

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



USAC
TRICENTENARIA
Universidad de San Carlos de Guatemala

GoLight

1. Competencias

1.1. Competencia General

1.2. Competencias Específicas

2. Descripción

2.1. Componentes de la Interfaz

2.1.1. Editor

2.1.2. Funcionalidades

2.1.3. Herramientas

2.1.4. Reportes

2.1.5. Consola

2.2. Flujo del proyecto

2.3. Tecnologías

3. Generalidades del lenguaje GoLight

3.1. Expresiones en el lenguaje GoLight

3.2. Ejecución:

3.3. Identificadores

3.4. Case Sensitive

3.5. Comentarios

3.6. Tipos estáticos

3.7. Tipos de datos primitivos

3.8. Tipos Compuestos

3.9. Valor nulo (nil)

3.10. Secuencias de escape

4. Sintaxis del lenguaje GoLight

4.1. Bloques de Sentencias

4.2. Signos de agrupación

4.2.1. Variables

4.3. Operadores Aritméticos

4.3.1. Suma

4.3.2. Resta

- [4.3.3. Multiplicación](#)
 - [4.3.4. División](#)
 - [4.3.5. Módulo](#)
 - [4.3.6. Operador de asignación](#)
 - [4.3.7. Negación unaria](#)
- [4.4. Operaciones de comparación](#)
 - [4.4.1. Igualdad y desigualdad](#)
 - [4.4.2. Relacionales](#)
- [4.5. Operadores Lógicos](#)
- [4.6. Precedencia y asociatividad de operadores](#)
- [4.7. Sentencias de control de flujo](#)
 - [4.7.1. Sentencia If Else](#)
 - [4.7.2. Sentencia Switch - Case](#)
 - [4.7.3. Sentencia For](#)
- [4.8. Sentencias de transferencia](#)
 - [4.8.1. Break](#)
 - [4.8.2. Continue](#)
 - [4.8.3. Return](#)
- [5. Estructuras de datos](#)
 - [5.1. Slice](#)
 - [5.1.1. Creación de slice](#)
 - [5.1.2. Función slices.Index](#)
 - [5.1.3. Funcion strings.Join](#)
 - [5.1.4. Función len](#)
 - [5.1.5. Función append](#)
 - [5.1.6. Acceso de elemento:](#)
 - [5.2. Slices multidimensionales](#)
 - [5.2.1. Creación de matrices](#)
- [6. Structs](#)
 - [6.1. Definición:](#)
 - [6.2. Uso de atributos](#)
 - [6.3. Funciones de Structs](#)
- [7. Funciones](#)
 - [7.1. Declaración de funciones](#)
 - [7.1.1. Parámetros de funciones](#)
 - [7.2. Funciones Embebidas](#)
 - [7.2.1. Función fmt.Println](#)
 - [7.2.1.1. Tipos permitidos](#)
 - [7.2.2. Función strconv.Atoi](#)
 - [7.2.3. Función strconv.ParseFloat](#)
 - [7.2.4. Función reflect.TypeOf\(\).string](#)
- [8. Reportes Generales](#)
 - [8.1. Reporte de errores](#)
 - [8.2. Reporte de tabla de símbolos](#)

[8.3. Reporte de AST](#)

[9. Entregables](#)

[10. Restricciones](#)

[11. Consideraciones](#)

[12. Entrega del proyecto](#)

1. Competencias

1.1. Competencia General

Capacidad para desarrollar un intérprete 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 y Recorrido del AST: Construcción y manipulación eficiente del Árbol de Sintaxis Abstracta (AST) para la correcta interpretación del código.

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: Selección e implementación de estructuras de datos y herramientas apropiadas para la construcción del intérprete.

Manejo del AST: Desarrollo de técnicas de recorrido del Árbol de Sintaxis Abstracta (AST) para la generación de estructuras fundamentales como la tabla de símbolos, asegurando una interpretación y ejecución correcta del lenguaje.

2. Descripción

El objetivo del proyecto es que el estudiante desarrolle un intérprete para el lenguaje de programación GoLight, un lenguaje diseñado con una sintaxis inspirada en Go, pero adaptado para explorar conceptos fundamentales de compiladores.

Como parte del proyecto, se requiere también el desarrollo de una interfaz funcional que permita a los usuarios crear, editar, y ejecutar 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 intérprete 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, el intérprete deberá incluir etapas esenciales como la construcción del Árbol de Sintaxis Abstracta (AST), la evaluación semántica y un sistema robusto de manejo de errores, las cuales serán implementadas utilizando el entorno de desarrollo de C#.

