

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



**USAC**  
TRICENTENARIA  
Universidad de San Carlos de Guatemala

**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

## Segundo proyecto de laboratorio

<b>Competencias</b>	<b>4</b>
Competencia general	4
Competencias específicas	4
<b>Descripción</b>	<b>5</b>
Descripción General	5
Flujo específico de la aplicación	5
Proceso de Compilación	5
Proceso de Optimización	6
Proceso de Comprobación	6
Proceso de Ejecución	7
Visualización de Reportes	7
<b>Componentes de la aplicación</b>	<b>8</b>
Página de Bienvenida	8
Página de Editor	9
Página de Reportes	10
<b>Sintaxis de JOLC</b>	<b>11</b>
Generalidades	11
Sistema de tipos	11
Tipos básicos	12
Tipos compuestos	12
Reglas	12
Comprobación dinámica	13
División entre cero	13
Índice fuera de los límites	14

Expresiones	15
Aritméticas	15
Multiplicación	16
División	17
Potencia	17
Módulo	17
Relacionales	18
Lógicas	18
Impresión	19
Asignaciones	19
Funciones	20
Creación de funciones	20
Funciones Nativas	21
Llamada a funciones	21
Paso por valor o por referencia	21
Condicionales	21
Loops	22
Ciclo while	22
Ciclo for	23
Sentencias de transferencia	23
Arreglos	24
Structs	25
<b>Generación de Código Intermedio</b>	<b>27</b>
Tipos de dato	27
Temporales	27
Etiquetas	27
Comentarios	28
Saltos	28
Saltos no condicionales	28
Saltos condicionales	29
Asignación a temporales	29
Métodos	30
Llamada a métodos	30
Impresión en consola	30
Estructuras en tiempo de ejecución	31
Stack	31
Heap	32
Acceso y asignación a estructuras en tiempo de ejecución	32
Encabezado	32
Método main	33
Comprobación de código tres direcciones	33
<b>Optimización de Código Intermedio</b>	<b>34</b>
Optimización por mirilla	34

Eliminación de instrucciones redundantes de carga y almacenamiento	34
Regla 1	34
Eliminación de código inalcanzable	35
Regla 2	35
Optimizaciones de flujo de control	35
Regla 3	35
Regla 4	35
Regla 5	36
Simplificación algebraica y reducción por fuerza	36
Regla 6	36
Regla 7	36
Regla 8	36
Optimización por bloque	38
Subexpresiones comunes	38
Regla 1	38
Propagación de copias	39
Regla 2	39
Eliminación de código muerto	39
Regla 3	39
Propagación de constantes	40
Regla 4	40
<b>Reportes generales</b>	<b>41</b>
Reporte de Tabla de Símbolos	41
Reporte de Tabla de errores	41
Reporte de Optimización	41
<b>Manejo de errores</b>	<b>42</b>
Errores semánticos	42
<b>Entregables y Calificación</b>	<b>43</b>
Entregables	43
Restricciones	43
Consideraciones	43
Calificación	44
Entrega de proyecto	45

# 1. Competencias

## 1.1. Competencia general

Que los estudiantes apliquen los conocimientos adquiridos en el curso para la construcción de un compilador 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, la generación de código intermedio y optimización del código intermedio del lenguaje JOLC.
- Que los estudiantes generen una traducción de código de alto nivel a código de tres direcciones.



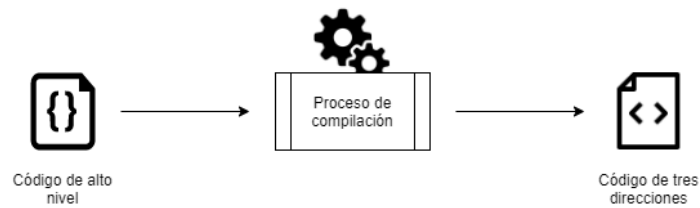


Ilustración 2. Proceso de Compilación.

### 2.2.2. Proceso de Optimización

Un compilador puede introducir secuencias de código que contienen instrucciones innecesarias que consumen el tiempo de ejecución, por tal razón, se deberá aplicar transformaciones de optimización del código generado para producir código más eficiente. El proceso de optimización se debe de hacer en dos pasos las cuales son optimización por mirilla y optimización por bloques.

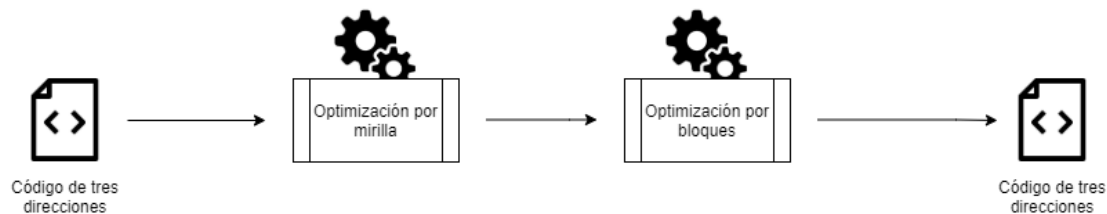


Ilustración 3. Proceso de Optimización.

### 2.2.3. Proceso de Comprobación

Como el código de tres direcciones utiliza sentencias del lenguaje Go, se debe asegurar que el compilador genere el formato correcto antes de su ejecución, por lo cual, se debe de realizar el proceso de comprobación después de generar el código de tres direcciones, después de la optimización por mirilla y después de la optimización por bloques. Así para cumplir con los objetivos del proyecto, por este motivo estará disponible un analizador de código de tres direcciones desarrollado por los tutores para constatar que el código generado y optimizado tenga la sintaxis correcta.

