Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Segundo semestre 2023
Catedráticos: Ing. Bayron López e Ing.Luis Espino
Tutores Académicos: Damihan Morales, Andres Rodas y Marco Mazariegos

T-Swift Compiler

1. Competencias

1.1. Competencia General

- Que el estudiante realice la fase de análisis de un compilador para un lenguaje de programación de alto nivel enfocado para el procesamiento de un lenguaje de programación utilizando herramientas para la generación de analizadores léxicos y sintácticos.

1.2. Competencias Específicas

- Que el estudiante utilice una herramienta léxica y una sIntáctica para el reconocimiento y análisis del lenguaje
- Que el estudiante implemente la traducción orientada por la sintaxis utilizando reglas semánticas haciendo uso de atributos sintetizados y/o heredados.
- Que el estudiante se familiarice con las herramientas y estructuras de datos disponibles para la creación de un compilador.
- Que el estudiante comprenda los fundamentos sobre la generación de una representación intermedia de un lenguaje de programación y cómo optimizarla.

2. Descripción

T-Swift es un lenguaje basado en el popular lenguaje de programación Swift siendo este un lenguaje multiparadigma que está ganando mucha popularidad debido a su sIntaxis moderna y diversas características distIntivas de un lenguaje moderno, además Incorpora características avanzadas como programación funcional, tipado estático, inferencia de tipos, entre otros. Esto lo convierte en un lenguaje moderno y eficiente, siendo apto para su estudio y comprensión para fines del laboratorio.

2.1. Componentes de la aplicación

Se requiere la implementación de un entorno de desarrollo que servirá para la creación de aplicaciones en lenguaje T-Swift, este IDE a su vez será el encargado de analizar el lenguaje de entrada. La aplicación contará con los siguientes componentes:

2.1.1. T-Swift IDE

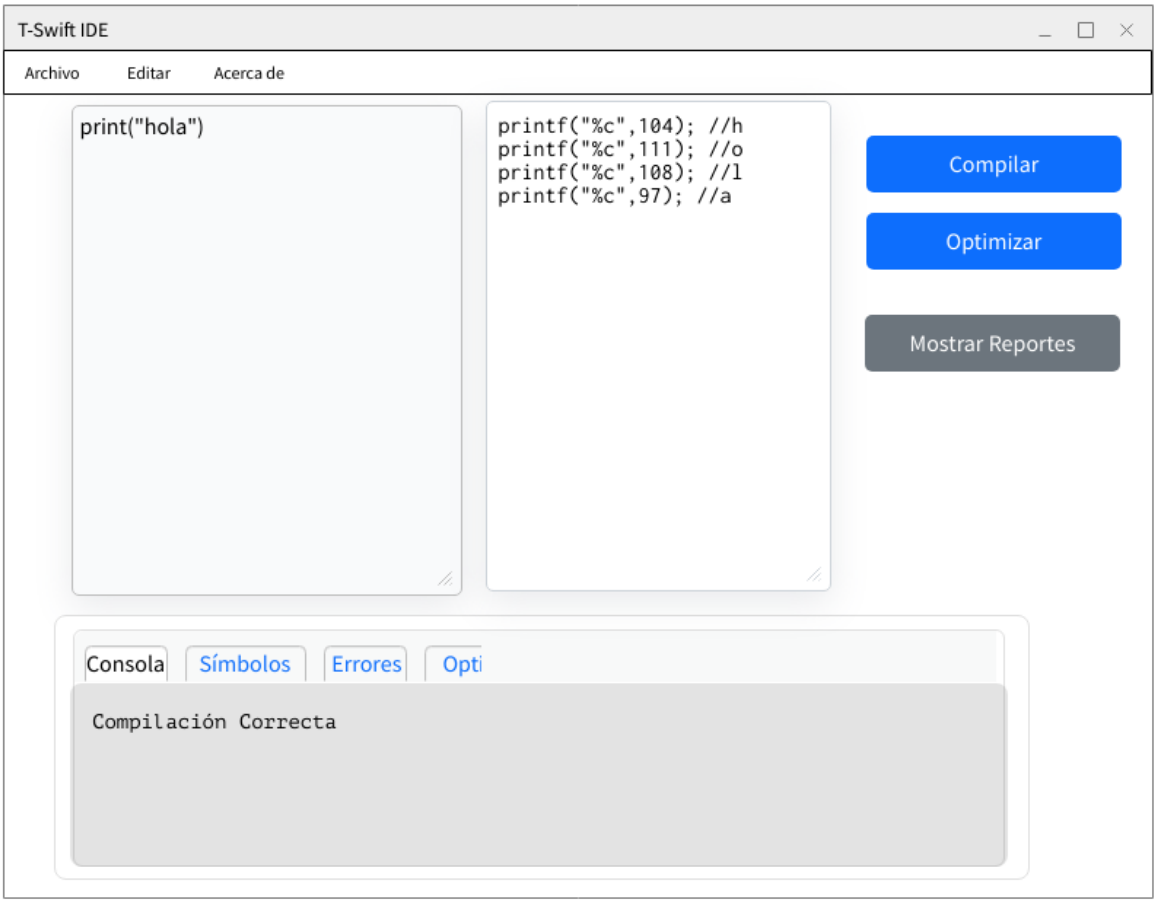
T-Swift IDE es un entorno de desarrollo que provee las herramientas para la escritura de programas en lenguaje T-Swift. Este IDE nos da la posibilidad de visualizar tanto la salida en consola como el código generado a partir del archivo fuente y los diversos reportes de la aplicación, que se explican más adelante. La Interfaz gráfica podrá ser desarrollada con la arquitectura y el framework a elección del estudiante, siempre y cuando se utilice el lenguaje y las herramientas indicadas por los tutores para el procesamiento y reconocimiento del lenguaje T-Swift.

2.1.2. Características Básicas

- Abrir, guardar y guardar como
- Editor de código
- Botón para compilar el archivo
- Menú con las opciones para realizar las optimizaciones.
- Reporte de errores en tiempo de compilación
- Reporte de tabla de símbolos
- Reporte de las optimizaciones realizadas
- Consola de salida de mensajes sobre el proceso de compilación

2.1.3. 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.

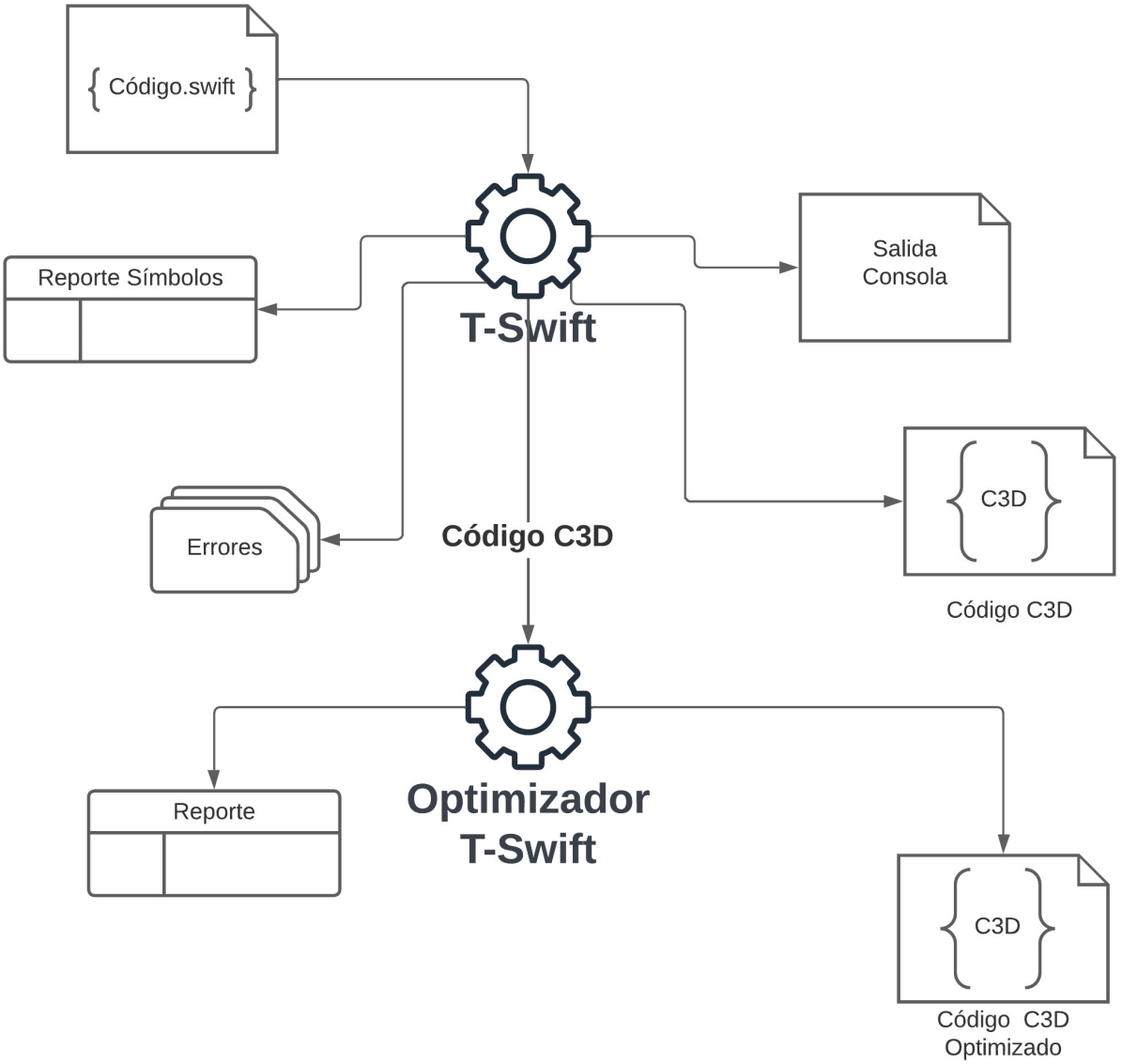


Interfaz propuesta

2.2. Flujo del proyecto

El flujo principal del proyecto comienza con un archivo en lenguaje T-Swift y concluye con la presentación de resultados en la consola, este proceso consta de los siguientes pasos:

