucciones restantes.
Return	La instrucción return se utiliza para devolver un resultado en las funciones.

## 5. Reportes generales

### 5.1. Tabla de Símbolos

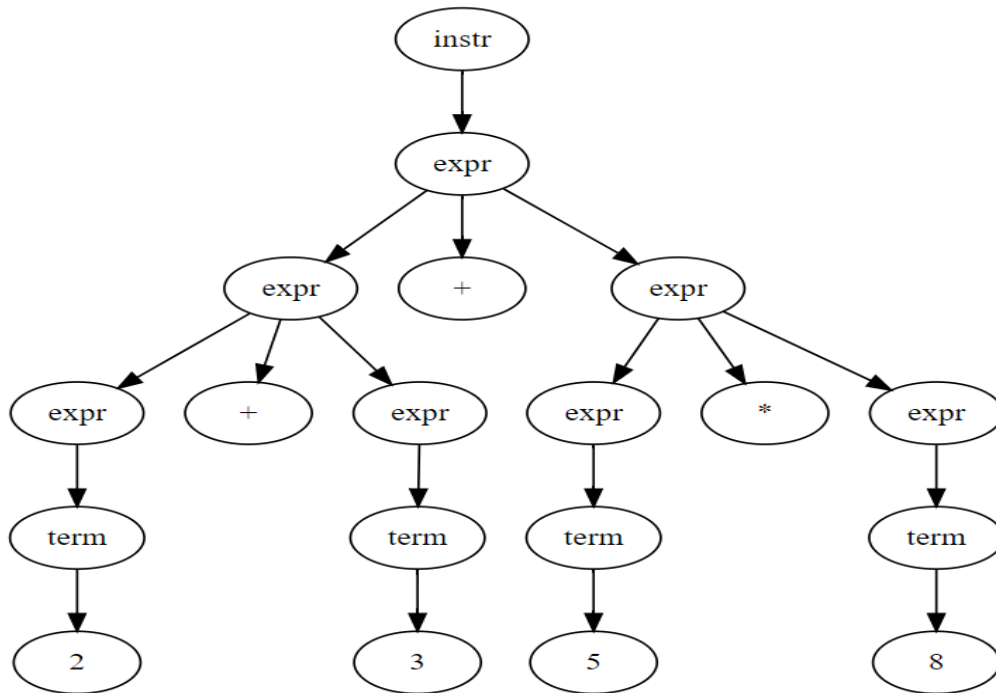
En este reporte se solicita mostrar la tabla de símbolos después de la ejecución del archivo. Se deberán mostrar todas las variables, funciones y struct reconocidas, junto con su tipo y toda la información que el estudiante considere necesaria. Este reporte al menos debe contener la fila y columna de la declaración del símbolo junto con su nombre, tipo y ámbito. En el caso de las funciones, deberá mostrar el nombre de sus parámetros, en caso tenga.

Nombre	Tipo	Ámbito	Fila	Columna
x		valores	2	18
valores	Funcion	Global	2	1
arr	arreglo	Global	6	1
x	arreglo	Global	7	1



## 5.2. Árbol de análisis sintáctico

Deberá mostrar el árbol de análisis sintáctico que se generó al analizar el archivo de entrada de acuerdo con su gramática. El cual es un árbol con las siguientes propiedades: a) la raíz se etiqueta con el símbolo inicial. b) Cada hoja se etiqueta con un terminal, o con  $\epsilon$ . c) Cada nodo interior se etiqueta con un no terminal.



## 5.3. Tabla de errores

Su aplicación deberá ser capaz de detectar y reportar todos los errores **semánticos** que se encuentren durante la ejecución. Su reporte debe contener como mínimo la siguiente información.

- Descripción del error.
- Número de línea donde se encontró el error.
- Número de columna donde se encontró el error.
- Fecha y hora en el momento que se produce un error.

