Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Primer semestre 2025
Catedráticos: Ing. Bayron López, Ing. Edgar Saban e Ing. Luis Espino
Tutores Académicos: Henry Mendoza, Ronaldo Posadas y Damián Peña

GoLight

1. Competencias

1.1. Competencia General

Capacidad para desarrollar un compilador funcional para un lenguaje de programación, aplicando los principios fundamentales de la teoría de compiladores. Esto incluye:

Análisis Léxico: Generación de un analizador léxico para la identificación y clasificación de tokens.

Análisis Sintáctico: Implementación de un analizador sintáctico que valide la estructura gramatical del código.

Análisis Semántico: Aplicación de reglas semánticas para garantizar la coherencia del lenguaje.

Generación de Código: Transformación del código fuente en una representación ejecutable, generando código ensamblador ARM64 y asegurando su correcta ejecución en entornos como QEMU y Raspberry Pi.

1.2. Competencias Específicas

Uso de ANTLR: Implementación de analizadores léxicos y sintácticos mediante ANTLR en un entorno de desarrollo basado en C#.

Estructuras y Herramientas Adecuadas: Para la construcción del compilador, se seleccionarán e implementarán las estructuras de datos y herramientas más apropiadas, garantizando eficiencia y claridad en el procesamiento del código. Esto incluye:

- Definición de reglas gramaticales mediante ANTLR.
- Uso de árboles de sintaxis abstracta (AST) para representar la estructura del código fuente.
- Implementación de estrategias de recorrido del AST para la evaluación de expresiones e instrucciones.
- Generación de código ensamblador ARM64 como salida del compilador.
- Preparación del código para su ejecución en entornos embebidos y virtualizados.

2. Descripción

El proyecto consiste en el desarrollo de un compilador para el lenguaje de programación GoLight, un lenguaje diseñado con una sintaxis inspirada en Go, pero adaptado para explorar conceptos fundamentales de los compiladores.

Como parte del proyecto, se requiere también el desarrollo de una interfaz funcional que permita a los usuarios crear, editar, y compilar código escrito en GoLight. Esto implica la definición y construcción de una gramática que describa de forma precisa la sintaxis del lenguaje GoLight, permitiendo la generación automática del análisis y asegurando el correcto procesamiento del código fuente.

Los analizadores léxico y sintáctico del compilador deberán ser generados mediante la herramienta ANTLR. Esto implica la definición y construcción de una gramática que describa de forma precisa la sintaxis del lenguaje GoLight, permitiendo la generación automática del análisis y asegurando el correcto procesamiento del código fuente. Adicionalmente, la evaluación semántica y un sistema robusto de manejo de errores, las cuales serán implementadas utilizando el lenguaje de programación C#.

El proyecto fundamentalmente requiere la generación de código ensamblador **ARM64**, lo que permitirá a los estudiantes comprender el proceso de traducción de un lenguaje de alto nivel a instrucciones de bajo nivel en una arquitectura de conjunto de instrucciones reducido (RISC). A través de esta implementación, los estudiantes adquirirán un entendimiento profundo de los principios de la compilación, desde el análisis hasta la ejecución del código en QEMU o una Raspberry Pi.

2.1. Componentes de la Interfaz

2.1.1. Editor

El editor formará parte del entorno de desarrollo y proporcionará funcionalidades esenciales para la escritura y gestión del código fuente. Su función principal será permitir el ingreso y la edición del código en GoLight. Deberá permitir la apertura de múltiples archivos y mostrar la línea actual en la que se encuentra el usuario.

El diseño del editor de texto en la interfaz de usuario (GUI) quedará a discreción del estudiante.

2.1.2. Funcionalidades

Las funcionalidades básicas que se solicitan implementar, serán las siguientes:

Crear archivos: El editor deberá ser capaz de crear archivos en blanco.
Abrir archivos: El editor deberá abrir archivos **.glt**
Guardar archivo: El editor deberá guardar el estado del archivo en el que se estará trabajando con extensión **.glt**

2.1.3. Herramientas

Compilar: hará el llamado al compilador, el cual se hará cargo de realizar los análisis léxico, sintáctico y semántico, además de traducir todas las sentencias a su equivalente en ARM64.

2.1.4. Reportes

Reporte de Errores: Se mostrarán todos los errores encontrados al realizar el análisis léxico, sintáctico y semántico.

Reporte de Tabla de Símbolos: Se mostrarán todas las variables, métodos y funciones que han sido declarados dentro del flujo del programa, así como el entorno en el que fueron declarados.

2.1.5. Consola

La consola es un área especial dentro del IDE que permite visualizar las notificaciones, errores, advertencias, así como el resultado de la traducción que se produjeron durante el proceso de análisis de un archivo de entrada.

Se recomienda la utilización de un framework de desarrollo web para generar de forma rápida y práctica la interfaz del proyecto.

2.2. Flujo del proyecto

Recepción del Código Fuente (.glt):

- El usuario carga o escribe un archivo de código fuente con extensión .glt en la interfaz del proyecto.

Validación y envío del código:

- El código fuente capturado en la interfaz debe ser enviado al compilador construido mediante C# y ANTLR, queda a discreción del estudiante la forma de enviar dicha información.

Análisis del Código Fuente:

- El código fuente es procesado mediante el analizador generado con ANTLR.
- En caso de errores léxicos o sintácticos, estos son reportados y el proceso se detiene.

Análisis Semántico:

- Si el AST se genera correctamente, se realiza un análisis semántico para validar reglas como declaraciones, tipos y asignaciones. En caso de encontrarse errores semánticos, estos son reportados y la ejecución se detiene.

Recorrido del AST:

- El AST es recorrido para evaluar expresiones e instrucciones definidas en el lenguaje GoLight.

Generación de Resultados:

- Se generan los siguientes reportes:
    - **Reporte de Tabla de Símbolos:** Información sobre las variables, funciones y otros elementos declarados en el código fuente.
    - **Reporte de Errores:** Lista de errores encontrados durante las distintas fases del análisis.
- La consola mostrará el resultado de la compilación y permitirá descargar ese contenido en un archivo con extensión **“.s”**.

Preparación para la Ejecución

Se realiza la transformación del código en lenguaje ensamblador **ARM64** y se genera el código objeto. Luego, se lleva a cabo el proceso de linkeo para obtener un binario ejecutable.