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

Ejecutar: hará el llamado al intérprete, el cual se hará cargo de realizar los análisis léxico, sintáctico y semántico, además de interpretar todas las sentencias.

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.

Reporte de AST: Este reporte mostrará el Árbol de Sintaxis Abstracta (AST) generado a partir del código de entrada.

2.1.5. Consola

La consola es un área especial dentro del IDE que permite visualizar las notificaciones, errores, advertencias e impresiones 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 intérprete 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. El resultado del análisis es un Árbol de Sintaxis Abstracta (AST).
- En caso de errores léxicos o sintácticos, estos son reportados.

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.

Recorrido del AST:

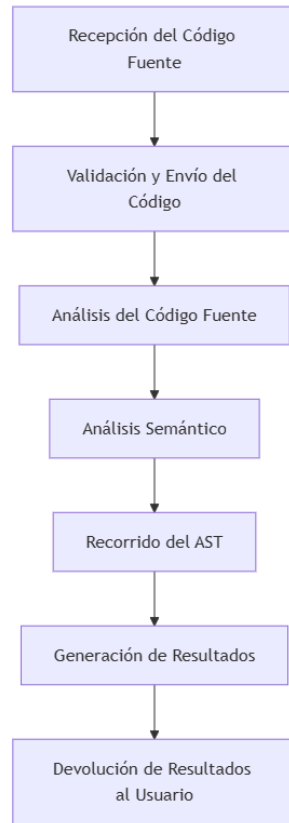
- El AST es recorrido para evaluar expresiones e instrucciones definidas en el lenguaje GoLight.

Generación de Resultados:

- Se generan los siguientes reportes:
 - **Reporte AST:** Representación visual del Árbol de Sintaxis Abstracta.
 - **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 muestra el resultado de la ejecución del código.

Devolución de Resultados al Usuario:

- Los reportes generados son enviados a la interfa, donde el usuario puede visualizarlos junto con el resultado de la ejecución.



Flujo de ejecución del proyecto

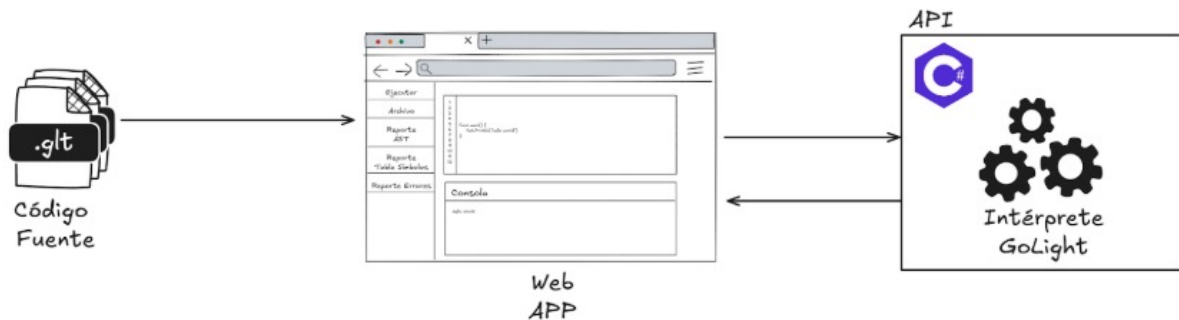
2.3. Tecnologías

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 intérprete y cualquier API, debe estar implementada en C#.

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 comunice 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 interpretación 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 intérprete 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 caracteres 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 intérprete 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.	$\pm 1.7E \pm 308$ (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 con 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
- Structs

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 := 10  
    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 intérprete 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 intérprete 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

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
<code>int [==,!=] int</code>	<code>bool</code>	<code>1 == 1 = true</code> <code>1 != 1 = false</code>
<code>float64 [==,!=] float64</code>	<code>bool</code>	<code>13.0 == 13.0 = true</code> <code>0.001 != 0.001 = false</code>
<code>int [==,!=] float</code> <code>float64 [==,!=] int</code>	<code>bool</code>	<code>35 == 35.0 = true</code> <code>98.0 = 98 = true</code>
<code>bool [==,!=] bool</code>	<code>bool</code>	<code>true == false = false</code> <code>false != true = true</code>
<code>string [==,!=] string</code>	<code>bool</code>	<code>"ho" == "Ha" = false</code> <code>"Ho" != "Ho" = false</code>
<code>rune [==,!=] rune</code>	<code>bool</code>	<code>'h' == 'a' = false</code> <code>'H' != 'H' = false</code>

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
<code>int[>,<,>=,<=] int</code>	<code>bool</code>	<code>1 < 1 = false</code>

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)  
}
```

```
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:

```
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:

```
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}

// Agregar un elemento
numeros = append(numeros, 4)
fmt.Println(numeros) // Salida: [1 2 3 4]
```

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])
```

5.2. Slices multidimensionales

En GoLight, los slices multidimensionales permiten almacenar datos de un solo tipo primitivo. A diferencia de las matrices, los slices pueden cambiar de tamaño durante la ejecución. La manipulación de datos se realiza utilizando la notación `[]`, y los índices comienzan desde 0.