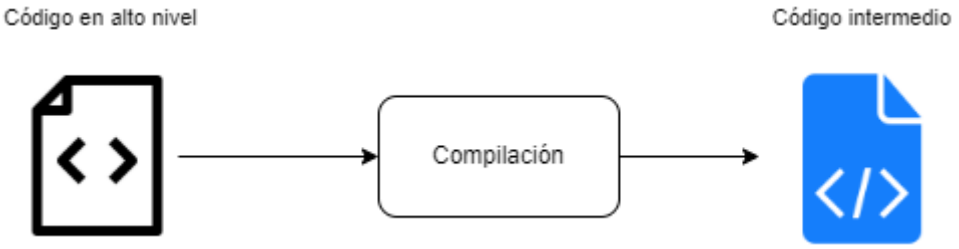
- El proceso de compilación.
- El proceso de optimización.
- El proceso de comprobación
- El proceso de ejecución.



Flujo de ejecución del T-Swift IDE

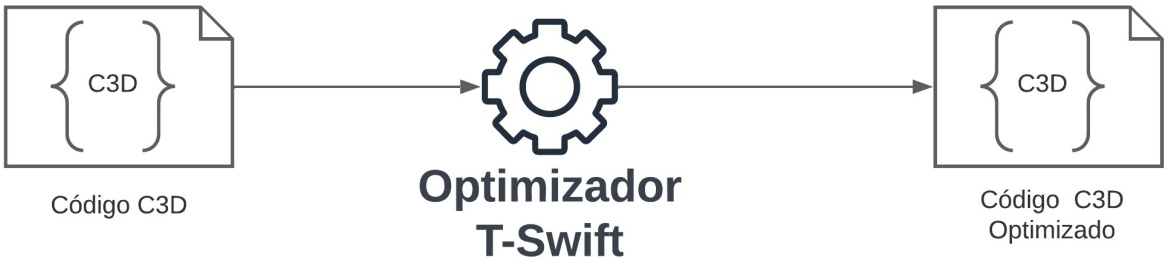
2.2.1. Proceso de compilación

El proceso de compilación T-Swift IDE recibe una entrada de código fuente en alto nivel generada por parte del usuario para generar una salida que será una representación intermedia en formato de código de tres direcciones, este formato utilizará sentencias del **lenguaje C** para su posterior ejecución.



2.2.2. Proceso de optimización

Un compilador puede introducir secuencias de código que contienen instrucciones innecesarias que consumen el tiempo de ejecución, por tal razón, T-Swift IDe deberá aplicar transformaciones de optimización del código generado para producir código más eficiente.
El proceso de optimización se hará por medio de la optimización por mirilla.
mirilla.

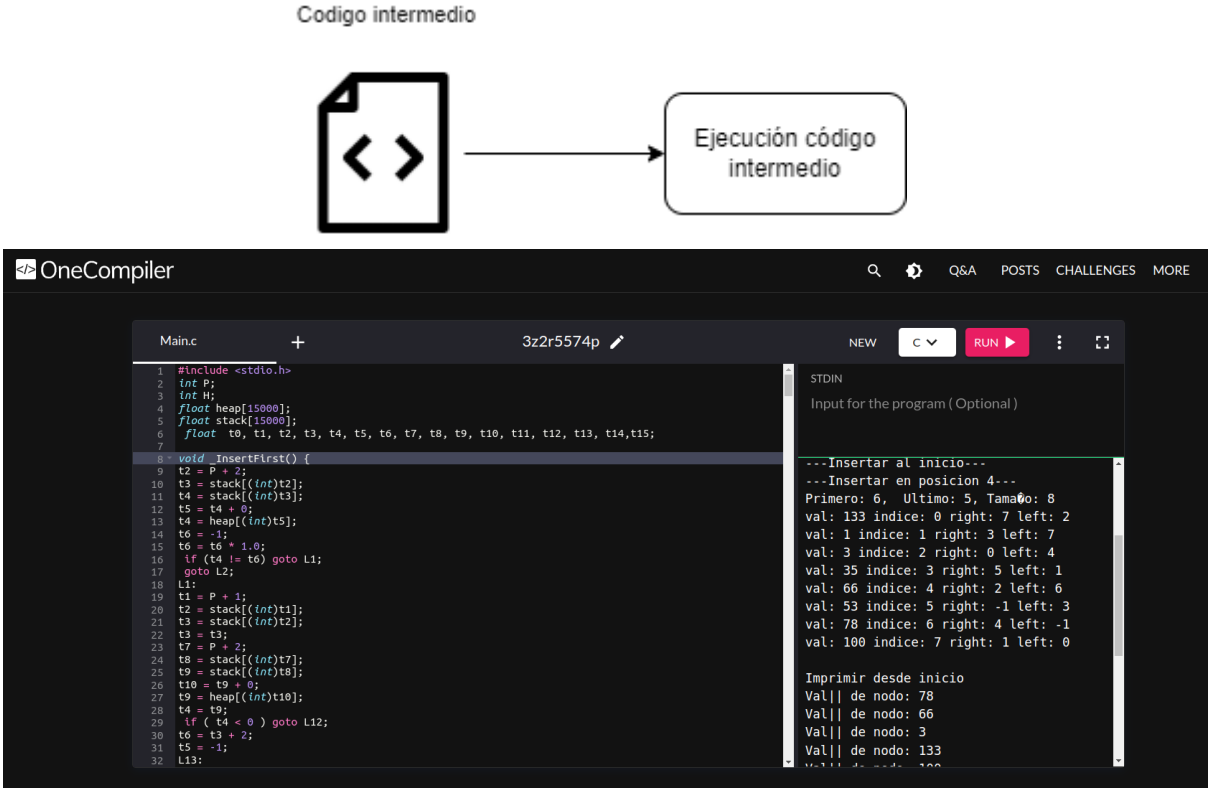


2.2.3. Proceso de de comprobación:

Como el código de tres direcciones utiliza sentencias del lenguaje C, se debe asegurar que el compilador genere el formato correcto antes de su ejecución, por lo cual, se debe de realizar el proceso de comprobación después de generar el código de tres direcciones. Para cumplir con los objetivos del proyecto, por este motivo estará disponible un analizador de código de tres direcciones desarrollado por los tutores de semestres anteriores para constatar que el código generado tenga la sintaxis correcta. Dicha herramienta se puede encontrar en el siguiente enlace.

2.2.4. Proceso de Ejecución

En el proceso de ejecución recibe como entrada el código de tres direcciones generado por T-Swift IDE, ya sea compilado o no y será ejecutado en un compilador en línea del lenguaje de programación C, **únicamente se ejecutará sí el código de tres direcciones tiene el formato correcto.**



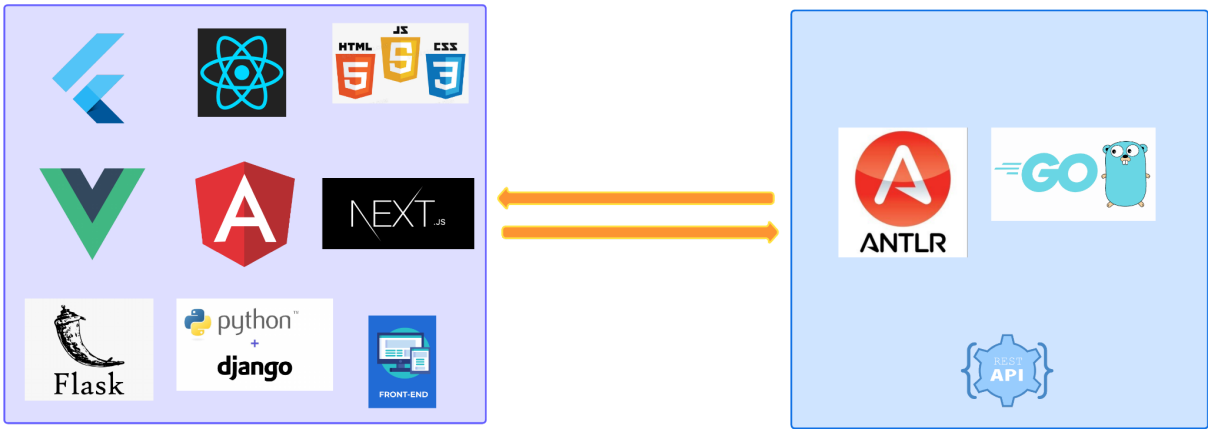
Ejecución y compilación del código de tres direcciones en línea