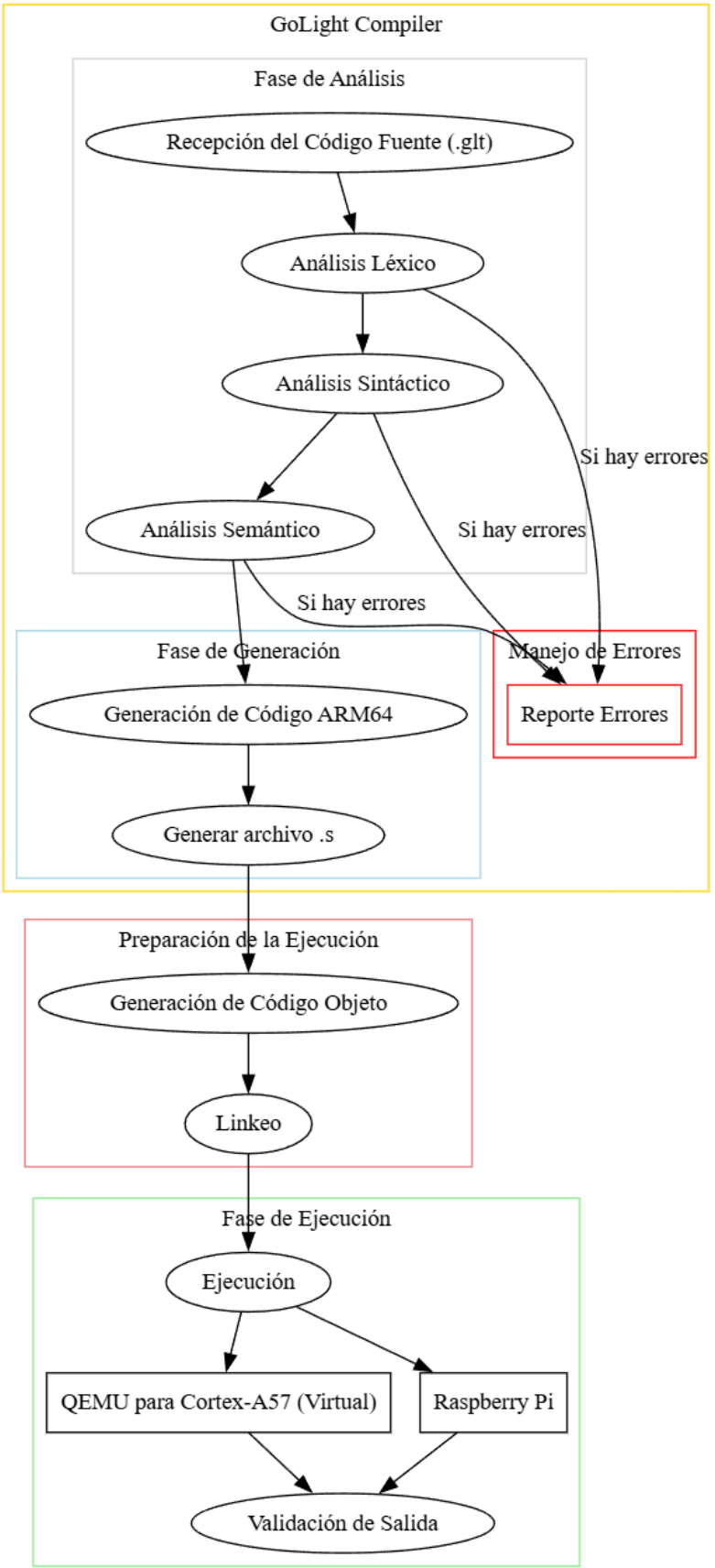
Ejecución y Validación

El código generado puede ejecutarse en:

- QEMU para Cortex-A57 (Virtual).
- Raspberry Pi.

Validación de la Salida del Código Equivalente en GoLang

Para garantizar la correcta traducción del código fuente en GoLight, se ejecuta el código en GoLang de alto nivel y se comparan los resultados con la ejecución en ARM64 (mediante QEMU o Raspberry Pi). Esto asegura la precisión y consistencia del compilador en la traducción del código.



Flujo de ejecución del proyecto

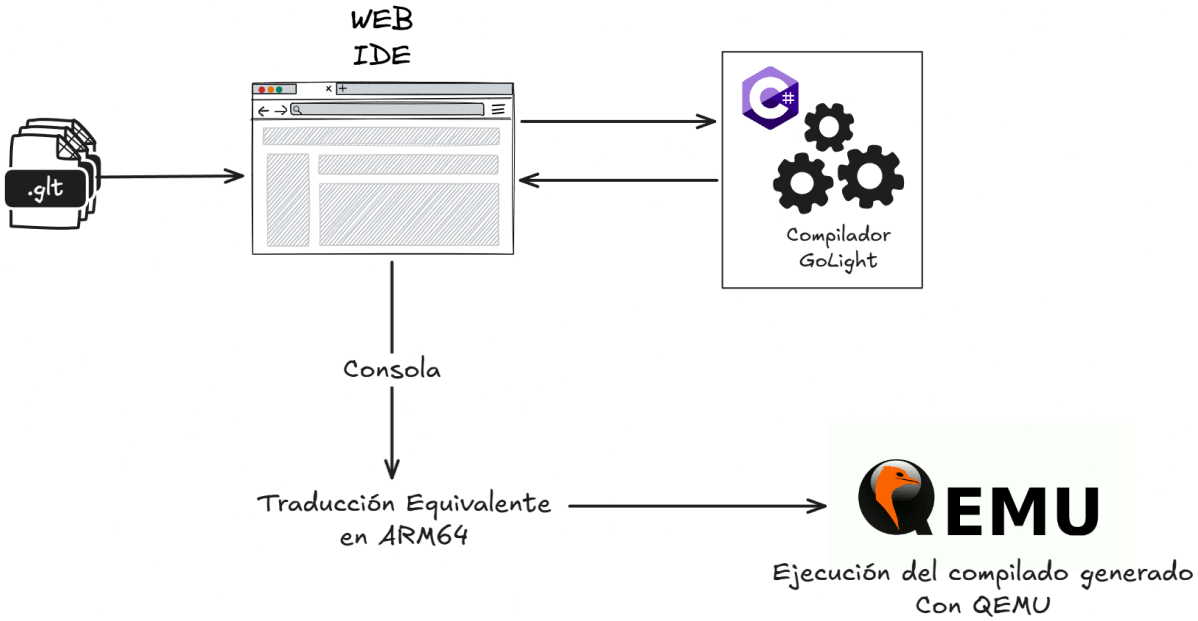
2.3. Tecnologías

Para el desarrollo del proyecto, es obligatorio el uso de las siguientes tecnologías:

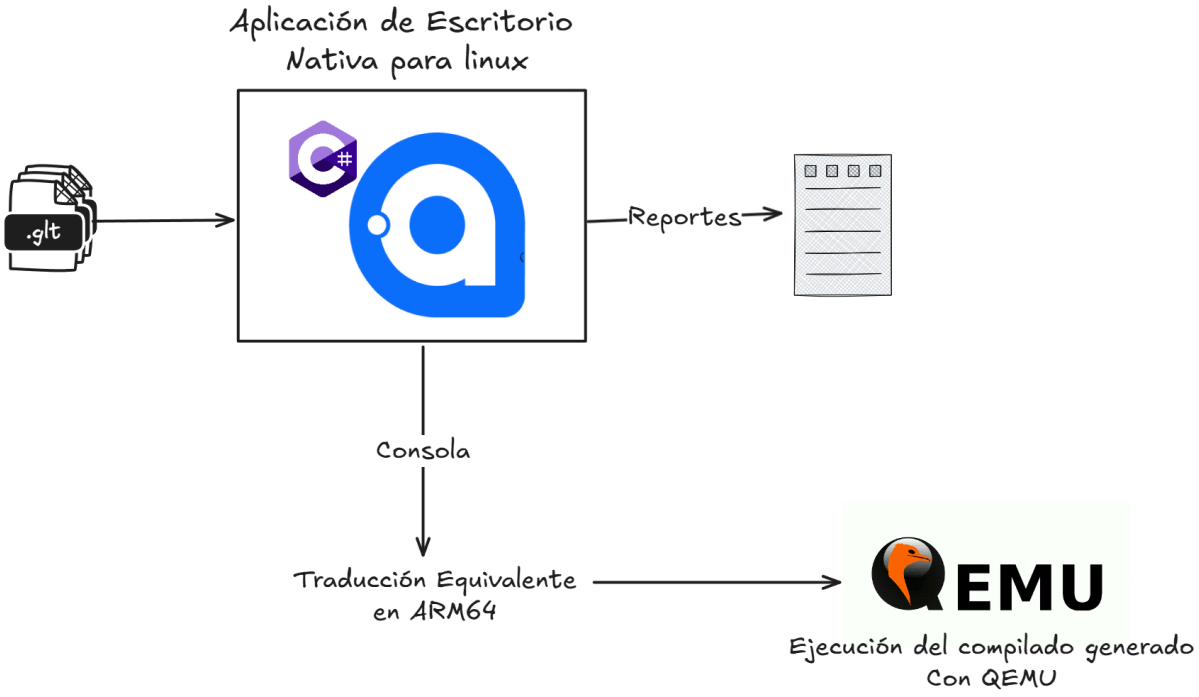
- **ANTLR:** Se usará para la generación del analizador léxico y sintáctico del lenguaje.
- **Linux:** La aplicación debe ejecutarse de forma nativa en Linux.
- **C#:** Toda la lógica del sistema, incluyendo el compilador y cualquier API, debe estar implementada en C#.
- **ARM64:** El código generado por el compilador debe traducirse a instrucciones compatibles con la arquitectura ARM64.
- **QEMU:** Se utilizará QEMU para emular un entorno ARM64 y ejecutar el código compilado, asegurando su correcto funcionamiento.

En cuanto al diseño del sistema, queda a discreción del estudiante cuál utilizar, sin embargo, se proponen las siguientes:

1. Implementado con ASP.NET, donde una aplicación web o de escritorio se comunique con una API en C# para procesar la lógica del lenguaje.



2. Desarrollada con Avalonia, permitiendo una aplicación de escritorio independiente que ejecute directamente el análisis y la compilación del código.