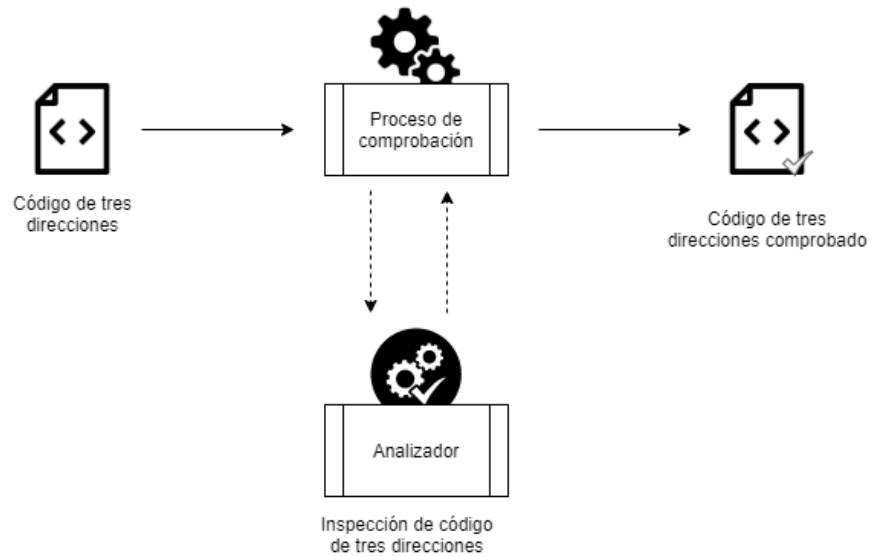


Ilustración 4. Proceso de Comprobación.

#### 2.2.4. Proceso de Ejecución

En el proceso de ejecución recibe como entrada el código de tres direcciones y será ejecutado en un compilador en línea del lenguaje de programación Go, este proceso se puede realizar después del proceso de compilación, después de la optimización por mirilla y después de la optimización por bloques, se ejecutará el código únicamente si el código de tres direcciones tiene el formato correcto.



Ilustración 5. Proceso de Ejecución.

#### 2.2.5. Visualización de Reportes

Luego de haber finalizado el proceso de compilación el usuario podrá consultar el reporte de errores y tabla de símbolos. Asimismo, una vez finalizado el proceso de optimización, el usuario podrá consultar el reporte de optimización.

## 3. Componentes de la aplicación

A continuación, se describen los componentes de la aplicación.

### 3.1. Página de Bienvenida

La aplicación debe de contar con una página principal, en donde servirá para identificar al estudiante con sus datos personales, por lo cual, se deberá mostrar los datos tales como el número de carnet, nombre completo y sección a la que pertenece.



Ilustración 6: Página de bienvenida

La aplicación debe tener una barra de menú que permite la navegación entre las páginas y el correcto funcionamiento de la misma, mediante las siguientes opciones:

- **Home:** Redirecciona a la página principal.
- **Editor:** Redirecciona a la página de editor.
- **Compilador:** Se deberá tener opciones para el correcto funcionamiento del compilador, estas opciones se detallan en la página de editor.
- **Reportes:** Se deberá tener opciones para mostrar los distintos reportes generados por el compilador, estas opciones se detallan en la página de reportes.



Ilustración 7: Barra de menú



## 3.2. Página de Editor

JOLC tendrá una página que contiene un editor de texto que recibirá como entrada el código fuente de alto nivel y así llevar a cabo el proceso de compilación mostrando como resultado el código de tres direcciones en la consola de salida. Y para los procesos de optimización la entrada será el código de tres direcciones mostrando como resultado el código de tres direcciones optimizado en la consola de salida.

Para este editor no hace falta abrir archivos, basta con copiar y pegar la entrada.

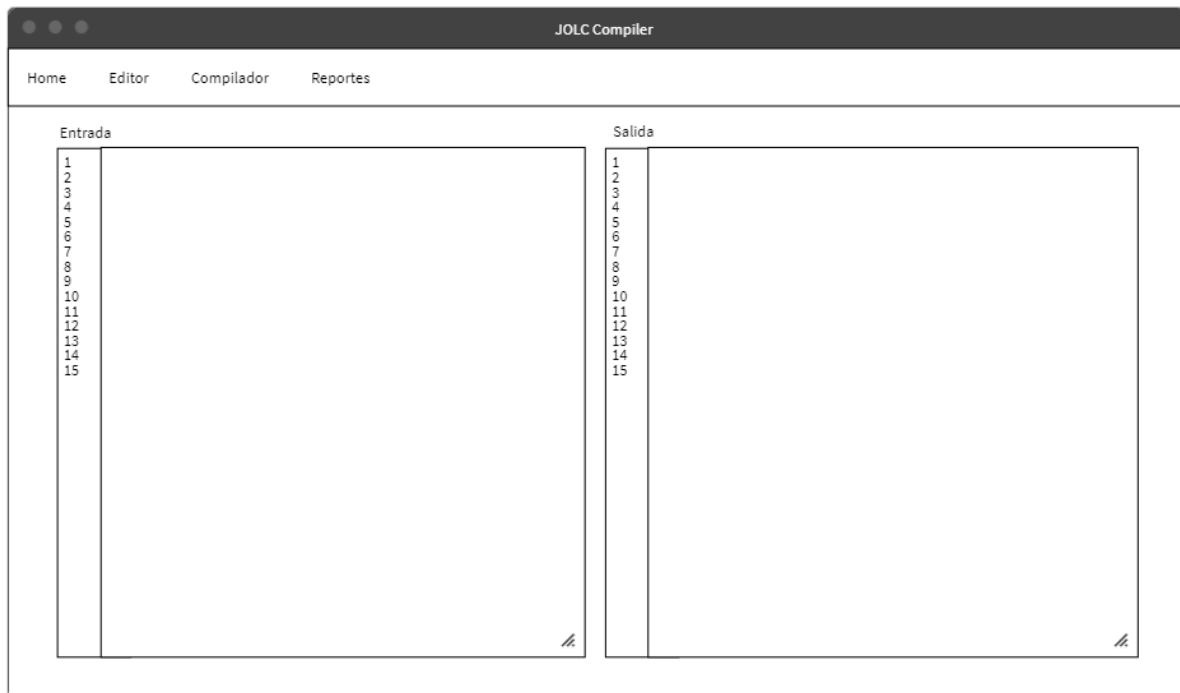


Ilustración 8: Vista de editor

El editor contará con tres opciones que apoyen a la aplicación para realizar los procesos de compilación y optimización, estas opciones son:

- **Compilar:** Realizará el análisis del código fuente para generar el código de tres direcciones.
- **Optimizar por mirilla:** Realizará la optimización por mirilla aplicando las nueve reglas descritas en la sección 6.1.
- **Optimizar por bloques:** Realizará la optimización por bloques aplicando las cuatro reglas descritas en la sección 6.2.