No.	Descripción	Línea	Columna	Fecha y hora
1	El struct Persona no fue declarado	112	15	14/8/2021 20:16
2	El tipo string no puede multiplicarse con un real	80	10	14/8/2021 20:16
3	No se esperaba que la instrucción break estuviera fuera de un ciclo.	1000	5	14/8/2021 20:16

# Entregables y Calificación

Para el desarrollo del proyecto se deberá utilizar un repositorio de GitHub, este repositorio deberá ser privado y tener a los auxiliares como colaboradores.

## 5.4. Entregables

El código fuente del proyecto se maneja en GitHub por lo tanto, el estudiante es el único responsable de mantener actualizado dicho repositorio hasta la fecha de entrega, si se hacen más commits luego de la fecha y hora indicadas no se tendrá derecho a calificación.

- Código fuente y archivos de compilación publicados en un repositorio de GitHub cada uno en una carpeta independiente.
- Enlace al repositorio y permiso a los auxiliares para acceder. Para darle permiso a los auxiliares, agregar estos usuarios al repositorio:
  - ManuelMiranda99
  - ronald1512
  - pablorocad
  - Losajhonny
  - checha18964
- Aplicación web con la funcionalidad del proyecto publicada en Heroku.

## 5.5. Restricciones

- La herramienta para generar los analizadores del proyecto será Python PLY. La documentación se encuentra en el siguiente enlace <https://www.dabeaz.com/ply/>.
- No está permitido compartir código con ningún estudiante. Las copias parciales o totales tendrán una nota de 0 puntos y los responsables serán reportados a la Escuela de Ingeniería en Ciencias y Sistemas.
- El resultado final del proyecto debe ser una aplicación web funcionando en Heroku, no será permitido descargar el repositorio y calificar localmente.
- El desarrollo y entrega del proyecto es individual.

## 5.6. Consideraciones

- Es válido el uso de cualquier Framework para el desarrollo de la aplicación siempre y cuando la aplicación final pueda ser publicada en Heroku.
- El repositorio únicamente debe contener el código fuente empleado para el desarrollo, no deben existir archivos PDF o DOCX.
- El sistema operativo a utilizar es libre.
- Se van a publicar archivos de prueba y sintaxis del lenguaje en el siguiente repositorio: <https://github.com/ManuelMiranda99/JOLC>.
- El lenguaje está basado en Julia (<https://julialang.org/>), por lo que el estudiante es libre de realizar archivos de prueba en esta herramienta, el funcionamiento debería ser el mismo y limitado a lo descrito en este enunciado.

## 5.7. Calificación

- La calificación se realizará dentro de la máquina de los auxiliares, ya que es muy importante que tengan la última versión de su proyecto subida a Heroku y las rutas definidas anteriormente.
- Se probará que el estudiante genere el compilado correcto y que esté siendo ejecutado en Heroku.
- Durante la calificación se realizarán preguntas sobre el código y reportes generados para verificar la autoría de este, de no responder correctamente la mayoría de las preguntas se reportará como copia.
- Se tendrá un máximo de 45 minutos por estudiante para calificar el proyecto.
- La hoja de calificación describe cada aspecto a calificar, por lo tanto, si la funcionalidad a calificar falla en la sección indicada se tendrá 0 puntos en esa funcionalidad y esa nota no podrá cambiar si dicha funcionalidad funciona en otra sección.
- Si una función del programa ya ha sido calificada, esta no puede ser penalizada si en otra sección la función falla o es errónea.
- Los archivos de entrada permitidos en la calificación son únicamente los archivos de pruebas preparados por los tutores.
- Los archivos de entrada podrán ser modificados solamente antes de iniciar la calificación eliminando funcionalidades que el estudiante indique que no desarrolló.
- Los archivos de entrada podrán ser modificados si contienen errores semánticos no descritos en el enunciado o provocados para verificar el manejo y recuperación de errores.

## 5.8. Entrega de proyecto

- La entrega será mediante GitHub, y se va a tomar como entrega el código fuente publicado en el repositorio a la fecha y hora establecidos.
- Cualquier commit luego de la fecha y hora establecidas invalidará el proyecto, por lo que se calificará hasta el último commit dentro de la fecha válida.
- Fecha de entrega:

**Sábado 18 de septiembre hasta las 23:59 P.M.**