3. Generalidades del lenguaje GoLight

El lenguaje GoLight está inspirado en la sintaxis del lenguaje Go, por lo tanto se conforma por un subconjunto de instrucciones de este, con la diferencia de que GoLight tendrá una sintaxis más reducida pero sin perder las funcionalidades que caracterizan al lenguaje original.

3.1. Expresiones en el lenguaje GoLight

Cuando se haga referencia a una 'expresión' a lo largo de este enunciado, se hará referencia a cualquier sentencia u operación que devuelve un valor. Por ejemplo:

- Una operación aritmética, comparativa o lógica
- Acceso a un variable
- Acceso a un elemento de una estructura
- Una llamada a una función

3.2. Ejecución:

GoLight contará con una función **main**, la cual será el punto de entrada para la ejecución del programa. El compilador deberá localizar esta función y ejecutar las instrucciones definidas en su interior de manera estructurada y secuencial.

3.3. Identificadores

Un identificador será utilizado para dar un nombre a variables y métodos. Un identificador está compuesto básicamente por una combinación de letras, dígitos, o guión bajo.

Ejemplos de identificadores válidos:

```
IdValido
id_Valido
i1d_valido5
_value
```

Ejemplo de identificadores no válidos

```
&idNoValido
.5ID
true
Tot@l
1d
```

Consideraciones:

- El identificador puede iniciar con una letra o un guión bajo _
- No pueden comenzar con un número.
- Por simplicidad el identificador no puede contener carácteres especiales (.$,-, etc)
- Los identificadores no pueden coincidir con palabras reservadas del lenguaje.

3.4. Case Sensitive

El lenguaje GoLight es case sensitive, esto quiere decir que diferenciará entre mayúsculas con minúsculas, por ejemplo, el identificador variable hace referencia a una variable específica y el identificador Variable hace referencia a otra variable. Las palabras reservadas también son case sensitive por ejemplo la palabra **if** no será la misma que **IF.**

3.5. Comentarios

Un comentario es un componente léxico del lenguaje que no es tomado en cuenta para el análisis sintáctico de la entrada. Existirán dos tipos de comentarios:

- Los comentarios de una línea que serán delimitados al inicio con el símbolo de ***//*** y al final como un carácter de finalización de línea.
- Los comentarios con múltiples líneas que empezarán con los símbolos ***/**** y terminarán con los símbolos *****/***

```
// Esto es un comentario de una línea
/*
Esto es un comentario multilínea
*/
```

3.6. Tipos estáticos

El lenguaje GoLight no permitirá reasignar valores de diferentes tipos a una variable. Una vez que una variable haya sido declarada con un tipo específico, sólo será posible asignarle valores compatibles con ese tipo durante toda la ejecución del programa. Si se intenta asignar un valor de un tipo diferente al tipo declarado, el compilador deberá generar un mensaje detallado indicando el error y la incompatibilidad detectada.

3.7. Tipos de datos primitivos

Se utilizan los siguientes tipos de datos primitivos:

Tipo primitivo	Definición	Rango (teórico)	Valor por defecto
`int`	**Acepta valores números enteros**	**-2^{31} a $2^{31}-1$**	`0`
`float64`	**Número de punto flotante de 64 bits.**	**±1.7E ±308 (15-16 dígitos de precisión).**	`0.0`
`string`	**Acepta cadenas de caracteres**	**[0, 65535] caracteres (acotado por conveniencia)**	`“” (cadena vacía)`
bool	**Acepta valores lógicos de verdadero y falso**	**true false**	`false`
`rune`	**Representa un byte (alias de uint8).**	**0 a $2^{32}-1$.**	`0`

Consideraciones:

- Por conveniencia y facilidad de desarrollo, el tipo String será tomado como un tipo primitivo.
- Cuando se haga referencia a *tipos numéricos* se estarán considerando los tipos **`int`** y **`float64`**
- Cualquier otro tipo de dato que no sea primitivo tomará el valor por defecto de **nil** al no asignarle un valor en su declaración.
- Cuando se declare una **variable** y no se defina su valor, automáticamente tomará el valor por defecto del tipo correspondiente.
- El literal `0` se considera tanto de tipo **`int`** como **`float`**.
- Las literales de tipo **rune** deben de ser definidas con comilla simple (‘ ‘) mientras que las literales de **string** deben ser definidas com comilla doble (“ ”)

3.8. Tipos Compuestos

Cuando hablamos de tipos compuestos nos vamos a referir a ellos como no primitivos, en estos tipos vamos a encontrar las estructuras básicas del lenguaje GoLight.

- Slices

Estos tipos especiales se explicarán más adelante.

3.9. Valor nulo (`nil`)

En el lenguaje GoLight, la palabra reservada nil se utiliza para representar la ausencia de valor. Esto es aplicable únicamente a tipos que puedan contener referencias, como punteros, slices.

Cualquier operación sobre un valor nil será considerada un error semántico y deberá generar un mensaje detallado indicando la naturaleza del problema. Este comportamiento asegura que nil no pueda ser utilizado como un valor válido para operaciones lógicas o aritméticas.

3.10. 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	Definición
\"	**Comilla Doble**
****	**Barra invertida**
\n	**Salto de línea**
\r	**Retorno de carro**
\t	**Tabulación**

4. Sintaxis del lenguaje GoLight

A continuación, se define la sintaxis para las sentencias del lenguaje GoLight

4.1. Bloques de Sentencias

Un bloque de sentencias es un conjunto de instrucciones delimitado por llaves “{ }”. Cada bloque define un ámbito local que puede contener sus propias variables y sentencias. Las variables declaradas dentro de un bloque solo son accesibles dentro de ese bloque y en bloques anidados.

Ámbito global: Las variables declaradas fuera de cualquier bloque son accesibles desde todos los bloques.

Ámbito local: Las variables declaradas dentro de un bloque solo son accesibles dentro de ese bloque o en bloques anidados.

```
func main() {
    // Variable global
    i := 10c
    z := 0

    // Imprime 10
    fmt.Println("Valor de i en el ámbito global:", i)

    // Bloque independiente
    {
        // Variable local al bloque
        j := 20

        // Imprime 20
        fmt.Println("Valor de j en el bloque independiente:", j)
        // Imprime 10
        fmt.Println("Acceso a i desde el bloque independiente:", i)

        // Modifica i usando j
        i = i + j
        // Imprime 30
        fmt.Println("Nuevo valor de i después de modificarlo en el bloque:", i)

        // Variable con el mismo nombre que variable en entorno superior
        z := 40

        // Imprime 40
        fmt.Println("Valor de z en el bloque independiente:", z)
    }

    // Imprime 0
    fmt.Println("Valor de z fuera del bloque independiente:", z)

    // Imprime 30
    fmt.Println("Valor de i fuera del bloque:", i)

    // fmt.Println("Valor de j fuera del bloque:", j) // Error: j no es accesible aquí
}
```

Consideraciones:

- Bloques independientes: Los bloques de sentencias pueden declararse de forma independiente dentro de funciones, sin necesidad de estar asociados a una sentencia de control de flujo.
- Reglas de alcance: Las variables declaradas dentro de un bloque ocultan variables con el mismo nombre declaradas en ámbitos superiores.
- Finalización de sentencias: NO es obligatorio que todas las sentencias terminen con un punto y coma (;).
- Errores de acceso: Si se intenta acceder a una variable fuera de su ámbito, el compilador debe generar un error detallado indicando que la variable no está definida en ese contexto.

4.2. Signos de agrupación

Para dar orden y jerarquía a ciertas operaciones aritméticas se utilizarán los signos de agrupación. Para los signos de agrupación se utilizarán los paréntesis ( )

```
3 - (1 + 3) * 32 / 90 // 1.5
```

4.2.1. Variables

Una variable es un elemento de datos cuyo valor puede cambiar durante la ejecución de un programa, siempre y cuando mantenga su tipo de dato. Cada variable tiene un nombre único y un valor asociado. Los nombres de las variables no pueden coincidir con palabras reservadas ni con nombres de otras variables definidas en el mismo ámbito.

Para utilizar una variable, ésta debe ser declarada previamente. La declaración puede incluir o no un valor inicial, y el tipo de la variable puede ser definido explícitamente o inferido implícitamente del valor asignado.