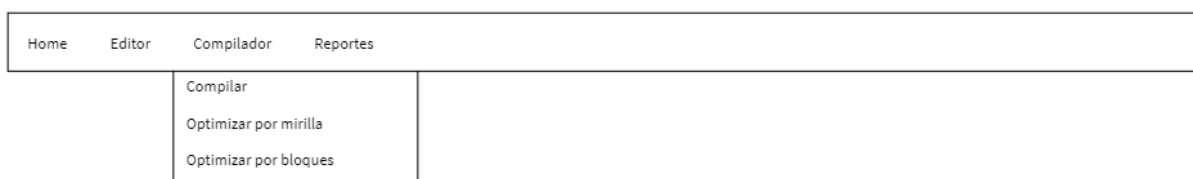
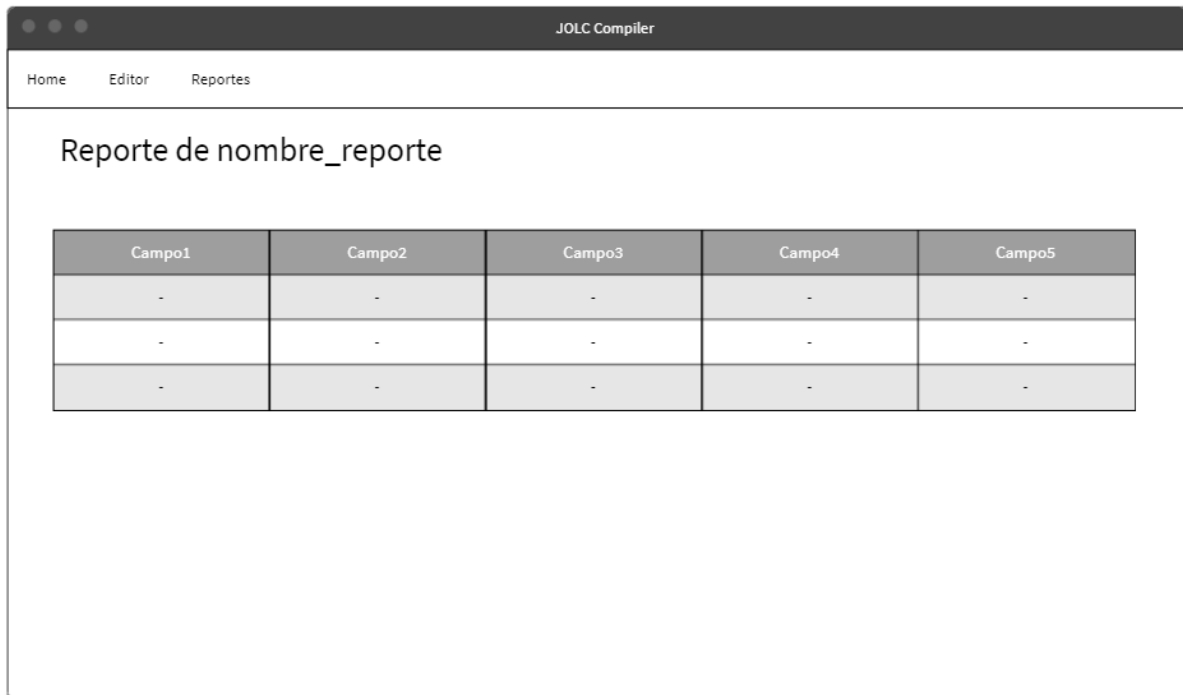


Ilustración 9: Opciones del compilador

### 3.3. Página de Reportes

En esta vista se podrán consultar los reportes de tabla de símbolos y tabla de errores después de la compilación, además se puede consultar el reporte de optimización después de haber optimizado el código de tres direcciones.



Campo1	Campo2	Campo3	Campo4	Campo5
-	-	-	-	-
-	-	-	-	-
-	-	-	-	-

Ilustración 10: Página de consulta de reportes

Para mostrar un reporte en específico, la aplicación contará con tres opciones, se usará la vista de la ilustración 9 como plantilla para mostrar cada reporte:

- **Reporte de tabla de símbolos:** Este reporte mostrará la tabla de símbolos del código fuente generado en el proceso de compilación.
- **Reporte de errores:** Este reporte mostrará los errores encontrados durante el proceso de compilación.
- **Reporte de optimización:** Este reporte mostrará todas las optimizaciones que fueron posible realizar en el código de tres direcciones.



Home	Editor	Compilador	Reportes
			Reporte de tabla de símbolos Reporte de errores Reporte de optimización

Ilustración 11: Opciones 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. Sistema de tipos

El sistema de tipos de JOLC es dinámico, pero obtiene algunas de las ventajas de los sistemas de tipos estáticos al permitir que ciertos valores son de tipos específicos. Esto puede ser de gran ayuda para generar código eficiente, el comportamiento predeterminado en JOLC cuando se omiten los tipos es permitir que los valores sean de cualquier tipo. Por lo tanto, se pueden escribir muchas funciones útiles de JOLC sin usar tipos explícitamente.

Solo los valores, no las variables, tienen tipos; las variables son simplemente nombres ligados a valores, aunque para simplificar podemos decir tipo de una variable como abreviatura de tipo del valor al que se refiere una variable.

Únicamente aceptará los siguientes tipos de datos, cualquier otro no se deberá tomar en cuenta.

### 4.2.1. Tipos básicos

Son tipos que se definen en el compilador y podemos denominarlos como predefinidos en el lenguaje.

Nombre	Descripción	Expresión de tipo
Nulo	Representación de ausencia de valor.	Nothing
Entero	Representación numérica de enteros. Por ejemplo: 3, 2, 1.	Int64
Decimal	Representación numérica de punto flotante. Por ejemplo: 3.2, 45.6.	Float64
Booleano	Representación lógica. Por ejemplo: true o false.	Bool
Carácter	Representación de carácter, se define con comillas simples. Por ejemplo: 'a'.	Char
Cadena	Representación de cadena de texto definida con comillas dobles.	String

### 4.2.2. Tipos compuestos

Los constructores de tipo utilizan los tipos básicos para crear nuevos tipos de datos. Los constructores de tipo que utilizan la mayoría de los lenguajes de programación son arreglos y estructuras.