2.3. Arquitecturas propuestas

Para la implementación de una solución se sugieren las siguientes arquitecturas, el estudiante tiene la libertad de escoger la herramienta y/o librería a su elección para el desarrollo de la GUI del IDE

2.3.1. Stack: Frontend / Api - REST

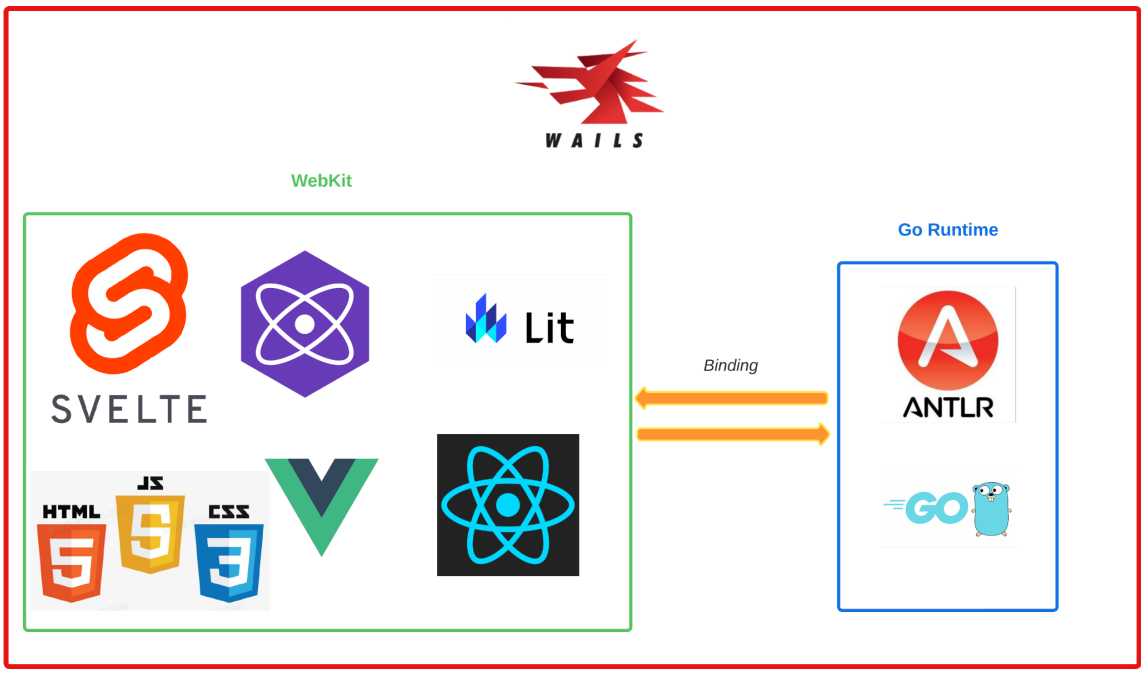
Es permitido crear una Api-REST en Go y luego consumirla con un frontend, la tecnología para desarrollar el frontend queda a discreción del estudiante. Dicha aplicación puede ser implementada de forma local y el framework / librería en Go para construir la Api queda a discreción del estudiante .



2.3.2. Aplicación de Escritorio

Se puede implementar una aplicación de escritorio con Go, se sugieren las siguientes tecnologías, sin embargo el framework para desarrollar la GUI queda a discreción del estudiante

- **Wails**: Un framework multiplataforma que utiliza Webkit para crear interfaces con los frameworks: Svelta, React, Preact, Lit, Vue.js y Vanila y conectarla con aplicaciones hechas con Go por medio de un Binding.



- **Fyne**: Es un framework multiplataforma basado en Material Design para la creación de GUI de forma nativa..

Consideraciones:

- La implementación de una GUI para la aplicación es **obligatoria** no se calificarán aplicaciones en consola ni Apis consumidas de forma manual

3. Generalidades del lenguaje T-Swift

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

3.1. Expresiones en el lenguaje T-Swift

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:

T-Swift por su naturaleza carece de una función **main**, por ello se debe considerar generar el código intermedio de tal forma que este código posea el comportamiento esperado.

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 _
- Por simplicidad el identificador no puede contener carácteres especiales (.$,-, etc)
- El identificador no puede comenzar con un número.

3.4. Case Sensitive

El lenguaje T-Swift 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 T-Swift no soportará múltiples asignaciones de diferentes tipos para una variable, por lo tanto si una variable es asignada con un tipo, solo será posible asignar valores de ese tipo a lo largo de la ejecución, sí alguna variable se le asignase un valor que no corresponde a su tipo declarado, el programa debe mostrar un mensaje detallando el error.

3.7. Tipos de datos primitivos

Se utilizan los siguientes tipos de datos primitivos:

Tipo primitivo	Definición	Rango (teorico)	Valor por defecto
`Int`	**Acepta valores números enteros**	**-2,147,483,648 a +2,147,483,647**	`nil`
`Float`	**Acepta valores numéricos de punto flotante**	**1.2E-38 a 3.4E+38 (6 dígitos de precisión)**	`nil`
`String`	**Acepta cadenas de caracteres**	**[0, 65535] caracteres (acotado por conveniencia)**	`nil`
Bool	**Acepta valores lógicos de verdadero y falso**	**true false**	`nil`
`Character`	**Acepta un solo caracter ASCII**	**[0, 65535] caracteres**	`nil`

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 `Float`
- 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, `nil` esto para evitar la lectura de basura durante la ejecución.
- El literal `0` se considera tanto de tipo `Int` como `Float`.

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 T-Swift.

- Vectores
- Matrices
- Structs

Estos tipos especiales se explicarán más adelante.

3.9. Valor nulo (`nil`)

En el lenguaje T-Swift se utiliza la palabra reservada `nil` para hacer referencia a la nada, esto indicará la ausencia de valor, por lo tanto cualquier operación sobre `nil` será considerada un **error y dará** `nil` como resultado.

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**
\r	**Tabulación**

4. SIntaxis del lenguaje T-Swift

A continuación, se define la sIntaxis para las sentencias del lenguaje T-Swift

4.1. Bloques de Sentencias

Será un conjunto de sentencias delimitado por llaves “{ }”, cuando se haga referencia a esto querrá decir que se está definiendo un ámbito local con todo y sus posibles instrucciones y que tiene acceso a las variables del ámbito global, además las variables declaradas en dicho ámbito únicamente podrán ser utilizadas en este ámbito o en posibles ámbitos anidados

```
// <sentencias de control>
{
// sentencias
}

var i = 10 // variable global es accesible desde este ámbito
if i == 10
{
    var j:Int = 10 + i // i es accesible desde este ámbito
    if  i == 10
    {
        var k: Int = j + 1    // i y j son accesibles desde este ámbito
    }
    j = k // error k ya no es accesible en este ámbito
}
```

Consideraciones:

- Estos bloques estarán asociados a alguna sentencia de control de flujo por lo tanto no podrán ser declarados de forma independiente.