Sintaxis:

```
// Declaración explícita con tipo y valor
var <identificador> <Tipo> = <Expresión>

// Declaración explícita con tipo y sin valor
var <identificador> <Tipo>

// Declaración implícita infiriendo el tipo
<identificador> := <Expresión>
```

Consideraciones:

- Inmutabilidad del tipo: Una variable puede cambiar su valor, pero su tipo no puede ser modificado una vez declarado.
- Se puede declarar una sola variable por sentencia.
- Si una variable ya existe, su valor puede ser actualizado, pero el nuevo valor debe ser del mismo tipo que el original.
- No puede ser una palabra reservada ni coincidir con el nombre de otra variable en el mismo ámbito.
- Si se intenta asignar un valor de un tipo diferente al declarado, el compilador generará un error.
- No se permite asignar valores de tipo diferente a la declaración inicial, excepto en la conversión implícita de int a float64.

```
// 5 es int, pero se convierte a float64 automáticamente.
var a float64 = 5
```

- Al usar var, no es obligatorio asignar un valor inicial, pero si no se asigna, la variable tomará el valor por defecto según su tipo

4.2.1.1. Asignación de variables

Consideraciones**:**

- Una vez declarada, la variable conserva su tipo y solo puede recibir valores compatibles con este. Esto aplica tanto a **tipos primitivos** (`int`, `float64`, `bool`, `string`, etc.) como a **slices** (`[]int`, `[]string`, etc.).
- Expresión válida: La parte derecha de la asignación debe evaluar a un tipo compatible con la variable de la izquierda.

Sintaxis:

```
// Asignación simple
<identificador> = <Expresión>
```

Ejemplos:

```
func main() {
    // Correcto, declaración sin valor inicial
    var valor int

    // Correcto, declaración de una variable tipo int con valor
    var valor1 int = 10

    // noInicializada tomará el valor por defecto de un int, que es 0
    var noinicializada int

    // Error: No se puede asignar un float64 a un int
    // var tipoincorrecto int = 10.01

    // Correcto, declaración de una variable tipo float64 con valor
    var valor2 float64 = 10.2

        // Correcto, las operaciones aritméticas de enteros se convierten a float64
implícitamente
    var valor2_1 float64 = 10 + 1

    // Correcto, declaración de una variable tipo string usando inferencia
    valor3 := "esto es una variable"

    // Correcto, declaración de una variable tipo rune
    var caracter rune = 'A'

    // Correcto, declaración de una variable tipo bool
    var valor4 bool = true

    // Error: No se puede asignar un valor bool a una variable de tipo string
    // var valor4 string = true

    // Error: No se puede redefinir una variable existente en el mismo ámbito
    // var valor3 int = 10

    // Correcto: Es posible declarar una variable con el mismo nombre en un nuevo bloque
    {
        valor3 := "redefiniendo variable con un tipo distinto"
        fmt.Println(valor3) // Imprime "redefiniendo variable con un tipo distinto"
```

```
    }

    // Error: .58 no es un nombre válido para una variable
    // var .58 int = 4

    // Error: "if" es una palabra reservada
    // var if string = "10"

    // Ejemplo de asignaciones

    // Correcto, se puede reasignar un nuevo valor del mismo tipo
    valor1 = 200

    // Correcto, se puede reasignar un nuevo valor del mismo tipo
    valor3 = "otra cadena"

    // Error: No se puede asignar un int a una variable de tipo bool
    // valor4 = 10

    // Correcto, asignación de un int a un float
    valor2 = 200

    // Error: No se puede asignar un string a una variable de tipo rune
    // caracter = "otra cadena"
}
```

4.3. Operadores Aritméticos

Los operadores aritméticos toman valores numéricos de expresiones y retornan un valor numérico único de un determinado tipo. Los operadores aritméticos estándar son adición o suma +, sustracción o resta -, multiplicación *, y división /, adicionalmente vamos a trabajar el módulo %.

4.3.1. Suma

La operación suma se produce mediante la suma de tipos numéricos o Strings concatenados, debido a que GoLight está pensado para ofrecer una mayor versatilidad ofrecerá conversión de tipos de forma implícita como especifica la siguiente tabla:

Operandos	Tipo resultante	Ejemplo
`int + int` `int + float64`	`int` `float64`	`1 + 1 = 2` `1 + 1.0 = 2.0`
`float64 + float64` `float64 + int`	`float64` `float64`	`1.0 + 13.0 = 14.0` `1.0 + 1 = 2.0`
`string + string`	`string`	`"ho" + "la" = "hola"`

Consideraciones:

- Cualquier otra combinación será inválida y se deberá reportar el error.

4.3.2. Resta

La resta se produce cuando existe una sustracción entre tipos numéricos, de igual manera que con otras operaciones habrá conversión de tipos implícita en algunos casos

Operandos	Tipo resultante	Ejemplo
`int - int` `int - float64`	`int` `float64`	`1 - 1 = 0` `1 - 1.0 = 0.0`
`float64 - float64` `float64 - int`	`float64` `float64`	`1.0 - 13.0 = -12.0` `1.0 - 1 = 0.0`

4.3.3. Multiplicación

La multiplicación se produce cuando existe un producto entre tipos numéricos, de igual manera que con otras operaciones habrá conversión de tipos implícita en algunos casos.

Operandos	Tipo resultante	Ejemplo
`int * int` `int * float64`	`int` `float64`	`1 * 10 = 10` `1 * 1.0 = 1.0`
`float64 * float64` `float64 * int`	`float64` `float64`	`1.0 * 13.0 = 13.0` `1.0 * 1 = 1.0`

Consideraciones:

- Cualquier otra combinación será inválida y se deberá reportar el error.

4.3.4. División

La división produce el cociente entre tipos numéricos, de igual manera que con otras operaciones habrá conversión de tipos implícita en algunos casos a su vez truncamiento cuando sea necesario.

Operandos	Tipo resultante	Ejemplo
`int / int` `int / float64`	`int` `float64`	`10 / 3 = 3` `1 / 3.0 = 0.3333`
`float64 / float64` `float64 / int`	`float64` `float64`	`13.0 / 13.0 = 1.0` `1.0 / 1 = 1.0`

Consideraciones:

- Cualquier otra combinación será inválida y se deberá reportar el error.
- Se debe verificar que **no haya división por 0**, de lo contrario se debe mostrar un mensaje de error.

4.3.5. Módulo

El módulo produce el residuo entre la división entre tipos numéricos de tipo **`int`**.

Operandos	Tipo resultante	Ejemplo
`int % int`	`int`	`10 % 3 = 1`

Consideraciones:

- Cualquier otra combinación será inválida y se deberá reportar el error.
- Se debe verificar que **no haya división por 0**, de lo contrario se debe mostrar una mensaje de error.

4.3.6. Operador de asignación

4.3.6.1. Suma

El operador += indica el incremento del valor de una **expresión** en una **variable** de tipo ya sea **`int`** o de tipo **`float64`** . El operador += será como una suma implícita de la forma: `variable = variable + expresión` Por lo tanto tendrá las validaciones y restricciones de una suma.

Ejemplos:

```
func main() {

```

```
    // Declaración e inicialización de variables
    var var1 int = 10
    var var2 float64 = 0.0

    // Correcto: Operación += con valores del mismo tipo
    var1 += 10 // var1 tendrá el valor de 20
    fmt.Println("var1:", var1)

    // Error: No se puede asignar un float64 a un int
    // var1 += 10.0

    // Correcto: Operación += con valores float64
    var2 += 10 // var2 tendrá el valor de 10.0
    fmt.Println("var2:", var2)

    // Correcto: Operación += con valores del mismo tipo (float64)
    var2 += 10.0 // var2 tendrá el valor de 20.0
    fmt.Println("var2:", var2)

    // Declaración de una cadena de texto
    str := "cad"

    // Correcto: Concatenación de strings con +=
    str += "cad" // str tendrá el valor de "cadcad"
    fmt.Println("str:", str)

    // Error: Operación inválida string + int
    // str += 10
}
```

4.3.6.2. Resta

El operador -= indica el decremento del valor de una **expresión** en una **variable** de tipo ya sea **`int`** o de tipo **`float64`** . El operador -= será como una resta implícita de la forma: `variable = variable - expresión` Por lo tanto tendrá las validaciones y restricciones de una resta.