Nombre	Descripción	Expresión de tipo
Arreglos	Conjunto de valores indexados entre 1 hasta n. Puede almacenar diferentes tipos. Para más información, consulte la sección 4.9.	Array
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

### 4.2.3. Reglas

En la siguiente tabla se detalla el comportamiento de las operaciones aritméticas según los tipos de los operandos, cualquier otra operación con otro tipo de dato se reporta como error.

Este comportamiento es definido como una expresión seguido por un operador aritmético seguido por otra expresión, donde la primera expresión equivale al operando izquierdo y la segunda expresión equivale al operando derecho.

Operador	Operando izquierdo	Operando derecho	Resultado
+, -, %, /	Int64 Float64 Int64 Float64	Int64 Int64 Float64 Float64	Int64 Float64 Float64 Float64
*	Int64 Float64 Int64 Float64 String	Int64 Int64 Float64 Float64 String	Int64 Float64 Float64 Float64 String - concatenación
^	Int64 Float64 Int64 Float64 String	Int64 Int64 Float64 Float64 Int64	Int64 Float64 Float64 Float64 String - multiplicidad

En la siguiente tabla se detalla el comportamiento de las operaciones relacionales y operaciones lógicas según los tipos de los operandos, cualquier otra operación con otro tipo de dato se reporta como error.

El comportamiento del operador not cambia debido a que este solo tiene una expresión asociada, por eso, esa expresión se representa como operando derecho para este ejemplo.

Operador	Operando izquierdo	Operando derecho	Resultado
>, <, >=, <=, ==, !=	Int64 Float64 Int64 Float64 String	Int64 Int64 Float64 Float64 String	Bool
, &&	Bool	Bool	Bool
!	-	Bool	Bool

Como el lenguaje JOLC es dinámico, puede ocurrir que las variables no se les asigne un valor, estas variables trae por defecto un valor nothing que representa a la nada, por lo cual, todas las variables se pueden comparar con el valor nothing por medio de los operadores == y != para verificar si las variables nunca han sido asignadas.

#### 4.2.4. Comprobación dinámica

Existen ciertos casos en los que no es posible comprobar la validez de operaciones en tiempo de compilación, solamente en tiempo de ejecución. En el proyecto se tomarán en cuenta los siguientes casos:

##### División entre cero

Se deberá de realizar la comprobación de división entre cero siempre y cuando se realicen expresiones aritméticas con el operador de división (/) o el operador de módulo (%). Se

deberá realizar la verificación en código de tres direcciones mostrando como mensaje "MathError". Por ejemplo:

Entrada	Salida
a=(55+3)/(3-3);	<pre> T1 = 55 + 3; T2 = 3 - 3; if (T2 != 0) {goto L1}; fmt.Printf("%c", 77); //M fmt.Printf("%c", 97); //a fmt.Printf("%c", 116); //t fmt.Printf("%c", 104); //h fmt.Printf("%c", 69); //E fmt.Printf("%c", 114); //r fmt.Printf("%c", 114); //r fmt.Printf("%c", 111); //o fmt.Printf("%c", 114); //r T3 = 0; // resultado incorrecto goto L2; L1: T3 = T1 / T2; // resultado correcto L2: </pre>

### Índice fuera de los límites

Se deberá realizar la comprobación de índice fuera de los límites, tanto superior como inferior, siempre que se realice un acceso a un arreglo. Se deberá realizar la verificación en código de tres direcciones mostrando como mensaje "BoundsError". Por ejemplo:

Entrada	Salida
<pre> numeros = [1,2,3]; numeros[10]=44; </pre>	<pre> T2 = 10; // índice al que desea acceder if (T2 &lt; 1) {goto L1}; // 1 es el límite inferior del arreglo if (T2 &gt; 3) {goto L1}; // 3 es el límite superior del arreglo goto L2; L1: fmt.Printf("%c", 66) //B fmt.Printf("%c", 111) //o fmt.Printf("%c", 117) //u fmt.Printf("%c", 110) //n fmt.Printf("%c", 100) //d fmt.Printf("%c", 115) //s fmt.Printf("%c", 69) //E fmt.Printf("%c", 114) //r fmt.Printf("%c", 114) //r fmt.Printf("%c", 111) //o fmt.Printf("%c", 114) //r // No continúa con la instrucción goto L3; L2: </pre>

	// Continúa con la instrucción L3:
--	---------------------------------------

Consideraciones:

- En caso se produzca un error al intentar ejecutar una instrucción, esta se debe omitir y continuar con la siguiente instrucción. En caso se produzca un error en una expresión, esta debe resultar con valor 0. Todo esto luego de imprimir en consola el texto solicitado según el caso.

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

uppercase(String)	<b>String</b>	<pre> animal = "Tigre"; println(uppercase(animal)); #TIGRE </pre>
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
Int64 - Float64 Float64 - Int64 Float64 - Float64	<b>Float64</b>	$2 + 3.3 = -1.3$ $2.3 + 8 = -5.7$ $1.2 + 5.4 = -4.2$
Int64 - Int64	<b>Int64</b>	$2 + 3 = -1$

## Multiplicación

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

Operandos	Tipo resultante	Ejemplos
Int64 * Float64 Float64 * Int64 Float64 * Float64	<b>Float64</b>	$2 * 3.3 = 6.6$ $2.3 * 8 = 18.4$ $1.2 * 5.4 = 6.48$
Int64 * Int64	<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>	$6 \% 3 = 0$

### 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	<b>Bool</b>	4 < 4.3 = true
Float64 [>, <, >=, <=, ==, !=] Int64		4.3 > 4 = true
Float64 [>, <, >=, <=, ==, !=] Float64		4.3 <= 4.3 = true
Int64 [>, <, >=, <=, ==, !=] Int64		4 >= 4 = true
String [>, <, >=, <=, ==, !=] String		"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.
- `string`: recibe cualquier tipo y lo convierte en una cadena de caracteres.