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
```

4.3. Variables

Una variable es un elemento de datos cuyo valor puede cambiar durante el curso de la ejecución de un programa **siempre y cuando sea el mismo tipo de dato**. Una variable cuenta con un nombre y un valor, los nombres de variables no pueden coincidir con una palabra reservada.

Para poder utilizar una variable se tiene que definir previamente, la declaración nos permite crear una variable y asignarle un valor o sin valor.

Además la definición de tipo durante la declaración puede ser implícita o explícita, es explícita cuando se indica el tipo del cual será la variable e implícita cuando esta toma el tipo del valor al cual se está asignando.

SIntaxis:

```
// declaración con tipo y valor
var <identificador> : <Tipo> = <Expresión>
// declaración con valor
var <identificador> = <Expresión>
// declaración con tipo y sin valor
var <identificador> : <Tipo> ?
```

Consideraciones:

- Las variables solo pueden tener **un tipo de dato** definido y este no podrá cambiar a lo largo de la ejecución.
- Solo se puede declarar **una** variable por sentencia.
- Si la variable ya existe se debe actualizar su valor por el nuevo, validando que el nuevo valor sea del mismo tipo del de la variable.
- El nombre de la variable **no puede** ser una **palabra reservada** ó del de una variable **previamente** definida.
- El lenguaje al ser case sensitive distinguirá a las variables declaradas como por ejemplo `id` y `Id` se toman como dos variables diferentes.
- Si la expresión tiene un tipo de dato **diferente** al definido previamente se tomará como **error** y la variable obtendrá el valor de `nil` para fines prácticos.
- Cuando se asigna un valor de tipo ***Int*** a una variable de tipo ***Float*** el valor será considerado como ***Float***, esta es la única conversión implícita que habrá.

Ejemplo:

```
// declaración de variables
```

```
var valor: String? //correcto, declaración sin valor

// correcto, declaración de una variable tipo Int con valor
var valor1 = 10

var valor1:Int = 10.01 // Error: no se puede asignar un Float a un Int

var valor2:Float = 10.2 // correcto

var valor2_1:Float = 10 + 1 // correcto será 11.0

var valor3 = "esto es una variable";  //correcto variable tipo String

var char:Character = "A";  //correcto variable tipo Character

var valor4:Bool = true   //correcto

//debe ser un error ya que los tipos no son compatibles
var valor4:String = true

// debe ser un error ya que existe otra variable valor3 definida
previamente
var valor3:Int = 10

var .58 = 4; // debe ser error porque .58 no es un nombre válido

var if = "10" // debe ser un error porque "if" es una palabra reservada

// ejemplo de asignaciones

valor1 = 200 // correcto

valor3 = "otra cadena" //correcto

valor4 = 10;  //error tipos incompatibles (Bool, Int)

valor2 = 200 // correcto conversión implícita (Float, Int)

char = "otra cadena" //error tipos incompatibles (Character, String)
```

4.4. Constantes

Las constantes en el lenguaje T-Swift se definen anteponiendo la palabra reservada **`let`**, esto indica que el valor de dicha constante no podrá cambiar a lo largo de la ejecución, por **lo tanto no será válida cualquier asignación** que se haga sobre una constante y deberá ser notificada como un **error.**

Consideraciones:

- Solo será posible crear constantes con tipos de datos primitivos
- Una constante no puede ser definida sin un valor específico.
- El nombre de las constantes tiene las mismas propiedades que las variables.
- Las constantes cuentan con declaración con tipo de forma implícita y explícita como las variables.

Ejemplo

```
// declaración de constantes

//Incorrecto, la constante debe tener un valor asignado
let valor: String?

// correcto, declaración de una constante tipo Int con valor
Let valor1 = 10

let valor1:Int = 10.01 // Error: no se puede asignar un Float a un Int

let valor2:Float = 10.2 // correcto

let valor3 = "esto es una variable";  //correcto constante tipo String

let valor4:Bool = true   //correcto

let .58 = 4; // debe ser error porque .58 no es un nombre válido

let if = "10" // debe ser un error porque "if" es una palabra reservada

// error valor1 al ser una constante no puede cambiar su valor
valor1 = 200
```

4.5. Operadores Aritméticos

Los operadores aritméticos toman valores numéricos de expresiones y retornan un valor numé rico **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 %.

4.5.1. Suma

La operación suma se produce mediante la suma de tipos numéricos o Strings concatenados, debido a que T-Swift 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 + Float`	`Int` `Float`	`1 + 1 = 2` `1 + 1.0 = 2.0`
`Float + Float` `Float + Int`	`Float` `Float`	`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.

4.5.2. 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 - Float`	`Int` `Float`	`1 - 1 = 0` `1 - 1.0 = 0.0`
`Float - Float` `Float - Int`	`Float` `Float`	`1.0 - 13.0 = -12.0` `1.0 - 1 = 0.0`

4.5.3. 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 * Float`	`Int` `Float`	`1 * 10 = 10` `1 * 1.0 = 1.0`
`Float * Float` `Float * Int`	`Float` `Float`	`1.0 * 13.0 = 13.0` `1.0 * 1 = 1.0`

Consideraciones:

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

4.5.4. 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 / Float`	`Int` `Float`	`10 / 3 = 3` `1 / 3.0 = 0.3333`
`Float / Float` `Float / Int`	`Float` `Float`	`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 una advertencia y por conveniencia el resultado será un valor `nil`.

4.5.5. 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 advertencia y por conveniencia el resultado será un valor `nil`.

4.5.6. Operador de asignación

4.5.6.1. Suma

El operador += indica el incremento del valor de una **expresión** en una **variable** de tipo ya sea `Int` o de tipo `Float` . 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:

```
var var1:Int = 10
var var2:Float = 0.0

var1 += 10 //var1 tendrá el valor de 20

var1 += 10.0 // error, no puede asignar un valor de tipo Float a un Int

var2 += 10 // var2 tendrá el valor de 10.0
```

```
var2 += 10.0 //var tendrá el valor de 20.0

var str:String = "cad"

str += "cad" //str tendrá el valor de "cadcad"

str += 10 //operación inválida String + Int
```

4.5.6.2. Resta

El operador -= indica el decremento del valor de una **expresión** en una **variable** de tipo ya sea `Int` o de tipo `Float` . 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:

```
var var1:Int = 10
var var2:Float = 0.0

var1 -= 10 //var1 tendrá el valor de 0

var1 -= 10.0 // error, no puede asignar un valor de tipo Float a un Int

var2 -= 10 // var2 tendrá el valor de -10.0

var2 -= 10.0 //var tendrá el valor de -20.0
```

4.5.7. 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`
`-Float`	`Float`	`-(1.0) = -1.0`

4.6. 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*.

4.6.1. 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`
`Float [==,!=] Float`	`Bool`	`13.0 == 13.0 = true` `0.001 != 0.001 = false`
`Bool [==,!=] Bool`	`Bool`	`true == false = false` `false != true = true`
`Character [==,!=] Character`	`Bool`	`"ho" == "Ha" = false` `"Ho" != "Ho" = false`
`String [==,!=] String`	`Bool`	`"ho" == "Ha" = false` `"Ho" != "Ho" = false`

Consideraciones

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

4.6.2. Relacionales

Las operaciones relacionales que soporta el lenguaje T-Swift 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`
`Float [>,<,>=,<=] Float`	`Bool`	`13.0 >= 13.0 = true`
`String [>,<,>=,<=] String`	`Bool`	`"aAA" <= "bA" = true`
`Character [>,<,>=,<=] Character`	`Bool`	`"a" <= "b" = true`

Consideraciones

- Cualquier otra combinación será inválida y se deberá reportar el error.
- La limitación de las operaciones también se aplica a comparación de literales.

4.7. 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.

4.8. 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.

4.8.1. Sentencia If Else

Ejecuta un bloque de sentencias si una condición especificada es evaluada como verdadera. Si la condición es evaluada como falsa, otro bloque de sentencias puede ser ejecutado.
SIntaxis:

```
SI -> if  <EXPRESIÓN> { <BLOQUE_SENTENCIAS> }
| if <EXPRESIÓN> { <BLOQUE_SENTENCIAS> } else { <BLOQUE_SENTENCIAS>}
| if <EXPRESIÓN> { <BLOQUE_SENTENCIAS> } else SI
```

Ejemplo:

```
if 3 < 4 {
// Sentencias
} else if 2 < 5 {
// Sentencias
} else {
// Sentencias
}
if true { // Sentencias }
if false {// Sentencias } else { // Sentencias }
if false {// Sentencias } else if true { // Sentencias }
```

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.

4.8.2. Sentencia Switch - Case

Evalúa una expresión, comparando el valor de esa expresión con un case, y ejecuta declaraciones asociadas a ese case, así como las declaraciones en los case que siguen.

Si ocurre una coincidencia, el programa ejecuta las declaraciones asociadas correspondientes. Si la expresión coincide con múltiples entradas, la primera será la seleccionada. Si no se encuentra una cláusula de case coincidente, el programa busca la cláusula **`default`** opcional, y si se encuentra, transfiere el control a ese bloque, ejecutando las declaraciones asociadas. Si no se encuentra un default el programa continúa la ejecución en la instrucción siguiente al final del switch. Por convención, el default es la última cláusula. La declaración break es opcional, puede o no estar entre las sentencias de cada bloque.

Sintaxis:

```
case -> switch  <Expresión>  {
case expr1:
# Declaraciones ejecutadas cuando el resultado de expresión coincide con
el expr1
break?
case expr2:
# Declaraciones ejecutadas cuando el resultado de expresión coincide con
el expr2
break?
...
case exprN:
# Declaraciones ejecutadas cuando el resultado de expresión coincide con
exprN
break?
default:
# Declaraciones ejecutadas cuando ninguno de los valores coincide con el
valor de la expresión
break?
}
```

Ejemplo:

```
let numero = 2
switch numero {
  case 1:
    print("Uno")
  case 2:
    print("Dos")
  case 3:
    print("Tres")
  default:
    print("Invalid day")
}
/* Salida esperada:
Dos
*/
```

4.8.3. Sentencia While

Crea un bucle que ejecuta un bloque de sentencias especificadas mientras cierta condición se evalúe como verdadera (**true**). Dicha condición es evaluada **antes** de ejecutar el bloque de sentencias y al final de cada iteración.

SIntaxis:

```
WHILE -> while  <Expresión>  {
 <BLOQUE SENTENCIAS>
}
```

Ejemplo:

```
while true {
   //sentencias
}
var num =  10
while num != 0 {
    num -= 1
    print(num)
}
/* Salida esperada:
9
8
7
6
5
4
3
2
1
0
*/
```

Consideraciones:

- El ciclo while recibirá una expresión de tipo **Bool**, en caso contrario deberá mostrar un error.

4.8.4. Sentencia For

Un bucle **for** en el lenguaje T-Swift se comportará como un for moderno, que recorrerá alguna estructura compuesta, en este caso sobre: **cadenas, arreglos unidimensionales** y **rangos**. La variable que recorre los valores se comportará como una constante, por lo tanto no se podrán modificar su valor en el bloque de sentencias, su valor únicamente cambiará con respecto a las iteraciones.

SIntaxis:

```
for  <ID> in <EXPRESIÓN | RANGO>  {
```

```
    <BLOQUE SENTENCIAS>
}
```

Ejemplo

```
// for que recorre un rango
for i in 1...5 {
    print(i)
}
// for que recorre una cadena
for c in "hola" {
    print(c)
}

let letras = ["O", "L", "C", "2"]
// for que recorre un arreglo unidimensional
for letra in letras {
    print(letra)
    letra = "cadena" //error no es posible asignar algo a letra
}
/*Salida esperada:
1
2
3
4
5
h
o
l
a
O
L
C
2
*/
```

Consideraciones:

- La declaración de rangos será únicamente propia de la sentencias for, no se utilizarán en otras áreas del lenguaje, además los rango únicamente serán de valores de tipo **`Int`**.
- Los valores de los rangos deben estar ordenados, la expresión de la derecha debe ser siempre mayor o igual a la de la izquierda, en caso contrario se tomará como un error.

- La constante que recorre un arreglo, su tipo será del tipo de dato que contiene el arreglo, cuando recorre un rango será de tipo **`Int`** y cuando recorre una cadena será de tipo **`Character`**.

4.8.5. Guard

Esta sentencia perteneciente al lenguaje T-Swift su finalidad es transferir el control del programa *fuera* del ámbito actual cuando no se cumple cierta condición, a diferencia de la sentencia **`if`** guard evalúa la expresión y si es **falsa** ejecuta el bloque **`else`**. Guard está pensado para ser utilizado exclusivamente para este fin, por lo tanto requiere que el bloque de sentencias termine con una sentencia de tipo **`break`**, **`return`** o **`continue`**, en caso contrario será considerado un error.

Sintaxis:

```
GUARD -> guard  <Expresión> else {
 <BLOQUE SENTENCIAS>
<continue | break | return>
}
```

Ejemplo:

```
var i = 2

while (i <= 10) {
  // la sentencia guard verifica si i es impar
  guard i % 2 == 0 else {
     i = i + 1
    continue
  }
  print(i)
  i = i + 1
}
/* Salida
2
4
6
8
10
*/
```

Consideraciones:

- En caso que la expresión no fuese de tipo **`Boolean`** se considerará un error.

4.9. Sentencias de transferencia

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

4.9.1. Break

Esta sentencia termina el bucle actual ó sentencia switch y transfiere el control del programa a la siguiente sentencia a la sentencia de terminación de estos elementos.

Ejemplo:

```
while  true {
i = 0;
break; //finaliza el bucle en este punto
}
```

Consideraciones:

- Si se encuentra un **break** fuera de un ciclo y/o sentencia switch se considerará como un error.

4.9.2. Continue

Esta sentencia termina la ejecución de las sentencias de la iteración actual (en un bucle) y continúa la ejecución con la próxima iteración.

Ejemplo:

```
while 3 < 4 {
    continue
}
var i = 0;
var j = i;
while i < 2 {
    if  j == 0{
          i = 1;
          i += 1
          continue;
    }
    i += 1
}
// i posee el valor de 2 al finalizar el ciclo
```

Consideraciones:

- Si se encuentra un **continue** fuera de un ciclo se considerará como un error.

4.9.3. 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 funcion1() -> Int {
    return 1;  // retorna un valor Int
}

func funcion() {
   return; // no retorna nada
}
```

5. Estructuras de datos

Las estructuras de datos en el lenguaje T-Swift son los componentes que nos permiten almacenar un conjunto de valores agrupados de forma ordenada, las estructuras básicas que incluye el lenguaje son las siguientes: Vectores y Matrices

5.1. Vectores

Los vectores son la estructura compuesta más básica del lenguaje T-Swift, los tipos de vectores que existen son con base a los tipos **primitivos** y **structs** del lenguaje. Su notación de posiciones por convención comienza con 0.

5.1.1. Creación de vectores

Para crear vectores se utiliza la siguiente sIntaxis.
SIntaxis:

```
<Declaracion_vector> ->
           var Id : [<TIPO>] = ( <Defincion_vector> )
           var Id <Defincion_vector_Alt>
<Definicion_vector> ->
           [<LISTA EXPRESIONES>?]
           | []
           | <Expresion>
<Defincion_vector_Alt> -> = [<TIPO>] ()
```

Consideraciones:

- Un vector al declararse sin elementos se convertirá en un vector vacío
- La lista de expresiones deben ser del mismo tipo que el tipo del vector.
- El tamaño del vector puede aumentar o disminuir a lo largo de la ejecución.
- Cuando la definición de un vector sea otro vector, *se hará una copia* del vector dando origen a otro nuevo vector con los mismos datos del vector.

5.1.2. Función append(<Expresión>)

Esta función se encarga de colocar insertar un valor al final del vector, no retorna nada.

5.1.3. Función removeLast()

Esta función remueve el último elemento de la lista, en caso que el vector este vació deberá mostrarse un mensaje de **error**, no retorna nada.

5.1.4. Función remove(at: <Expresion>)

Esta función remueve un elemento en la posición dada, no retorna nada, en caso que la posición no exista, se deberá mostrar un mensaje de **error**, no retorna nada.

5.1.5. IsEmpty

Este atributo indica si el vector se encuentra vacío o no, retornando **`true`** o **`false`** según sea el caso.

5.1.6. count

Este atributo indica la cantidad de elementos que posee el vector, dicha cantidad la devuelve con un valor de tipo Int

5.1.7. Acceso de elemento:

Los vectores 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 y retornar el valor de nil.

Ejemplo:

```
//vector con valores
var vec1: [Int] = [10,20,30,40,50]
//vector vacío
var vec2: [Float] = []
//vector vacío
var vec3: [String] = [ ];

//imprime 0
print(vec2.count)
```

```
//inserta 100 al final
vec1.append(100) //[10,20,30,40,50,100]

//inserciones en vacíos
vec2.append(1.0) // [1.0]
vec3.append("cadena") // ["cadena"]

//elimina la primera posición
vec1.remove( at: 0) //[20,30,40,50,100]

//elimina la última posición
vec1.remove(at: vec1.count - 1) //[20,30,40,50]

//se realiza una copia completa de vector
var copiaVec: [Int] = vec1

//Aceso a un elemento
let val: Int = vec1[3] // val = 50

//asignación con []
vec1[1] = vec1[0] //[20,20,40,50]

//vec1 = [20,20,40,50]
//copiaVec = [20,30,40,50]
```

5.2. Matrices

Las matrices en T-Swift nos permiten almacenar solamente datos de ***tipo primitivo***, la diferencia principal entre el vector y la matriz es que esta última organiza sus elementos en n dimensiones y la manipulación de datos es con la notación [ ] *además que* ***su tamaño no puede cambiar en tiempo de ejecución.***

5.2.1. Creación de matrices

Las matrices en T-Swift pueden ser de **2 a n dimensiones** pero solo de un tipo específico, además su tamaño será constante y será definido durante su declaración.

Consideraciones:

- La declaración del tamaño puede ser explícita o en base a se definición.
- Si la declaración es explícita pero su definición no es acorde a esta declaración se debe marcar como un error. Por lo tanto se deben verificar que la cantidad de dimensiones sea acorde a la definida.
- La asignación y lectura valores se realizará con la notación [ ]

- Los índices de declaración comienzan a partir de `1`
- Los índices de acceso comienzan a partir de `0`
- Las matrices **no** van a cambiar su tamaño durante la ejecución.
- Si se hace un acceso con índices en fuera de rango se devuelve **`nil`** y se debe notificar como un error.
- Si se declara una matriz con índices negativos o 0, será considerado un error
- El atributo `count` solo recibirá **número enteros** en forma de literales, no podrán ser asignadas ni variables ni elementos de otras estructuras a este atributo.