Ejemplos:

```
func main() {
    // Declaración e inicialización de variables
```

```
    var var1 int = 10
    var var2 float64 = 0.0

    // Correcto: Operación -= con valores del mismo tipo
    var1 -= 10 // var1 tendrá el valor de 0
    fmt.Println("var1 después de -= 10:", var1)

    // Error: No se puede asignar un float64 a un int
    // var1 -= 10.0

    // Correcto: Operación -= con valores float64
    var2 -= 10 // var2 tendrá el valor de -10.0
    fmt.Println("var2 después de -= 10:", var2)

    // Correcto: Operación -= con valores del mismo tipo (float64)
    var2 -= 10.0 // var2 tendrá el valor de -20.0
    fmt.Println("var2 después de -= 10.0:", var2)
}
```

4.3.7. Negación unaria

El operador de negación unaria precede su operando y lo niega (*-1) esta negación se aplica a tipos numéricos

Operandos	Tipo resultante	Ejemplo
`-int`	`int`	`-(-(10)) = 10`
`-float64`	`float64`	`-(1.0) = -1.0`

4.4. Operaciones de comparación

Compara sus operandos y devuelve un valor lógico en función de si la comparación es verdadera (**`true`**) o falsa (**`false`**). Los operandos pueden ser numéricos, Strings o lógicos, *permitiendo únicamente la comparación de expresiones del mismo tipo*.

4.4.1. Igualdad y desigualdad

- **El operador de igualdad** (==) devuelve **`true`** si ambos operandos tienen el mismo valor, en caso contrario, devuelve **`false`**.

- **El operador no igual a** (!=) devuelve **`true`** si los operandos no tienen el mismo valor, de lo contrario, devuelve **`false`**.

Operandos	Tipo resultante	Ejemplo
`int [==,!=] int`	`bool`	`1 == 1 = true` `1 != 1 = false`
`float64 [==,!=] float64`	`bool`	`13.0 == 13.0 = true` `0.001 != 0.001 = false`
`int [==,!=] float` `float64 [==,!=] int`	`bool`	`35 == 35.0 = true` `98.0 = 98 = true`
`bool [==,!=] bool`	`bool`	`true == false = false` `false != true = true`
`string [==,!=] string`	`bool`	`"ho" == "Ha" = false` `"Ho" != "Ho" = false`
`rune [==,!=] rune`	`bool`	`'h' == 'a' = false` `'H' != 'H' = false`

Consideraciones

- Cualquier otra combinación será inválida y se deberá reportar el error.
- Las comparaciones entre cadenas se hacen lexicográficamente (carácter por carácter).

4.4.2. Relacionales

Las operaciones relacionales que soporta el lenguaje GoLight son las siguientes:

- **Mayor que:** (>) Devuelve **`true`** si el operando de la izquierda es mayor que el operando de la derecha.
- **Mayor o igual que:** (>=)Devuelve true si el operando de la izquierda es mayor o igual que el operando de la derecha.
- **Menor que:** (<) Devuelve true si el operando de la izquierda es menor que el operando de la derecha.
- **Menor o igual que:** (<=) Devuelve true si el operando de la izquierda es menor o igual que el operando de la derecha.

Operandos	Tipo resultante	Ejemplo
`int[>,<,>=,<=] int`	`bool`	`1 < 1 = false`

Operandos	Tipo resultante	Ejemplo
`float64 [>,<,>=,<=] float64`	`bool`	`13.0 >= 13.0 = true`
`int [>,<,>=,<=] float64`	`bool`	`65 >= 70.7 = false`
`float64 [>,<,>=,<=] int`	`bool`	`40.6 >= 30 = true`
`rune [>,<,>=,<=] rune`	`bool`	`'a' <= 'b' = true`

Consideraciones

- Cualquier otra combinación será inválida y se deberá reportar el error.
- La comparación de valores tipos rune se realiza comparando su valor ASCII.
- La limitación de las operaciones también se aplica a comparación de literales.

4.5. Operadores Lógicos

Los operadores lógicos comprueban la veracidad de alguna condición. Al igual que los operadores de comparación, devuelven el tipo de dato **`bool`** con el valor **`true`** ó **`false`**.

- **Operador and** (&&) devuelve **`true`** si ambas expresiones de tipo **`bool`** son **`true`**, en caso contrario devuelve **`false`**.
- **Operador or** (||) devuelve **`true`** si alguna de las expresiones de tipo **`bool`** es **`true`**, en caso contrario devuelve **`false`**.
- **Operador not** (`!`) Invierte el valor de cualquier expresión booleana.

A	B	A && A	A || B	! A
`true`	`true`	`true`	`true`	`false`
`true`	`false`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`true`
`false`	`false`	`false`	`false`	`true`

Consideraciones:

- Ambos operadores deben ser booleanos, si no se debe reportar el error.

4.6. Precedencia y asociatividad de operadores

La precedencia de los operadores indica el orden en que se realizan las distintas operaciones del lenguaje. Cuando dos operadores tengan la misma precedencia, se utilizará la asociatividad para decidir qué operación realizar primero.

A continuación, se presenta la precedencia en orden de mayor a menor de operadores lógicos, aritméticos y de comparación .

Operador	Asociatividad
( ) [ ]	izquierda a derecha
! -	derecha a izquierda
/ % *	izquierda a derecha
+ -	izquierda a derecha
< <= >= >	izquierda a derecha
== !=	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha

4.7. Sentencias de control de flujo

Las estructuras de control permiten regular el flujo de la ejecución del programa. Este flujo de ejecución se puede controlar mediante sentencias condicionales que realicen ramificaciones e iteraciones. Se debe considerar que estas sentencias se encontrarán únicamente dentro funciones.

4.7.1. Sentencia If Else

La sentencia if-else permite ejecutar bloques de código dependiendo del resultado de una condición. Si la condición evaluada resulta verdadera, se ejecuta el bloque de código asociado al if. Si es falsa, se puede evaluar un bloque else if o ejecutar un bloque else en su defecto.

Ejemplo:

```
var condicion = true
if condicion {
    // Bloque de sentencias para el if
} else if condicion {
    // Bloque de sentencias para el else if
} else {
```

```
    // Bloque de sentencias para el else
}
```

Consideraciones:

- Puede venir cualquier cantidad de if de forma anidada
- La expresión debe devolver un valor tipo **`bool`** en caso contrario debe tomarse como error y reportarlo.
- La condición puede o no ir entre paréntesis.

4.7.2. Sentencia Switch - Case

La sentencia switch evalúa una expresión y ejecuta el bloque de declaraciones correspondiente al primer case que coincida. Si no hay coincidencia, se ejecutará la cláusula default, si está presente. Por convención, el bloque default se coloca al final del switch.

Sintaxis:

```
switch <expresión> {
    case valor1:
        // Declaraciones ejecutadas si <expresión> == valor1
    case valor2:
        // Declaraciones ejecutadas si <expresión> == valor2
    // ...
    default:
        // Declaraciones ejecutadas si ningún caso coincide
}
```

Ejemplo:

```
switch numero {
    case 1:
        fmt.Println("Uno") // Se ejecuta si numero == 1
    case 2:
        fmt.Println("Dos") // Se ejecuta si numero == 2
    case 3:
        fmt.Println("Tres") // Se ejecuta si numero == 3
    default:
        fmt.Println("Número inválido") // Se ejecuta si ninguno de los casos coincide
}
```

Consideraciones:

- En GoLight, el break implícito está incluido al final de cada case
- Si no se incluye un bloque default y no hay coincidencias, simplemente no se ejecuta nada dentro del switch.
- No es necesario utilizar paréntesis alrededor de la expresión en un switch.

4.7.3. Sentencia For

En GoLight, el bucle for es la única estructura de iteración disponible. Es flexible y puede usarse para replicar el comportamiento de bucles como while o do-while

Sintaxis

```
for <condición> {
    // Bloque de sentencias
}


for inicialización; condición; incremento {
    // Bloque de sentencias
}

for índice, valor := range slice {
    //...
}
```

Ejemplos

```
i := 1
for i <= 5 {
   fmt.Println(i)
   i++
}

for i := 1; i <= 5; i++ {
    fmt.Println(i)
}
```

```go
numeros := []int{10, 20, 30, 40, 50}
for índice, valor := range numeros {
    fmt.Println("índice:", índice, "valor:", valor)
}
```

4.8. Sentencias de transferencia

Estas sentencias transferirán el control a otras partes del programa y se podrán utilizar en entornos especializados.