También se incluyen las siguientes funciones nativas para arreglos:

- `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]"
```

#### 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;
```

#### # 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**

## 5. Generación de Código Intermedio

El código intermedio es una representación intermedia del programa fuente que se ingresó en JOLC. Esta representación intermedia se realizará en código en tres direcciones, las cuales son secuencias de pasos de programa elementales.

En el proyecto se utilizará las sentencias del lenguaje Go para escribir el código tres direcciones, manejando este lenguaje con limitaciones para que se apliquen correctamente los conceptos de generación de código tres direcciones aprendido en la clase magistral. El código tres direcciones que genera JOLC se detalla en esta sección.

**No está permitido el uso de toda función o característica del lenguaje Go no descrita en este apartado. Se utilizará una herramienta de análisis para verificar que el código de tres direcciones generado tenga el formato correcto.**

### 5.1. Tipos de dato

El lenguaje a compilar solo acepta tipos de datos numéricos, es decir, tipos int y float. Consideraciones:

- No está permitido el uso de otros tipos de datos como cadenas o booleanos.
- El uso de arreglos no está permitido, únicamente para las estructuras heap y stack que se explican con más detalle más adelante.
- Por facilidad, se recomienda trabajar todas las variables de tipo float.

### 5.2. Temporales

Los temporales serán creados por el compilador en el proceso de generación de código de tres direcciones. Estas serán variables de **tipo float**. El identificador asociado a un temporal puede ser de la siguiente manera.

```
t[0-9]+  
Ej.  
t1  
t145
```

### 5.3. Etiquetas

Las etiquetas son identificadores únicos que indican una posición en el código fuente, estas mismas serán creadas por el compilador en el proceso de generación de código en tres direcciones. El identificador asociado a una etiqueta puede ser de la siguiente manera.

```
L[0-9]+  
Ej.  
L1  
L21
```

Las operaciones aritméticas contarán con:

- Resultado
- Argumento 1
- Argumento 2
- Operador

Operación	Símbolo	Ejemplo
Suma	+	t1=t0+1
Resta	-	t2=50-12
Multiplicación	*	t33=5*5
División	/	t67=4/1
Módulo	%	t44=4 % 2

## Comentarios

Para llevar un mejor control de las instrucciones que se realizan dentro del código tres direcciones se recomienda el uso de comentarios donde se podrá definir el flujo de cada bloque de código, los comentarios se definen de la siguiente manera:

- Comentarios de una línea. Inician con un conjunto de barras diagonales (//) y continúan hasta el final de la línea.
- Comentarios de múltiples líneas. Inician con los símbolos “/\*” y finalizan con los símbolos “\*/”

## 5.4. Saltos

Para definir el flujo que seguirá el programa se contará con bloques de código, estos bloques están definidos por etiquetas. La instrucción que indica que se realizará un salto hacia una etiqueta es la palabra reservada “goto”.

En el proyecto se utilizarán los dos formatos de saltos que son:

- Condicional. Se realiza una evaluación para determinar si se realiza el salto.
- No condicional. Realiza el salto sin realizar una evaluación.

### 5.4.1. Saltos no condicionales

El formato de saltos no condicionales contará únicamente con una instrucción **goto** que indicara una etiqueta destino específica, en la cual se continúa con la ejecución del programa.

//Ejemplo de salto no condicional

```
goto L1
fmt.Printf("%c", 64) //código inalcanzable
L1:
t2 = 100 + 5
```

### 5.4.2. Saltos condicionales

El formato de los saltos condicionales utilizará la instrucción `if` del lenguaje Go donde se realizará un salto a una etiqueta donde se encuentre el código a ejecutar si la condición es verdadera, seguida de otro salto a una etiqueta donde están las instrucciones si la condición no se cumple.

Las instrucciones `if` tendrán como condición una expresión relacional, dichas expresiones se definen en la siguiente tabla:

Operación	Símbolo	Ejemplo
Menor que	<	t3<4
Mayor que	>	t6>44
Menor o igual que	<=	t55<=50
Mayor o igual que	>=	t99>=100
Igual que	==	t23==t44
Diferente que	!=	t34!=99

```
//Ejemplo de saltos condiciones
If (10 == 10) {goto L1}
goto L2
L1:
//código si la condición es verdadera
L2:
//código si la condición es falsa
```

## 5.5. Asignación a temporales

La asignación nos va a permitir cambiar el valor de los temporales, para lograrlo se utiliza el operador igual, este permite una asignación directa o con una expresión.

```
//Entrado código alto nivel
print(1+2*5);

//Salido código en tres direcciones en lenguaje Go
t1 = 2 * 5
t2 = 1 + 1
```

```
fmt.Printf("%d", int(t2))
```

## 5.6. Métodos

Estos son bloques de código a los cuales se accede únicamente con una llamada al método.

```
//Definición de métodos
func x() {
    goto L0
    fmt.Printf("%d", int(100))
L0:
    return
}
```

Consideraciones:

- No está permitido el uso de parámetros en los métodos. Debe utilizar el stack para el paso de parámetros.
- Al final de cada método se debe incluir la instrucción "return".

## 5.7. Llamada a métodos

Esta instrucción nos permite invocar a los métodos. Al finalizar su ejecución se retorna el control al punto donde fue llamada para continuar con las siguientes instrucciones.

```
func funcion1(){
    fmt.Printf("%d", int(100))
    return
}

func main(){
    // INSTRUCCIONES DE MAIN
    funcion1();                // Llamada a método funcion1
    // DESPUÉS DE EJECUTAR funcion1 REGRESA A MAIN
}
```

## 5.8. Impresión en consola

Su función principal es imprimir en consola un valor, el primer parámetro que recibe la función es el formato del valor a imprimir, y el segundo es el valor en sí.

La siguiente tabla lista los parámetros permitidos para el proyecto:

Parámetro	Acción
%c	Imprime el carácter del identificador, se

	basa según el código ASCII.
%d	Imprime valores enteros. El segundo parámetro debe ser una conversión explícita de int.
%f	Imprime valores con punto decimal.

```
fmt.Printf("%d", int(100)) // Imprime 100
fmt.Printf("%c", 36)     // Imprime $
fmt.Printf("%f", 32.2)   // Imprime 32.200000
```

## 5.9. Estructuras en tiempo de ejecución

El proceso de compilación genera el código en tres direcciones, el cual se ejecutará en un compilador de GO separado. En el código de tres direcciones no existen cadenas, operaciones complejas, llamadas a métodos con parámetros y otras características que sí están presentes en los lenguajes de alto nivel.