SIntaxis:

```
Decl_Mat -> var Id (: Tipo_Matriz)? = Defincion_ Mat

Tipo_Matriz -> [ Tipo_Matriz ]
             | [ <Tipo_Primitivo> ]

Defincion_Mat -> ListaValores_Mat
           |     Simple_Mat

ListaValores_Mat -> [ ListaValores_Mat2 ]

ListaValores_Mat2 -> ListaValores_Mat2 , ListaValores_Mat
                  |  ListaValores_Mat
                  |  <Lista_Expresiones>

Simple_Mat -> Tipo_Matriz ( repeating: Simple_Mat  , count: <Entero> )
           |  Tipo_Matriz ( repeating: <Expresión>, count: <Entero> )
```

Ejemplo:

```
var matrix : [[[Int]]] = [[[Int]]] (repeating: [[Int]] (repeating: [Int]
(repeating: 0, count:2), count:3), count:4)

/*
esta matriz sería:
[
  [
   [0, 0], [0, 0], [0, 0]
  ],
  [
   [0, 0], [0, 0], [0, 0]
```

```
  ],
  [
   [0, 0], [0, 0], [0, 0]
  ],
  [
   [0, 0], [0, 0], [0, 0]
  ]
]
*/

//otro ejemplo
var  matrix0 : [[[String]]] = [[[String]]] (repeating: [[String]]
(repeating: [String] (repeating:"OLC2", count:2), count:1), count:3)
/*
[
 [
  ["OLC2", "OLC2"]
 ],
 [
  ["OLC2", "OLC2"]
 ],
 [
  ["OLC2", "OLC2"]
 ]
]
*/
// declaración mediante definición
var mtx1 : [[Int]] = [[1,2,3],[4,5,6],[7,8,9]]

var mtx2 : [[[Int]]] =  [[[1,2,3],[4,5,6],[7,8,9]],
[[10,11,12],[13,14,15],[16,17,18]],
[[19,20,21],[22,23,24],[25,26,27]]]


//asignacion de valores
mtx1[1][1] = 10  //cambia 5 por 10
print(mtx1[0][0] ) //imprime 1
print(mtx2[0][1][2]) //imprime 6

//error indices fuera de rango - error
mtx1[100][100] = 10
```

6. Structs

El lenguaje T-Swift tiene la capacidad de permitir al programador en crear sus propios tipos compuestos personalizados, estos elementos se les denomina structs, los structs permiten la creación de estructuras de datos y manipulación de información de una manera más versátil. Estos están compuestos por *tipos primitivos* o por otros structs. Los structs también pueden ser utilizados por funciones y arrays.

Los struct en el lenguaje T-Swift se manejan por **valor**, esto implica que los structs que contenga un struct *no pueden ser del mismo tipo que el struct que los contiene,* esto se aplica para referencias de forma directa e indirecta.

En el caso que un struct posea atributos de tipo struct, estos se manejan por medio de valor 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:

- Si un atributo no posee un valor por defecto, entonces se debe establecer dicho valor de forma obligatoria en el constructor, en caso contrario será un error ya que no será posible que hayan atributos sin valor.
- Los atributos pueden ser mutables (var) o inmutables(let), si un atributo es inmutable sólo se podrá asignarle un valor de alguna de las siguientes formas:
    - En la declaración del atributo
    - En el constructor
- Los struct **solo** pueden ser *declarados* en el ámbito global
- Los structs pueden tener o no tener atributos.
- Será posible declarar funciones dentro de los structs, siendo estas declaradas como mutables o inmutables.
    - Una función *mutable* permite modificar los atributos del struct del cual se hace la llamada de dicha función, para esto se utiliza el atributo `self`
    - Una función *inmutable* no permite modificar los atributos del struct del cual se hace la llamada de dicha función, solo permite el acceso y lectura de los mismos.
    - Las funciones pueden tener todas las características del lenguaje

SIntaxis:

```
Def_Struct -> struct ID {   Lista_Atributos *    }

Lista_Atributos ->  (let | var ) ID  (: <TIPO> ' )? (= <Expresion>)?
                        | mutating? <Declaracion de Funcion>
```

6.2. Creación de structs

El lenguaje T-Swift ofrece diversas formas de crear structs las cuales son las siguientes:

- Creación de structs con constructor con Dupla - Expresión: Es una lista duplas donde consta de dos valores *ID* y *Expresion* , el ID indica el nombre del atributo del struct y la expresión el valor asignado, en esta lista pueden o no indicarse el valor de todos los atributos del struct, los atributos que quedan sin valor deben tener un valor por defecto en la declaración, en caso contrario será un error, además el orden de los atributos debe ser el mismo en el que fueron declarados.
    - Si un atributo es declarado como inmutable y tienen un valor asignado en la declaración, este no podrá ser modificado con el constructor, si esto llegase a pasar será considerado un error.
- Copia por **valor** de otro struct del mismo tipo: Es cuando se asigna un struct a otro, en este proceso se realiza una copia completa del struct asignado, ya sea por medio de una variable o por la llamada de una función que devuelve un struct.

SIntaxis

```
Struct_Expr -> (var|let) ID (: ID)? =  ID( L_Dupla ) ?
             | (var|let) ID (: ID)? =  ID ?
             | (var|let) ID (: ID)? =  <llamada_expresión>

L_Dupla  -> (ID : <Expresión> )*
```

6.3. Uso de atributos

El lenguaje T-Swift permite la edición y acceso a atributos de los structs por medio del operador '.' , el cual nos permite acceder a los atributos ya sea para asignarles un valor ó acceder al valor.

SIntaxis:

```
Atributo -> ID( .ID)+
```

Ejemplo:

```
//struct con atributo sin valor por defecto
// y con un atributo con valor por defecto
struct Persona{
    var Nombre: String
    var edad = 0
}
// struct con funciones
```

```
struct Avion {

  var pasajeros = 0
  var velocidad = 100
  var nombre: String
  var piloto: Persona
  // metodo dentro de struct
  mutating func frenar(){
    print("Frenando")
    //al ser mutable sí afecta al struct
    self.velocidad = 0
  }
  // funcion inmutable
  func mostrarVelocidad(){
    print("Velocidad",self.velocidad)
  }

}

//  creación de una instancia
var avioneta = Avion( nombre: "78496", piloto: Persona(Nombre: "Joel",
edad: 43 )  )
// acceso a un atributo
print(avioneta.pasajeros)
// modificion de un atributo
avioneta.pasajeros = 5

print("Velocidad", avioneta.pasajeros)
// llamada de la funcion
avioneta.mostrarVelocidad()
// copia de structs por valor
var avioneta2 = avioneta
avioneta2.pasajeros = 0
//imprime: avioneta.pasajeros: 5
print("avioneta.pasajeros:",avioneta.pasajeros)
//avioneta2.pasajeros 0
print("avioneta2.pasajeros:",avioneta2.pasajeros)

print("avioneta.piloto.Nombre:",avioneta2.piloto.Nombre )

struct Fruta{
    let nombre: String = "pera"
    var precio: Int
    }

// solo se puede definir precio en el constructor
```

```
// si se llega a definir nombre será un error
var pera = Fruta(precio: 10)

struct Verdura{
    let nombre: String
    var precio: Int
}
// nombre se puede definir
//al no tener valor por defecto
var brocoli = Verdura(nombre:"brocoli", precio: 5)

struct Person {
    var name: String
    var age: Int
}

var personas = [Persona]()

//  se agregan valores al arra
personas.append(Persona(Nombre: "Celeste", edad: 23))
personas.append(Persona(Nombre: "Roel", edad: 32))
personas.append(Persona(Nombre: "Flor", edad: 17))

// copia por valor
var persona1 = personas[0]
persona1.Nombre = "Nancy"

print(persona1.Nombre) //imprime Nancy
print(personas[0].Nombre) //imprime Celeste
// se modifica un array
personas[1].edad = 26

// otras formas permitidas
struct Distro {
    var Nombre: String
    var Version: String
}

var Distros = [
    Distro(Nombre: "Ubuntu", Version: "22.04"),
    Distro(Nombre: "Fedora", Version: "38"),
    Distro(Nombre: "OpenSUSE", Version: "Leap 15")
]

//
print(Distros[0].Nombre) //  Imprime Ubuntu
```

```
print(Distros[1].Version) // Imprime 13

// for con accesxo a los structs
for distro in Distros {
    print(distro.Nombre)
}
/* salida:
Ubuntu
Fedora
OpenSUSE
*/

// función que devuelve structs
func crearVerdura( precioV: Int, nombreV: String ) -> Verdura {
    return Verdura( nombre: nombreV, precio: precioV )
}

// creación de struct por llmada de función
var verdura :Verdura = crearVerdura( precioV: 10, nombreV: "Apio")


// error por referencia indirecta
//este tipo de definciones es inválida

//Coordenada depende de Ubicacion
struct Coordenada{
    var ubicacion: Ubicacion
    var valorX: Int
    var valorY: Int
}
// Ubicación depende de Coordenada
struct Ubicacion{
    var nombre: String
    var coordenada: Coordenada
}

// error por auto referencia
struct Nodo {
//no es posible que un struct tenga atributos de
//su mismo tipo
    var siguiente: Nodo
}
```

7. Funciones

En términos generales, una función es un "subprograma" que puede ser llamado por código externo (o Interno en caso de recursión) a la función. Al igual que el programa en sí mismo, una función se compone de una secuencia de declaraciones y sentencias, que conforman el cuerpo de la función.