5.2.1. Creación de matrices

Las matrices en GoLight pueden ser de **n a m dimensiones** pero solo de un tipo específico, además su tamaño será constante y será definido durante su declaración.

Consideraciones:

- La declaración del tamaño es implícita. Esto quiere decir que: No es necesario colocar el número de dimensiones, únicamente se debe declarar la lista de valores.
- La asignación y lectura valores se realizará con la notación `[]`
- Los índices de acceso comienzan a partir de 0
- Las matrices pueden cambiar su tamaño durante la ejecución.
- Si se hace un acceso con índices en fuera de rango se debe notificar como un error.

Ejemplo:

```
// Definición y asignación
mtx2 := [][]int{
    {0, 0, 0}, // Fila 1
    {0, 0, 0}, // Fila 2
    {0, 0, 0}, // Fila 3
}

// Asignación de valores
mtx2[0][0] = 7
mtx2[0][1] = 6
mtx2[0][2] = 5

fmt.Println(mtx2[0][1]) // Imprime 6

// Error: índices fuera de rango
// mtx1[100][100] = 10 // Esto generará un error
```

```

// Definición e inicialización directa
numeros := []int{1, 2, 3, 4}

// Slice multidimensional inicial
mtx1 := [][]int{
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
}

// Agregar el slice `numeros` como una nueva fila
mtx1 = append(mtx1, numeros)

fmt.Println(mtx1[3][2]) // 3

// Los slices dentro de un slice no tiene porque tener el mismo tamaño
matriz := [][]int{
    {1, 2, 3},      // Slice con 3 elementos
    {4, 5},        // Slice con 2 elementos
    {6, 7, 8, 9},  // Slice con 4 elementos
}

```

6. Structs

En GoLight, los structs son tipos compuestos que permiten al programador definir estructuras de datos personalizadas. Estos están compuestos por tipos primitivos u otros structs y son útiles para organizar y manipular información de manera versátil.

Un struct que contiene otro struct como una de sus propiedades puede ser del mismo tipo que el struct que lo contiene.

En el caso que un struct posea atributos de tipo struct, estos se manejan por medio de referencia así como sus instancias en el flujo de ejecución. Si un struct es el tipo de retorno o parámetro de una función, también se maneja por referencia.

6.1. Definición:

Consideraciones:

- Los structs **solo** pueden ser *declarados* en el ámbito global
- Cada struct debe tener al menos un atributo; no se permiten structs vacíos.
- Una vez definido, no es posible agregar, eliminar o modificar atributos de un struct.
- Un struct puede contener otros structs como atributos.

Ejemplo:

```
struct <NombreStruct> {  
    <Tipo> <NombreAtributo>;  
    ...  
}
```

Ejemplo:

```
struct Persona {  
    string Nombre;  
    int Edad;  
    bool EsEstudiante;  
}
```

6.2. Uso de atributos

Los atributos de un struct se acceden y modifican mediante el operador `..`. Este operador permite tanto leer el valor de un atributo como asignarle un nuevo valor.

Ejemplo:

```
struct Persona {  
    string Nombre;  
    int Edad;  
    bool EsEstudiante;  
}  
  
Persona miInstancia = { Nombre: "Alice", Edad: 25, EsEstudiante: false };  
  
// Acceso a atributos  
string nombre = miInstancia.Nombre; // "Alice"
```

```
// Modificación de atributos
miInstancia.Nombre = "Bob"; // Cambia el nombre a "Bob"
miInstancia.Edad = 30;      // Cambia la edad a 30
```

6.3. Funciones de Structs

GoLight permite asociar funciones a structs, las cuales siempre operan **por referencia**. Esto significa que las funciones pueden modificar directamente los valores de los atributos del struct al que pertenecen.

Consideraciones:

- Todas las funciones asociadas a un struct reciben una referencia al mismo, permitiendo modificar sus atributos directamente.
- Las funciones deben ser declaradas en el ámbito global y deben especificar el tipo del struct al que están asociadas.