4.8.1. Break

La sentencia break finaliza inmediatamente el bucle actual o una sentencia switch y transfiere el control al siguiente bloque de código después de ese elemento.

Ejemplo:

```go
for i := 0; i < 10; i++ {
    if i == 5 {
        fmt.Println("Se encontró un break en i =", i)
        break // Finaliza el bucle cuando i es igual a 5
    }
    fmt.Println(i)
}
```

Consideraciones:

- La sentencia break solo se puede usar dentro de un bucle (for) o un switch.
- Si se encuentra un **break** fuera de un ciclo y/o sentencia switch se considerará como un error.

4.8.2. Continue

La sentencia continue detiene la ejecución de las sentencias restantes en la iteración actual de un bucle y pasa directamente a la siguiente iteración.

Ejemplo:

```go
for i := 1; i <= 5; i++ {
    if i % 2 == 0 {
```

```
        continue // Salta a la siguiente iteración si i es par
    }
    fmt.Println(i) // Solo imprime números impares
}
```

Consideraciones:

- La sentencia continue solo se puede usar dentro de un bucle (for).
- Si se encuentra un **`continue`** fuera de un ciclo se considerará como un error.

4.8.3. Return

Sentencia que finaliza la ejecución de la función actual, puede o no especificar un valor para ser devuelto a quien llama a la función.

```
func suma(a int, b int) int {
    return a + b // Retorna la suma de a y b
}

func main() {
    resultado := suma(3, 7)
    fmt.Println("Resultado:", resultado)
}
```

5. Estructuras de datos

Las estructuras de datos en el lenguaje GoLight son los componentes que nos permiten almacenar un conjunto de valores agrupados de forma ordenada, las estructuras básicas que incluye el lenguaje son los **Slice.**

5.1. Slice

Los slice son la estructura compuesta más básica del lenguaje GoLight, los tipos de slice que existen son con base a los tipos **primitivos** del lenguaje. Su notación de posiciones por convención comienza con 0.

5.1.1. Creación de slice

Para crear vectores se utiliza la siguiente sIntaxis.
Sintaxis:

```
// Declaración con inicialización de valores
numbers = []int {1, 2, 3, 4, 5};

// Declaración de slice vacío
var slice []int

slice = numbers
```

Consideraciones:

- La lista de expresiones debe ser del mismo tipo que el tipo del slice.
- El tamaño de un arreglo no puede ser negativo
- El tamaño del slice puede aumentar o disminuir a lo largo de la ejecución.

5.1.2. Función `slices.Index`

Retorna el índice de la primer coincidencia que encuentre, de lo contrario retornará -1

```
numeros := []int{10, 20, 30, 40, 50}

// Usar slices.Index para buscar valores
fmt.Println(slices.Index(numeros, 30))  // Salida: 2
fmt.Println(slices.Index(numeros, 100)) // Salida: -1
```

5.1.3. Funcion `strings.Join`

Permite unir todos los elementos de un slice de cadenas (**`[]string`**) en una sola cadena de texto. Los elementos se concatenan utilizando un separador especificado, que puede ser cualquier string.

```
palabras := []string{"hola", "mundo", "go"}
fmt.Println(strings.Join(palabras, " ")) // Salida: "hola mundo go"
```

Consideraciones:

- Esta función sólo será válida para los slices del tipo `[]string`

5.1.4. Función `len`

Devuelve la cantidad de elementos presentes en un slice. El valor retornado es de tipo `int`.

```
numeros := []int{1, 2, 3, 4, 5}
fmt.Println(len(numeros)) // Salida: 5
```

5.1.5. Función append

Agrega elementos a un slice, retornando un nuevo slice con los elementos añadidos.

```
numeros := []int{1, 2, 3}

numeros = append(numeros, 4)
fmt.Println(numeros)
```

5.1.6. Acceso de elemento:

Los arreglos soportan la notación para la asignación, modificación y acceso de valores, únicamente con los valores existentes en la posición dada, en caso que la posición no exista deberá mostrar un mensaje de error.

Ejemplo:

```
// Definición de un slice
numeros := []int{10, 20, 30, 40, 50}

// Acceso a un elemento existente
fmt.Println("Elemento en índice 2:", numeros[2]) // Salida: 30

// Modificación de un elemento existente
numeros[2] = 100

// Salida: [10, 20, 100, 40, 50]
fmt.Println("Slice después de la modificación:", numeros)

// Intentar acceder a un índice fuera de rango
// Esto genera un error
// fmt.Println(numeros[10])
```

6. Funciones

En GoLight, las funciones son bloques de código reutilizables que realizan tareas específicas. Una función puede aceptar parámetros y devolver un valor, o simplemente ejecutar instrucciones sin retornar un resultado.

6.1. Declaración de funciones

Consideraciones:

- Las funciones pueden declararse solamente en el ámbito global.
- Si una función no devuelve un valor, no se especifica ningún tipo de retorno.
- Si devuelve un valor, el tipo de retorno debe ser explícitamente declarado.
- El valor de retorno debe de ser del **mismo** tipo del tipo de retorno de la función.
- Las funciones, variables no pueden tener el mismo nombre.
- Por simplicidad, las funciones solo pueden retornar un valor a la vez.
- No pueden existir funciones con el mismo nombre aunque tengan diferentes parámetros o diferente tipo de retorno.
- El nombre de la función no puede ser una palabra reservada.
- Los parámetros deben declararse explícitamente según su tipo.
- Los slices se pasan por referencia; los demás tipos (int, float64, string, bool, etc.) se pasan por valor.

6.1.1. Parámetros de funciones

Los parámetros en las funciones son variables que podemos utilizar mientras nos encontremos en el ámbito de la función.

Consideraciones:

- Los parámetros de tipo struct y slice son pasados por referencia, mientras que el resto de tipos primitivos son pasados por valor.
- No pueden existir parámetros con el mismo nombre.
- Pueden existir funciones sin parámetros.
- Los parámetros deben tener indicado el tipo que poseen, en caso contrario será considerado un error.

Sintaxis:

```
// Función sin parámetros y sin retorno
func <nombreFuncion>() {
    // <cuerpo de la función>
}
```

```
// Función con parámetros y sin retorno
func <nombreFuncion>(<param1> <tipo1>, <param2> <tipo2>) {
    // <cuerpo de la función>
}

// Función con parámetros y con retorno
func <nombreFuncion>(<param1> <tipo1>, <param2> <tipo2>) <tipoRetorno> {
    // <cuerpo de la función>
    return <valorDeRetorno>
}
```

Ejemplo:

```
struct Producto {
    int id;
    string nombre;
}

// Función que devuelve un valor entero
func obtenerNumero() int {
    return 42;
}

// Función que no devuelve nada
func imprimirMensaje() {
    fmt.Println("Hola, GoLight!");
}

// Función que suma dos números
func sumar(a int, b int) int {
    return a + b;
}

// Función que modifica un struct por referencia
func actualizarProducto(p Producto, nuevoNombre string) {
    p.nombre = nuevoNombre;
}

// Función que trabaja con slices
```

```
func agregarElemento(slice []int, valor int) []int {
    return append(slice, valor);
}

// Programa principal
func main() {
    // Llamadas a funciones
    fmt.Println(obtenerNumero());          // Salida: 42
    imprimirMensaje();                     // Salida: "Hola, GoLight!"
    fmt.Println(sumar(5, 10));             // Salida: 15

    // Trabajando con slices
    numeros := []int{1, 2, 3};
    numeros = agregarElemento(numeros, 4);
    fmt.Println(numeros); // Salida: [1, 2, 3, 4]
}
```

6.2. Funciones Embebidas

El lenguaje GoLight está basado en Typescript y este a su vez es un superset de sentencias de Goscript, por lo que en GoLight contamos con algunas de las funciones embebidas más utilizadas de este lenguaje.

6.2.1. Función *`fmt.Println`*

Permite imprimir una o más expresiones en una línea, separándolas automáticamente con un espacio y finalizando con un salto de línea.
Consideraciones

- Puede venir cualquier cantidad de expresiones separadas por coma.
- Se debe de imprimir un salto de línea al final de toda la salida.
- Los elementos se imprimen separados por un espacio.
- Si no se proporcionan argumentos, simplemente imprime un salto de línea.