Se pueden pasar valores a una función y la función puede devolver un valor. Para devolver un valor específico, una función debe tener una sentencia return, que especifique el valor a devolver. Además la función debe tener especificado el tipo de valor que va a retornar. En esta ocasión el T-Swift sólo soporta tipos primitivos como tipos válidos de retorno de funciones.

En el lenguaje T-Swift a diferencia de otros lenguajes no maneja atributos por copia o por referencia de forma implícita, sino que se debe indicar de forma explícita, esto aplica para cualquier tipo de parámetro para cualquier tipo de dato.

Las funciones al igual que las variables se identifican con un ID válido.

7.1. Declaración de funciones

Consideraciones:

- Las funciones y variables si pueden tener el mismo nombre
- Las funciones pueden o no retornar un valor
- En el caso de no retornar un valor no se les debe indicar el tipo de retorno.
- El valor de retorno debe de ser del **mismo** tipo del tipo de retorno de la función
- Las funciones únicamente pueden ser declaradas en el ámbito global o dentro de las defincion de structs
- 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.

SIntaxis:

```
<Declaración_Funcion> -> func Id ( <Lista_Parámetros>? ) ( -> <Tipo_Retorno>) ?
                         {    }
                          <Bloque_Sentencias>
                         }
```

Ejemplo:

```
func func1 () -> Int {
  return 1
}
```

```
func fn2() -> String  {
  return "cadena"
}

func funcion() {
  return
}

// función inválida:
// ya se ha declarado una función llamada  func previamente

func func () -> String {
   return "valor"
}

// función inválida
// nombre inválido

func if() -> Float {
   return 21.0
}

func valor() -> String {
   // error: valor de retorno incompatible con el tipo de retorno
  return 10
}

func invalida()  {
  // error no se esperan valor de regreso
  return 1000
}
```

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. El lenguaje T-Swift tiene la particularidad de soportar los parámetros ya sea por medio de **referencia** y/o **valor**, además de soportar la nomenclatura de nombre **interno** y **externo**

Consideraciones:

- Los parámetros utilizan la palabra reservada **inout** para indicar los parámetros serán por referencia o por valor.
- No pueden existir parámetros con el mismo nombre y del mismo modo.
- Pueden existir funciones sin parámetros.
- Los parámetros deben tener indicado el tipo que poseen, en caso contrario será considerado un error.

- Si el parámetro solo posee un ID entonces este será externo e interno, si se utiliza un _ en el lugar del externo, la función podrá ser llamada sin indicar el nombre externo.
- Los parámetros que son por valor, son considerados como constantes en el cuerpo de la función, por lo tanto no se le podrán asignar algún tipo de valor, solo acceder a los mismos.

SIntaxis:

```
<Lista_Parámetros> -> , (ID | _ )? ID : inout?  <Tipo> Lista_Parametros
                    | (ID | _ )? ID : inout?  <Tipo>
```

Ejemplos:

```
// función suma
// Nombre externos: num1, num2
// Nombres internos: x, y
func suma( num1 x : Int, num2 y: Int) -> Int {
    return x + y
}
//funcion resta
// Nombres externos: ninguno
// Nombres internos: x, y
func resta(_ x : Int, _ y: Int) -> Int {
    return x - y
}

//función mul
// Nombres externos: x, y
// Nombres internos: x, y
func mul(x: Int, y: Int) -> Int {
    return x * y
}

//funciones por referencia

// duplica el valor ingresado
func duplicar(_  x: inout Int){
    x += x
}


// duplica los valores de un array
func duplicarA (_  array: inout [Int] ) {
    var i = 0
```

```
    while (i < array.count ) {
        array[i] += array[i]
        i += 1
    }
}

//funciones por valor
func ejemplo( _ v: Verdura ) {
    print(v.precio)
}
// función válida
//los nombres externos e internos son diferentes
func ejemplo2(verdura v: Verdura, v verdura: Verdura  ) {
    print(v.precio)
    print(verdura.precio)
}

// función inválida
func ejemplo2e(verdura v: Verdura, verdura v: inout Verdura, ) {

}
```

7.2. Llamada a funciones

Los parámetros en la llamada a una función son los argumentos de la función. Los argumentos se pasan a las funciones por **valor** o **referencia** según sea el caso.

Las llamadas a funciones pueden ser una sentencia o una expresión.

Consideraciones:

- Si se realiza una llamada de una función sin retorno dentro de una expresión, se deberá marcar como un error.
- Se deben verificar que los parámetros sean del mismo tipo, orden y cantidad esperada que se especificaron en la declaración.
- Se debe verificar que los nombres de los parámetros están correctos con respectos a los nombres declarados de la función.
- Una llamada se puede realizar ya sea si la función fue declarada antes o después de la llamada
- Los parámetros que son por referencia, *únicamente se les puede asignar variables,* si se asigna un literal, deberá considerarse como un error. Los parámetros por referencia se deben de referenciar con el carácter **&**

SIntaxis:

```
LLamada = Id ( <Lista_Parametros> ? )

Lista_Parámetros  -> ,  ( ID :)?  &? <Expresión> Lista_Parámetros
                        | ( ID :)?  &? <Expresión>
```

Ejemplos:

```
var numero1 = 1
var numero2 = 1
//llamada con nombres externos
print(suma(num1: numero1, num2: numero2)) //imprime 2

//llamada sin nombres externos
print(resta(numero1, numero2)) //imprime 0

//llamada con nombres externos e internos idénticos
print(mul(x: numero1, y: numero2)) //imprime 1

// llamada por referencia sin nombre externo
duplicar(&numero1)
print("numero2:", numero1) //numero tendrá el valor de 2

var array = [1,2,3,4,5,6]
duplicarA(&array)

for i in array {
    print(i)
}
/*Salida:
2
3
6
8
10
12
*/

//funcion por referencia y declaracion
// posterior de la llamada
var v = Verdura(nombre:"brocoli", precio: 5)
duplicarV(verdura: &v)
print(v.precio) // imprime 10

func duplicarV(verdura V: inout Verdura) {
```

```
   V.precio += V.precio
}

ejemplo(v.precio)  //imprime 10

//imprime:
//10
//10
ejemplo2(verdura: v, v: v)

// error el array es un parámetro por copia
// por lo tanto no se puede modificar
func errorArray( _, array: [Int]){
    array[0] = 10
}
//error el struct no está por referencia
//no se puede modificar
func errorV (_, v: Verdura){
    v.precio += 10
}

//error ejemplo no necesita nombre en ningún parámetro
ejemplo(v: v)
//error ejemplo2 si necesita los nombres de los parámetros
ejemplo2(v,v)

//error no puede haber literales en las funciones por referencia
duplicar(10)
duplicarV(verdura: &Verdura(nombre:"brocoli", precio: 5))
```

7.3. Funciones Embebidas

7.3.1. print

Esta función nos permitirá imprimir en consola únicamente expresiones de **tipo primitivo** (Int, Bool, String, etc)
Consideraciones

- Puede venir cualquier cantidad de expresiones
- Si la lista contiene un valor **`nil`**, se debe imprimir nil
- Se debe de agregar un espacio (‘ ’) entre expresiones
- Se debe de imprimir un salto de línea al final de toda la salida.
- Para valores de tipo **`Float`** se deben 4 cifras o más

Ejemplo:

```
print("cadena1","cadena2")  //mostraría: cadena1 cadena2
print("cadena1")  // mostraría cadena1
print("cadena1 \n cadena2")  // mostraría cadena1
                              //           cadena2
print("valor", 10) // mostraría valor 10
print(nil) // imprime nil
print(true) // mostraría true
print(1.00001) //imprime 1.00001
```

7.3.2. Int

Esta función propia del lenguaje T-Swift permite convertir una expresión ya sea de tipo **`String`** o **`Float`** en una expresión de tipo **`Int`**. Si la cadena que recibe como parámetro no se puede convertir a un valor numérico se debe desplegar un mensaje de error y devolver un valor **`nil`**, en caso de recibir un valor de tipo **`Float`**, debe realizar un truncamiento.