Sintaxis:

```
func (<ReferenciaStruct> <NombreStruct>) <NombreFuncion>(<Parámetros>) <TipoRetorno> {
    // Implementación
}

miInstancia.<NombreFuncion>(<Argumentos>);
```

Ejemplo:

```
struct Persona {
    string Nombre;
    int Edad;
    bool EsEstudiante;
}

// Función asociada para modificar los atributos del struct
func (p Persona) ActualizarDatos(nuevoNombre string, nuevaEdad int, esEstudiante bool) {
    p.Nombre = nuevoNombre;
    p.Edad = nuevaEdad;
    p.EsEstudiante = esEstudiante;
}
```

```
}  
  
Persona miInstancia = { Nombre: "Alice", Edad: 25, EsEstudiante: false };  
  
fmt.Println("Nombre: ", miInstancia.Nombre); // Alice  
miInstancia.ActualizarDatos("Bob", 30, true);  
fmt.Println("Nombre: ", miInstancia.Nombre); // Bob
```

7. 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.

7.1. 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 o structs 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 structs y slices se pasan por referencia; los demás tipos (int, float64, string, bool, etc.) se pasan por valor.

7.1.1. 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 structs
    Producto p = { id: 1, nombre: "Producto A" };
    actualizarProducto(p, "Producto B");
    fmt.Println(p.nombre); // Salida: "Producto B"

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

```

7.2. 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.

7.2.1. 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.

7.2.1.1. Tipos permitidos

- **Primitivos:** `int`, `float64`, `bool`, `char`, `string`.
- **Slices:** Se imprimen en formato de lista `[valores]`.
- **Structs:** Se imprimen mostrando todos sus campos y valores en el formato `NombreStruct{Campo1: Valor1, Campo2: Valor2}`.

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]

struct Persona {
    string Nombre;
    int Edad;
    bool EsEstudiante;
}

p := Persona{Nombre: "Alice", Edad: 25, EsEstudiante: true}
fmt.Println("Struct:", p)
// Salida: Struct: Persona{Nombre: Alice, Edad: 25, EsEstudiante: true}
```

7.2.2. 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
```

7.2.3. 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
```

7.2.4. 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 structs, 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
```

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 Intérprete 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

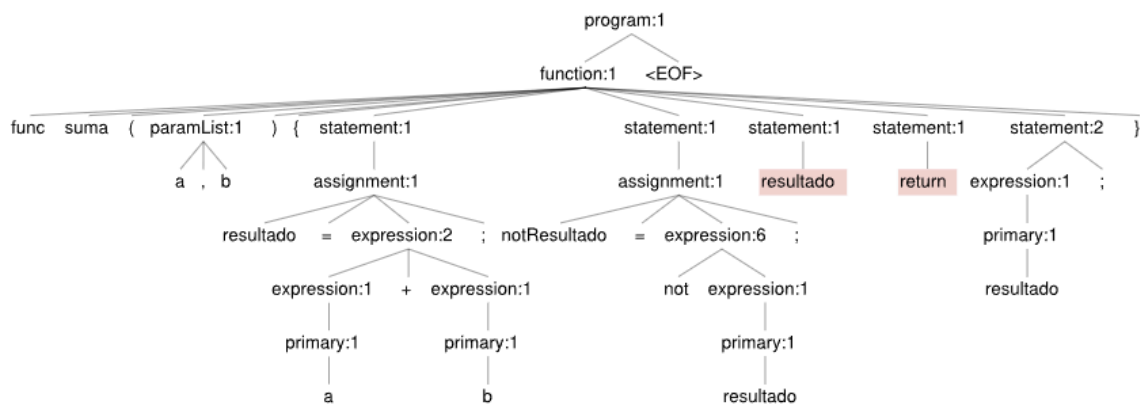
Este reporte mostrará la tabla de símbolos después de la ejecución del archivo. Se deberán de mostrar todas las variables, funciones y procedimientos que fueron declarados, así como su tipo y toda la información que el estudiante considere necesaria para demostrar que el Intérprete ejecutó correctamente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

8.3. Reporte de AST

Este reporte mostrará el Árbol de Sintaxis Abstracta (AST) generado a partir del código de entrada. El AST deberá incluir todas las estructuras sintácticas del programa, como declaraciones de variables, funciones, sentencias de control, expresiones y cualquier otro elemento del lenguaje.



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_Proyecto1_#Carnet**.
Ejemplo: OLC2_Proyecto1_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**.

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 virtual y se grabará para tener constancia o verificación posterior.
- 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: **14 de marzo de 2025 a media noche**.