En el proyecto se utilizarán las siguientes dos estructuras:

- Stack (pila)
- Heap (montículo)

Estas estructuras se utilizarán para almacenar los valores que sean necesarios durante la ejecución.

### 5.9.1. Stack

También conocido como pila de ejecución, es una estructura que se utiliza para guardar los valores de las variables locales, así como también los parámetros y el valor de retorno de las funciones en alto nivel.

Esta estructura utilizará un apuntador llamado "Stack Pointer", que se identifica con el nombre **P**, este valor va cambiando conforme se ejecute el programa, y su manejo debe ser cuidadoso para no corromper espacios de memoria ajenos al método que se está ejecutando, su asignación se realizará de la misma manera que se realizan las asignaciones temporales.

```
var stack [1000000]float64 // Stack
var P float64              // Stack Pointer

P = P + 5; // Cambio de ámbito
P = P - 5; // Regreso a ámbito

stack[int(P)] = 10
t1 = stack[int(P)]
```

### 5.9.2. Heap

También conocido como montículo, es una estructura de control del entorno de ejecución encargada de guardar las referencias a las cadenas, arreglos y estructuras. Esta estructura también cuenta con un apuntador que se identifica con el nombre **H**.

A diferencia del apuntador **P**, este apuntador no decrece, sino que sigue aumentando su valor, su función es apuntar a la primera posición de memoria libre dentro del heap.

```
var heap [100000]float64 // Heap
var H float64           // Heap Pointer

H = H + 1;
T1 = H

heap[int(P)] = 10
t1 = heap[int(P)]
```

Consideraciones:

- Al guardar cadenas, cada espacio debe ser ocupado por únicamente un carácter representado por su código ASCII.
- El heap solamente crece, nunca reutiliza espacios de memoria.

### 5.9.3. Acceso y asignación a estructuras en tiempo de ejecución

Para realizar las asignaciones y el acceso a estas estructuras, se debe respetar el formato de código de 3 direcciones:

- La asignación a las estructuras se debe realizar por medio de un temporal o un valor puntual, no es permitido el uso de operaciones aritméticas o lógicas para la asignación a estas estructuras.
- No se permite la asignación a una estructura mediante el acceso a otra, por ejemplo "Stack[0] = Heap[100]".

```
//Asignación
Heap[int(H)] = t1
<código>
Stack[int(t2)] = 150
//Acceso
t10 = Heap[int(t10)]
t20 = Stack[int(t150)]
```

## 5.10. Encabezado

En esta sección se definirán todas las variables y estructuras a utilizar para que al momento de ejecutar el código de tres direcciones funcione correctamente, puesto que, es necesario el uso de la librería fmt para la instrucción de imprimir y la declaración de variables antes de usar alguna variable en el código. Únicamente en esta sección se permite el uso de



declaraciones en Go, no es permitido realizar declaraciones dentro de métodos. El encabezado debe ser generado junto con el código tres direcciones para hacer uso de los temporales y las estructuras necesarias.

La estructura del encabezado es la siguiente:

```
package main
import ( "fmt" )           // importar para el uso de printf

var stack [1000]float64    // estructura Stack
var heap [1000]float64    // estructura Heap
var P, H float64          // declaración de Stack y heap pointer
var t1, t2, t3 float64    // declaración de temporales
```

Consideraciones:

- No es permitido el uso de otras librerías ajenas a “fmt”
- Todas las declaraciones de temporales se deben encontrar en el encabezado
- El tamaño que se le asigne al stack y heap queda a discreción del estudiante. Tomar en cuenta que el heap únicamente aumenta, por lo que el tamaño de este debe ser grande.

## 5.11. Método main

Este es el método donde iniciará la ejecución del código traducido. Su estructura es la siguiente:

```
func main(){
    // instrucciones
}
```

## 5.12. Comprobación de código tres direcciones

Una vez generado el código tres direcciones este será ingresado a un analizador de la sintaxis para corroborar que el código generado sea correcto y no se encuentre código diferente al explicado anteriormente, una vez analizado se procederá a ejecutar el código en tres direcciones en un compilador de GO para obtener el resultado esperado.

La herramienta que utilizarán los estudiantes para comprobar que estén generando el código de tres direcciones con la sintaxis correcta estará disponible a partir del 10 de octubre, los tutores serán los encargados de indicar el enlace donde estará publicado.

## 6. Optimización de Código Intermedio

JOLC deberá aplicar transformaciones de optimización al código de tres direcciones como parte de su compilación para producir código más eficiente, estas optimizaciones se realizarán mediante la optimización por mirilla y optimización por bloques que se describen a continuación.

### 6.1. Optimización por mirilla

Este método consiste en utilizar una ventana deslizable que se mueve a través del código de tres direcciones, esta ventana es conocida como la mirilla, por el cual se toman las instrucciones dentro de la mirilla y se sustituyen en secuencias equivalentes que sea de menor longitud y más rápido posible que el bloque original.

El proceso de mirilla permite que por cada optimización realizada se puedan obtener mejores beneficios, por lo que, es necesario efectuar varias pasadas del código fuente para obtener una mayor optimización, de tal forma, para el proyecto se debe de aplicar 10 pasadas en el código de tres direcciones.

La mirilla deberá iniciar con un tamaño de 20 líneas. Y se debe de revisar lo siguiente: Si en una pasada se encuentra más de una optimización, entonces el tamaño de la mirilla debe ser la misma para la siguiente pasada, en caso contrario si en la pasada no existe ninguna optimización el tamaño de la mirilla deberá aumentar 20 líneas más para la siguiente pasada. Esto se deberá de realizar hasta llegar a la última pasada.

Los tipos de transformación para realizar la optimización por mirilla serán los siguientes:

- Eliminación de instrucciones redundantes de carga y almacenamiento.
- Eliminación de código inalcanzable.
- Optimizaciones de flujo de control.
- Simplificación algebraica y reducción por fuerza.