6.2.1.1. Tipos permitidos

- **Primitivos:** `int`, `float64`, `bool`, `char`, `string`.
- **Slices:** Se imprimen en formato de lista `[valores]`.

Ejemplo:

```
fmt.Println("cadena1", "cadena2")    // Salida: cadena1 cadena2
fmt.Println("cadena1")               // Salida: cadena1
fmt.Println(10, true, 'A')           // Salida: 10 true A
fmt.Println(1.00001)                 // Salida: 1.00001

numeros := []int{1, 2, 3}
fmt.Println("Slice:", numeros)       // Salida: Slice: [1 2 3]
```

6.2.2. Función `strconv.Atoi`

Convierte una cadena de texto que representa un número entero en un valor de tipo `int`. Si la cadena no puede convertirse debe notificar un error.
Consideraciones:

- **No redondea valores decimales**. Si se intenta convertir un número en formato decimal, como `"123.45"`, generará un error.

Ejemplo:

```
numero := strconv.Atoi("123")
fmt.Println("Número:", numero) // Salida: Número: 123
```

6.2.3. Función `strconv.ParseFloat`

Permite convertir una cadena de texto que representa un número decimal o entero en un valor de tipo `float64`. Si la cadena no puede convertirse, se genera un error.

Consideraciones:

- La cadena debe representar un número decimal o entero válido.
- Los valores enteros se convierten automáticamente en flotantes sin errores.

Ejemplo:

```
numero := strconv.ParseFloat("123.45")
fmt.Println("Número:", numero) // Salida: Número: 123.45
```

6.2.4. Función reflect.TypeOf().string

Devuelve el tipo de un valor en tiempo de ejecución como un objeto de tipo `reflect.Type`. Este método se puede usar para determinar el tipo de datos asociado con variables, incluyendo tipos primitivos como `int`, `string`, `float`, `bool`, y tipos compuestos como lo son slices.

```
// Tipo int
numero := 42
tipoNumero := reflect.TypeOf(numero)
fmt.Println("Tipo de numero:", tipoNumero) // Salida: int

// Tipo float
decimal := 3.1416
tipoDecimal := reflect.TypeOf(decimal)
fmt.Println("Tipo de decimal:", tipoDecimal) // Salida: float64

// Tipo struct
p := Persona{Nombre: "Alice", Edad: 25}
tipoStruct := reflect.TypeOf(p)
fmt.Println("Tipo de p:", tipoStruct) // Salida: Persona

// Tipo slice
slice := []int{1, 2, 3}
tipoSlice := reflect.TypeOf(slice)
fmt.Println("Tipo de slice:", tipoSlice) // Salida: []int
```

7. Generación de Código Assembler

La generación de código assembler para la arquitectura ARM64 implica la traducción de las estructuras del lenguaje GoLight instrucciones específicas del conjunto de instrucciones ARM64. Durante este proceso, se mapean las construcciones del lenguaje, como declaraciones de variables, expresiones aritméticas, control de flujo y llamadas a funciones, a secuencias de instrucciones ARM64 que implementan la funcionalidad deseada.

Para la generación de código assembler en la arquitectura ARM64, se optará por utilizar el conjunto de instrucciones AArch64, que representa la implementación de 64 bits de la arquitectura ARM. Este conjunto de instrucciones proporciona operaciones fundamentales de aritmética, lógica y transferencia de datos, esenciales para la ejecución de programas en GoLight.

Para verificar la salida del compilador, se utilizará **QEMU**, un emulador y virtualizador de código abierto que permite ejecutar código en la arquitectura ARM64 emulando un procesador **Cortex-A57**. Esto permitirá probar el código sin necesidad de hardware físico. En este caso, QEMU proporcionará un entorno adecuado para ejecutar el código ensamblador generado y evaluar su comportamiento.

Alternativamente, el código ensamblador generado podrá ejecutarse directamente en hardware compatible, como una **Raspberry Pi** con un procesador ARM64. **Ambas**

opciones, ya sea virtualizando con QEMU o ejecutando el código directamente sobre un dispositivo físico compatible como lo es la Raspberry Pi, son válidas para verificar la traducción del código generado.

Por lo tanto, el compilador debe generar un archivo con extensión **".s"** que contenga el código ensamblador producido. Este archivo se utilizará para validar el código tanto en el simulador QEMU o en una Raspberry Pi, permitiendo ejecutar el código en un entorno simulado o en hardware real para verificar su correcto funcionamiento.

- Para poder utilizar el emulador de Qemu, tomar como referencia la siguiente documentación: Assembler Installation, Overview
- Para acceder a la documentación oficial de ARM64 utilizar el siguiente enlace: AArch64/ARM64 Assembly Tutorial

7.1 Comentarios

Un comentario es un componente léxico del lenguaje que no es tomado en cuenta para el análisis de la entrada. Se utilizará el formato del mismo ARM64 para comentarios:

- Los comentarios de una línea serán delimitados al inicio con los símbolos "#" y al final con un carácter de finalización de línea.
- Los comentarios con múltiples líneas empezarán con los símbolos "/*" *y terminarán con los símbolos "*/".
-

```
# Este es un comentario en ARM64
# La siguiente línea carga el valor 10 en el registro x10
mov x10, #10
/* Esto es un comentario multilínea */
```

En este ejemplo, las líneas que comienzan con # son comentarios y no tienen ningún impacto en el código ensamblador. Los comentarios son útiles para explicar el código y hacer anotaciones.

7.2 Registros

En ARM64, los tipos de datos se manejan principalmente a nivel de registros y direcciones de memoria, sin tipos de datos explícitos como en lenguajes de más alto nivel. Los registros almacenan diferentes tipos de datos dependiendo de su propósito.

7.2.1 Registros de propósito general:

Estos registros pueden usarse para almacenar enteros, direcciones de memoria o cualquier otro tipo de dato representable en el tamaño del registro.

```
# Cargar un entero en el registro x10
mov x10, #42

# Cargar una dirección de memoria en el registro x11
adr x11, mi_etiqueta
```

7.2.2 Registros de coma flotante:

Estos registros se utilizan para operaciones de punto flotante, soportando números en formatos de precisión simple o doble, de acuerdo con el estándar IEEE-754.

```
# Cargar un número de punto flotante de precisión simple
# en el registro s10
ldr s10, [mi_etiqueta]

# Cargar un número de punto flotante de doble precisión
# en el registro d11
ldr d11, [mi_etiqueta_doble]
```

7.2.3 Memoria:

La memoria en ARM64 se maneja como un conjunto de direcciones que apuntan a bytes. No existe un tipo de dato explícito para la memoria, pero se puede almacenar y cargar datos en direcciones específicas de memoria.

```
# Almacenar un valor en la memoria
str x10, [x11]   # Almacena el valor de x10 en la dirección apuntada por x11

# Cargar un valor de la memoria en un registro
ldr x12, [x11]   # Carga el valor de la dirección apuntada por x11 en x12
```

7.2.4 Etiquetas

En ARM64, las etiquetas también se utilizan para indicar ubicaciones específicas dentro del código y permitir saltos (saltos condicionales o incondicionales) durante la ejecución. Las

etiquetas se definen de manera similar a como se hace en otros lenguajes ensambladores, y se utilizan en combinación con instrucciones de salto.

Las etiquetas se definen en el código como identificadores únicos seguidos de dos puntos (:). Estas se utilizan para marcar ubicaciones en el código y se utilizan junto con instrucciones de salto como b (salto incondicional) o bne (salto condicional) para mover el flujo de ejecución.

Las etiquetas no requieren un formato específico, pero deben ser identificadores válidos. En ARM64, estas etiquetas se usan para referirse a ubicaciones en el código y permitir saltos a esas ubicaciones.

```
L1:
    # Código para inicializar un bucle
    mov x10, #0    # Inicializa el contador en 0
    mov x11, #10   # Inicializa el valor de salida del bucle

L3:
    # Código del bucle
    add x10, x10, #1   # Incrementa el contador en 1

    # Comprobación de la condición de salida del bucle
    cmp x10, x11        # Compara x10 con x11
    bne L3               # Salta al inicio del bucle si x10 no es igual a
    x11

L100:
    # Código después del bucle
    mov x12, #0    # Código posterior al bucle
```

Este ejemplo muestra cómo utilizar etiquetas para controlar el flujo de ejecución dentro de un bucle en ARM64.