#### 6.1.1. Eliminación de instrucciones redundantes de carga y almacenamiento

##### 6.1.1.1. Regla 1

Se puede eliminar la instrucción de almacenamiento si existe la primera instrucción de la forma  $a = b$ , posteriormente existe la segunda instrucción de la forma  $b = a$ , y entre ambas instrucciones no tuviera una etiqueta entonces se puede eliminar la segunda instrucción.

Ejemplo	Optimización
$t2 = b$ $b = t2$	$t2 = b$

### 6.1.2. Eliminación de código inalcanzable

Consistirá en eliminar las instrucciones que nunca serán utilizadas. Por ejemplo, instrucciones que estén luego de un salto condicional, el cual direcciona el flujo de ejecución a otra parte y nunca llegue a ejecutar las instrucciones posteriores al salto condicional. Las reglas aplicables son las siguientes:

#### 6.1.2.1. Regla 2

Si existen instrucciones sin etiqueta después de un salto incondicional, se podrá eliminar las instrucciones siempre y cuando no exista una etiqueta entre el código.

Ejemplo	Optimización
goto L1 <instrucciones> L1:	L1:

### 6.1.3. Optimizaciones de flujo de control

Con frecuencia los algoritmos de generación de código intermedio producen saltos hacia saltos, saltos condicionales hacia saltos condicionales o saltos condicionales hacia saltos. Los saltos innecesarios podrán eliminarse, con las siguientes reglas:

#### 6.1.3.1. Regla 3

En un salto condicional, si existe un salto inmediatamente después de sus etiquetas verdaderas se podrá reducir el número de saltos negando la condición, cambiando el salto condicional hacia la etiqueta falsa  $L_f$ , eliminando el salto innecesario a  $L_f$  y quitando la etiqueta  $L_v$ .

Ejemplo	Optimización
if a == 10 {goto L1} goto L2 L1: <instrucciones1> L2:	if a != 10 {goto L2} <instrucciones1> L2:

#### 6.1.3.2. Regla 4

Si tenemos un salto incondicional hacia una etiqueta  $L_x$ , seguidamente existen instrucciones y si inmediatamente después se tiene una etiqueta  $L_x$  en donde existe un salto incondicional hacia una etiqueta  $L_y$ , entonces el primer salto incondicional se debe cambiar a la etiqueta  $L_y$ .

Ejemplo	Optimización
goto L1 <instrucciones>	goto L2 <instrucciones>

L1: goto L2	L1: goto L2
----------------	----------------

#### 6.1.3.3. Regla 5

Si existe un salto incondicional hacia una etiqueta  $Lx$  de la forma  $if < cond > \{goto Lx\}$ , seguidamente existen instrucciones y si inmediatamente después se tiene una etiqueta  $Lx$  en donde existe un salto incondicional hacia una etiqueta  $Ly$ , entonces el primer salto incondicional se debe cambiar a la etiqueta  $Ly$ .

Ejemplo	Optimización
if a < b {goto L1} <instrucciones> L1: goto L2	if a < b {goto L2} <instrucciones> L1 goto L2

### 6.1.4. Simplificación algebraica y reducción por fuerza

#### 6.1.4.1. Regla 6

Se elimina la instrucción si una variable se asigna una expresión con operaciones de suma/resta con 0 o multiplicaciones/divisiones con 1 a la misma variable.

Ejemplo	Optimización
x = x + 0 x = x - 0 x = x * 1 x = x / 1	//Se elimina la instrucción //Se elimina la instrucción //Se elimina la instrucción //Se elimina la instrucción

#### 6.1.4.2. Regla 7

Es aplicable la eliminación de instrucciones con operaciones de variable distinta a la variable de asignación y una constante, si las operaciones son sumas/restas con 0 y multiplicaciones/divisiones con 1, la instrucción se transforma en una asignación.

Ejemplo	Optimización
x = y + 0 x = y - 0 x = y * 1 x = y / 1	x = y x = y x = y x = y

#### 6.1.4.3. Regla 8

Se deberá realizar la eliminación de reducción por fuerza para sustituir por operaciones de alto costo por expresiones equivalentes de menor costo.

Ejemplo	Optimización
---------	--------------

$x = y * 2$ $x = y * 0$ $x = 0 / y$	$x = y + y$ $x = 0$ $x = 0$
---	-----------------------------------

## 6.2. Optimización por bloque

Por lo regular, el código fuente tiene una serie de instrucciones que se ejecutan siempre en orden y están conformadas por bloques básicos del código.

Un bloque básico es una parte de código, en donde las instrucciones se ejecutan secuencialmente, por el cual, el flujo de control solo puede entrar en la primera instrucción y puede salir del bloque por una bifurcación o seguir con las siguientes instrucciones.

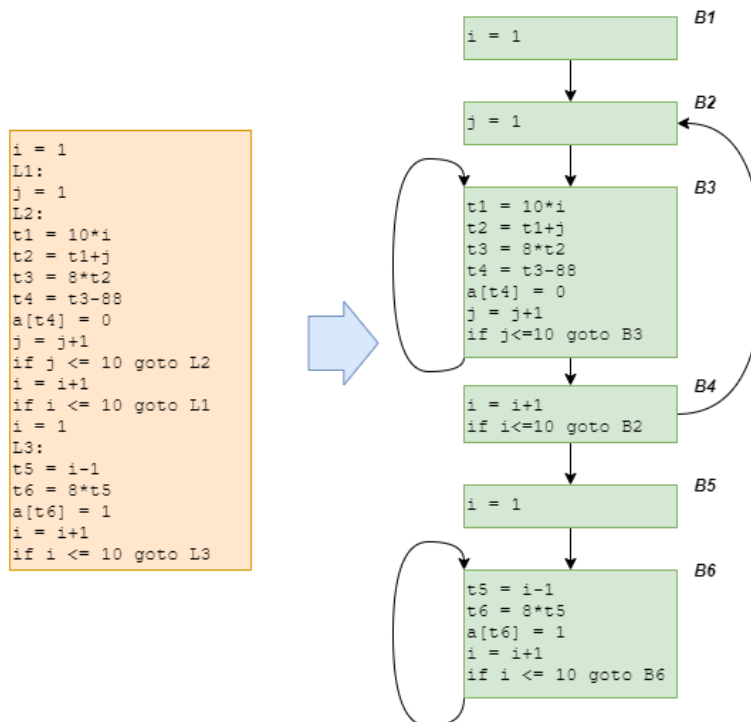


Ilustración 12. Grafo de bloques básicos.

Se debe de realizar la optimización sobre el código generado de las siguientes maneras:

- A nivel local: Consiste en optimizar el código de un bloque básico, se debe de realizar bloque por bloque.
- A nivel global: Consiste en optimizar el código de todos los bloques básicos, se debe de buscar candidatos posibles de optimización para aplicar las reglas.

Los tipos de transformación para realizar la optimización por bloque serán los siguientes:

- Subexpresiones comunes.
- Propagación de copias.
- Eliminación de código muerto.
- Propagación de constantes.

### 6.2.1. Subexpresiones comunes

#### 6.2.1.1. Regla 1

Consiste en buscar expresiones que se repiten y analizar si vale la pena reemplazarlas o eliminarlas. Considerando el siguiente ejemplo, `t6` tiene una subexpresión común con `t4`,

por lo cual, se reemplaza  $t6 = t2 * t3$  por la siguiente asignación  $t6 = t4$ , si los valores de la expresión de  $t4$  no cambian.

Ejemplo	Optimización
$t1 = 6 / 2$ $t2 = 3 * t1$ $t3 = 4 - 2$ $t4 = t2 * t3$ $t5 = 2 + 5$ $t6 = t2 * t3$	$t1 = 6 / 2$ $t2 = 3 * t1$ $t3 = 4 - 2$ $t4 = t2 * t3$ $t5 = 2 + 5$ $t6 = t4$

## 6.2.2. Propagación de copias

### 6.2.2.1. Regla 2

Debido a que el algoritmo normal para eliminar subexpresiones comunes las introduce, es necesario aplicar esta regla. La propagación de copias se relaciona con la asignación  $u = v$ , conocidas como instrucciones de copia o simplemente copias, la idea de la transformación por propagación de copias es utilizar  $v$  para  $u$ , siempre que sea posible después de la instrucción de copia  $u = v$ .

Ejemplo	Optimización
$t1 = 6 / 2$ $t2 = 3 * t1$ $t3 = 4 - 2$ $t4 = t2 * t3$ $t5 = 2 + 5$ $t6 = t4$ $t7 = t6 + t5$	$t1 = 6 / 2$ $t2 = 3 * t1$ $t3 = 4 - 2$ $t4 = t2 * t3$ $t5 = 2 + 5$ $t6 = t4$ $t7 = t4 + t5$

## 6.2.3. Eliminación de código muerto

### 6.2.3.1. Regla 3

Una variable está viva en un punto en el programa, si su valor puede utilizarse más adelante; en caso contrario, está muerta en ese punto. La propagación de copias es que a menudo convierte la instrucción de copia en código muerto como es el caso de  $t6$ .

Ejemplo	Optimización
$t1 = 6 / 2$ $t2 = 3 * t1$ $t3 = 4 - 2$ $t4 = t2 * t3$ $t5 = 2 + 5$ $t6 = t4$ $t7 = t6 + t5$	$t1 = 6 / 2$ $t2 = 3 * t1$ $t3 = 4 - 2$ $t4 = t2 * t3$ $t5 = 2 + 5$ $t7 = t4 + t5$

## 6.2.4. Propagación de constantes

### 6.2.4.1. Regla 4

Desde que se asigna a una variable un valor constante hasta la siguiente asignación, se considera a la variable equivalente a la constante.

Ejemplo	Optimización
$t1 = 3.14$ $t2 = t1 / 180$	$t2 = 3.14 / 180$



## 7. Reportes generales

### 7.1. Reporte de Tabla de Símbolos

En este reporte se solicita mostrar la tabla de símbolos después de la compilación de la entrada. 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	Función	Global	2	1
arr	arreglo	Global	6	1
x	arreglo	Global	7	1

### 7.2. Reporte de Tabla de errores

Su aplicación deberá ser capaz de detectar y reportar todos los errores semánticos que se encuentren durante la compilación. 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

### 7.3. Reporte de Optimización

Este reporte mostrará las reglas de optimización que fueron aplicadas sobre el código intermedio. Se debe indicar el tipo de optimización utilizada y la sección. Como mínimo se solicita la siguiente información:

- Tipo de optimización (Mirilla o por bloques)
- Regla de optimización aplicada
- Expresión original
- Expresión optimizada
- Fila

## 8. Manejo de errores

### 8.1. Errores semánticos

El compilador deberá ser capaz de detectar todos los errores semánticos que se encuentren durante el proceso de compilación. Todos los errores se deberán de recolectar y se mostrará un reporte de errores antes mencionado.

Un error semántico es cuando la sintaxis es la correcta, pero la lógica no es la que se pretendía, por eso, la recuperación de errores semánticos será de ignorar la instrucción en donde se generó el error.

Si se detecta cualquier tipo de error semántico el estudiante deberá descartar la instrucción completa, Un error semántico se dará por ejemplo al intentar usar una variable que no ha sido declarada.

```
...  
int numero1 = numero2 + 10;  
...
```

Como se ve en ejemplo la variable "numero2" no ha sido declarada por lo que se mostrará un error semántico, la forma en que se manejan los errores semánticos consistirá en descartar la instrucción la cual contiene el error y se reportará el error de en el reporte de errores antes mencionado.

## 9. 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.

### 9.1. 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.

### 9.2. 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.

### 9.3. 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.

## 9.4. 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.
- Durante la calificación se realizará un análisis en una herramienta desarrollada por los tutores, para verificar que se utilicen únicamente las instrucciones definidas en este enunciado. Esta herramienta será compartida a los estudiantes.

## 9.5. 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.
- La fecha de entrega será el domingo 7 de noviembre, tendrán un lapso de 12 horas para realizar la entrega del proyecto, el proceso de entrega inicia a las 8:00 y termina a las 20:00. Cualquier inconveniente notificar a su tutor encargado para poder brindar apoyo para el proceso.

**Domingo 7 de noviembre en horario de 8:00 a 20:00**