7.2.5 Saltos

En ARM64, las instrucciones de salto condicional e incondicional funcionan de manera similar a otros ensambladores, permitiendo cambiar el flujo de ejecución del programa.

Instrucciones de salto condicional:
Las instrucciones de salto condicional permiten realizar saltos basados en el estado de los registros, como resultados de comparaciones previas. ARM64 tiene una serie de instrucciones para realizar saltos condicionales dependiendo de los indicadores de estado (como Zero, Negative, Carry, y Overflow) almacenados en el registro NZCV.

```
# Saltar a la etiqueta L1 si x2 es igual a x3
cmp x2, x3      # Compara x2 con x3
b.eq L1         # Salta a L1 si x2 es igual a x3

# Saltar a la etiqueta L2 si x4 no es igual a x5
cmp x4, x5      # Compara x4 con x5
b.ne L2         # Salta a L2 si x4 no es igual a x5

# Saltar a la etiqueta L3 si x6 es menor que x7
cmp x6, x7      # Compara x6 con x7
b.lt L3         # Salta a L3 si x6 es menor que x7
```

Instrucciones de salto incondicional

Estas instrucciones siempre realizan el salto, sin ninguna condición asociada.

```
# Saltar incondicionalmente a la etiqueta L1
b L1            # Salta a L1

# Saltar incondicionalmente a la etiqueta L2 y guardar la dirección de
 retorno
bl L2           # Salta a L2 y guarda la dirección de retorno en el registro
 x30 (Link Register)

# Saltar a la dirección almacenada en el registro x9
br x9           # Salta a la dirección especificada en el registro x9
```

7.2.6 Operadores

En ARM64, las operaciones aritméticas básicas se realizan mediante instrucciones específicas, similares a las de otros lenguajes ensambladores. A continuación se detallan

cómo realizar las operaciones básicas de suma, resta, multiplicación, división, y operaciones de incremento y decremento en ARM64.

Suma:

```
add x3, x1, x2   # x3 = x1 + x2
```

Resta:

```
sub x3, x1, x2   # x3 = x1 - x2
```

Multiplicación:

```
mul x3, x1, x2   # x3 = x1 * x2
```

División:

ARM64 tiene instrucciones para división de enteros: sdiv (división con signo) y udiv (división sin signo)

```
sdiv x3, x1, x2  # x3 = x1 / x2 (división con signo)
```

Incremento y decremento:

```
subi x3, x3, 1   # Decrementa x3 en 1
addi x3, x3, 1   # Incrementa x3 en 1
```

7.2.7 Ejemplo de Entrada y Salida

Entrada:

```
func main() {
    a := 0
    b := 7
    var result int

    if a > b {
        result = a
    } else {
        result = b
    }

    fmt.Println(result)
```

```
}
```

Salida:

```
    .text
    .global _start

_start:
    mov x0, #0        // a = 0
    mov x1, #7        // b = 7
    mov x2, #0        // result = 0

    cmp x0, x1        // Compara a (x0) con b (x1)
    ble else_label    // Si a <= b, salta a "else_label"

    mov x2, x0        // Si a > b, result = a (x0)
    b end_label       // Salta al final

else_label:
    mov x2, x1        // Si a <= b, result = b (x1)

end_label:
    // En este punto, x2 contiene el valor de result (ya sea a o b)
    // Imprimir el valor de result (x2) en la salida estándar

    mov x0, #1        // Número de la salida estándar (stdout)
    mov x1, x2        // El valor a imprimir
    mov x8, #64       // Número de la llamada al sistema para write
    svc #0            // Llamada al sistema

    mov x8, #93       // Número de la llamada al sistema para salir
    svc #0            // Llamada al sistema
```

8. Reportes Generales

Como se indicaba al inicio, el lenguaje GoLight genera una serie de reportes sobre el proceso de análisis de los archivos de entrada. Los reportes son los siguientes:

8.1. Reporte de errores

El compilador deberá ser capaz de detectar todos los errores que se encuentren durante el proceso de compilación. Todos los errores se deberán de recolectar y se mostrará un reporte de errores en el que, como mínimo, debe mostrarse el tipo de error, su ubicación y una breve descripción de por qué se produjo.

No.	Descripción	Línea	Columna	Tipo
1	El struct “Persona” no fue definido.	5	1	semántico
2	No se puede dividir entre cero.	19	6	semántico
3	El símbolo “¬” no es aceptado en el lenguaje.	55	2	léxico

8.2. Reporte de tabla de símbolos

Este reporte mostrará la tabla de símbolos después de la ejecución del archivo. Se deberán de mostrar todas las variables, funciones y procedimientos que fueron declarados, así como su tipo y toda la información que el estudiante considere necesaria para demostrar que el compilador ejecutó correctamente el código de entrada.

ID	Tipo símbolo	Tipo dato	Ámbito	Línea	Columna
x	Variable	int	Global	2	5
Ackerman	Función	float64	Global	5	1
vector1	Variable	Slice	Ackerman	10	5

9. Entregables

El estudiante deberá entregar únicamente el link de un repositorio en GitHub, el cual será privado y contendrá todos los archivos necesarios para la ejecución de la aplicación, así

como el código fuente, la gramática y la documentación. El estudiante es responsable de verificar el contenido de los entregables, los cuales son:

9.1. Código fuente de la aplicación
9.2. Gramáticas utilizadas
9.3. Manual Técnico
9.4. Manual de Usuario

El nombre del repositorio debe seguir el siguiente formato: **OLC2_Proyecto2_#Carnet**. Ejemplo: OLC2_Proyecto2_202110568

Se deben conceder los permisos necesarios a **TODOS** los auxiliares para acceder a dicho repositorio. Los usuarios son los siguientes:

- damianpeaf
- Henry2311
- rposadas15

10. Restricciones

10.1. Para el analizador léxico y sintáctico se debe implementar una gramática con la herramienta **ANTLR**.
10.2. La ejecución del proyecto se debe de hacer en un sistema operativo **linux**.
10.3. **Las copias de proyectos tendrán de manera automática una nota de 0 puntos y los involucrados serán reportados a la Escuela de Ciencias y Sistemas.**
10.4. El desarrollo y la entrega del proyecto son de manera **individual**.
10.5. Las instrucciones del código fuente deben ser traducidas a su equivalente en **ARM64**. La ejecución y el resultado del código ensamblador serán validados mediante su ejecución en un entorno **QEMU** para un procesador **Cortex-A57** o en una **Raspberry Pi**.

11. Consideraciones

11.1. Durante la calificación se realizarán preguntas sobre el código para verificar la autoría de este, de no responder correctamente la mayoría de las preguntas ***los tutores harán una verificación exhaustiva en busca de copias*.**
11.2. **El uso de inteligencia artificial para la generación de código será considerado como plagio**. En consecuencia, será reportado a la escuela de sistemas.
11.3. Se necesita que el estudiante al momento de la calificación tenga el entorno de desarrollo y las herramientas necesarias para realizar pequeñas modificaciones en el código para verificar la autoría de este, en caso que el estudiante no pueda realizar dichas modificaciones en un tiempo prudencial,

el estudiante tendrá 0 en la sección ponderada a dicha sección y ***los tutores harán una verificación exhaustiva en busca de copias*****.**

11.4. Se tendrá un máximo de 45 minutos por estudiante para calificar el proyecto. La calificación será de manera presencial y se solicitará la firma del estudiante.

11.5. 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.

11.6. Si una función del programa ya ha sido calificada, esta no puede ser penalizada si en otra sección la función falla o es errónea.

11.7. Los archivos de entrada permitidos en la calificación son únicamente los archivos preparados por los tutores.

11.8. Los archivos de entrada podrán ser modificados solamente antes de iniciar la calificación eliminando funcionalidades que el estudiante indique que no desarrolló.

11.9. La sintaxis descrita en este documento son con fines descriptivos, el estudiante es libre de diseñar la gramática que crea apropiada para el reconocimiento del lenguaje GoLight.

12. Entrega del proyecto

12.1. La entrega se realizará de manera virtual, se habilitará un apartado en la plataforma de UEDI para que el estudiante realice su entrega.

12.2. No se recibirán proyectos fuera de la fecha y hora estipulada.

12.3. La entrega de cada uno de los proyectos es **individual.**

12.4. Fecha límite de entrega del proyecto: **2 de mayo de 2025 a media noche.**