Ejemplo:

```
var w = Int("10") // w obtiene el valor de 10

var x = Int("10.001") //trunca el valor y asigna el valor de 10

var x1 = Int(10.999999) //trunca el valor y asigna el valor de 10

var y = Int("Q10.00") //error no puede convertirse a Int devuelve nil
```

7.3.3. Float

Esta función permite convertir una cadena de caracteres en una variable de tipo **`Float`**. Si la cadena que recibe como parámetro no se puede convertir a un valor numérico con punto flotante se debe desplegar un mensaje de error y retorna un valor de tipo **`nil`**

```
var w = Float("10") // w obtiene el valor de 10.00

var x = Float("10.001") //x adopta el valor de 10.001

var y = Float("Q10.00") //error no puede convertirse a Float
```

7.3.4. String

Esta función es la contraparte de las dos anteriores, es decir, toma como parámetro un valor numérico y retorna una cadena de tipo **`String.`** Además sí recibe un valor **`Bool`** lo convierte en **`"true"`** o **`"false"`**. Para valores tipo **`Float`** la cantidad de números después del punto decimal queda a discreción del estudiante.

```
print( String(10) + String(3.5)) //imprime 103.5000
print( String( true )) //true

cadena = String(true) + "->" + String(3.504) //

print(cadena) // imprime true->3.50400000
```

8. Generación de código intermedio

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

En el proyecto se utilizarán las sentencias del lenguaje C para generar la correspondiente representación del código de alto nivel, en código intermedio. El código de salida ya no posee instrucciones y expresiones complejas, por lo que se debe implementar correctamente la traducción, para obtener el mismo resultado que obtenemos al escribir código en T-Swift.

No está permitido el uso de toda función o característica del lenguaje C, no descrita en este apartado. Se utilizará una herramienta de análisis para verificar que el código de tres direcciones generado tenga el formato correcto, la herramienta se puede encontrar en el siguiente enlace.

8.1. Formato del código intermedio

El formato estándar de código de tres direcciones es una secuencia de proposiciones que tiene formato general x = y op z, donde x, y, z son números ó temporales que han sido generadas por el compilador, y op representa a operadores aritméticos o relacionales. Este código deberá generarse de acuerdo al estándar de código intermedio tres direcciones visto en las clases magistrales. Debido a algunas restricciones del lenguaje C, se podrán realizar algunas instrucciones con casteo, dichas operaciones se explicarán más adelante.

8.2. Comentarios

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

Se utilizará el formato de C para comentarios:

- Los comentarios de una línea que serán delimitados al inicio con los símbolos “//” y al final con 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 en código c3d
/*
   Esto es un comentario multilínea en código c3d
*/
```

8.3. Tipos de datos

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

- *No está permitido el uso de otros tipos de datos como cadenas o boolea*nos.
- El uso de arreglos no está permitido, únicamente para las estructuras **heap** y **stack** las cuales serán detalladas más adelante.
- *Por facilidad, se recomienda trabajar todas las variables de tipo* `float`*, para no incurrir en problemas de casteo o precisión al momento de generar la correspondiente salida en c3d.*
- Se permitirá el casteo explícito de las siguientes formas: `(int) (float)` en algunos casos, esto se explicará más adelante.

8.4. Temporales

Los temporales serán creados por el compilador en el proceso de generación de código intermedio. Los temporales deberán empezar con la letra `t` seguida de un número de uno o más dígitos.

```
t[0-9]+

//ejemplos
t1000
t800
t1
t521
```

8.4.1. Asignación de temporales:

Dentro del código intermedio se realizarán asignaciones a los temporales, esto con la finalidad de poder manejar las operaciones creadas en el lenguaje de alto nivel, en cada instrucción, los temporales pueden recibir una operación aritmética, el valor de un literal , el valor de un apuntador ó el valor dentro de la estructura Stack o Heap.

Las operaciones aritméticas para asignar a los temporales tendrán el siguiente formato:

```
<Asignacion> = temp “=”  <Expr_c3d>  op  <Expr_c3d> “;”
                          | temp “=” <Val_Est> “;”
                          | temp “=” <Expr_c3d> “;”
```

```
<Expr_c3d> =   temp
                          | ( float | int ) temp     // casteo explícito
                          | ( float | int ) num
                          | num
                          | H
                          | P
                          | ( float | int ) H
                          | ( float | int ) P

<Val_Est> = heap [ <Expr_c3d >]
                          | stack[  <Expr_c3d >]
```

Las operaciones aritméticas permitidas serán:

Símbolo	Operación	Ejemplo
+	**Suma**	`t1 = (int)t2 + (int) H`
-	**Resta**	`t2 = 2.0 - (float) 100`
*	**multiplicación**	`t1000 = t800 * 5`
/	**División**	`t1 = (int) 3.0 / t3`
%	**Módulo**	`t3 = (int) t2 % (int) t3`

8.5. Etiquetas:

Las etiquetas serán identificadores únicos que indican una ubicación específica en el código fuente, permitiéndonos realizar *saltos* a lo largo en el código en la ejecución, estas se definirán durante el proceso de compilación, el formato de la etiqueta está definido de la siguiente manera.

```
L[0-9]+

//Ejemplos
L1
L3
L100
```

8.6. Saltos

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

Para tener los mismos resultados en cuanto a las instrucciones en alto nivel, se deberá manejar correctamente los saltos entre los bloques generados. Los saltos que se permiten se dividen en:

- Condicionales. Se realiza una evaluación para determinar si se realiza el salto o no.
- Incondicionales. Realiza el salto sin realizar una evaluación.

8.6.1. Saltos Incondicionales

Este tipo de salto se realiza sin necesidad de comparar alguna condición, ,contará únicamente con la palabra reservada **goto** indicando la etiqueta de destino donde continuará la ejecución del programa. el formato del salto es el siguiente:

```
goto [Etiqueta_Destino] ;

// Ejemplo de salto incondicional

goto L1;
t1 = 1; // código sin ejecutar

L1:  // la ejecución continuará desde aquí
t1 = 0 + 0;
```

8.6.2. Saltos Condicionales

Los saltos condicionales son saltos que se realizan *si y sólo si* una condición es **verdadera**, para este caso, se utilizará la instrucción if del lenguaje C para definir este tipo de salto. El formato del salto será el siguiente:

```
<Salto_C> =  if ( <Expr_c3d> operel <Expr_c3d> ) goto [Etiqueta] ;
```

Las instrucciones if tendrán como condición una expresión relacional compatible con el lenguaje C, dichas expresiones se definen en la siguiente tabla:

Símbolo	Operación	Ejemplo
<	Menor que	(int) t2 < (int) H
>	Mayor que	2.0 > (float) 100
<=	Menor o igual que	t1000 <= (int) t800
<=	Mayor o igual que	(int) 3.0 >= t3
==	Igual que	(int) t2 == (int) t3
!=	Diferente que	(float)t9 != t800

Ejemplos de saltos condicionales

```
// ejemplo de salto condicional
      if (5 >= 3) goto L11;
      goto L12;

//    if (t2 == 1) goto L1;

if (t2 >= (float) t2) goto L1;

if (t2 >= (float) t2) goto L1;

if (P >= (float) t2) goto L1; // condición con puntero

// No es un salto condicional válido

if (heap[H] ==0 ) goto L2:  // no es código c3d

if (t2 == stack[P] ) goto L2:  // no es código c3d
```

8.7. Métodos

Los métodos se definen como bloques de código identificados por medio de un ID, que únicamente se accede por medio de una llamada, la definición de los métodos es la siguiente:

```
<Definición_metodo> = void ID ( ) {  <Sentecias_c3d>
                                     return; }
```

Consideraciones:

- No está permitido el uso de parámetros en los métodos, Se debe utilizar el stack para el paso de parámetros
- Al final de cada métodos se debe incluir la instrucción **return**
- Las funciones deben ser de tipo **void** a excepción de la función main que debe ser de tipo **int**
-

8.7.1. LLamada a métodos:

Se podrán llamar a los métodos por medio de un ID perteneciente a una función existente, con la siguiente definición:

```
<LLamada_Método> = ID () ;
```

Las llamas permiten la ejecución de una función y luego retornan el control de la ejecución al lugar donde fueron realizadas, para seguir con la ejecución.

Consideraciones:

- La llamada no debe retornar ningún valor
- El nombre de la función debe ser válido para C

Ejemplo de definición de función y llamada

```
// Ejemplos de funciones
void printString() {
      goto L2;
      L1:
      return;
}

void sumar(){

      t11=t7+t9;
      stack[(int)P]=t11;
      goto L0;
      L0:
      return;
}
void restar(){
      t16=t12-t14;
      stack[(int)P]=t16;
      goto L3;
      L3:
      return;
}
 // función main
void main(){
```

```
        stack[(int)t18]=t17;
        P=P+0;
        sumar(); //llamada a función
        t18=stack[(int)P];
        P=P-0;
        stack[(int)t19]=10;
        P=P+0;
        restar(); //llamada a función
        P=P-0;
}
```

8.8. Sentencia printf

Debido a la falta de soporte de cadenas del código de tres dirección las única forma de imprimir caracteres en consola es por medio de la sentencia printf, esta recibe 2 parámetros los cuales son:

- Formato del valor a imprimir
- Expresión de c3d con un valor a imprimir

La siguiente tabla lista los parámetros permitidos para el proyecto:

Parámetro	**Acción**
%c	Imprime en forma de carácter el valor a imprimir, según el código ASCII que representa
%d	imprime valores enteros. El segundo parámetro debe ser una conversión explícita de int ó una expresión de tipo int.
%f	imprime valores con puntos decimal, puede ser una conversión explícita de tipo (float)
%e	imprime valores con puntos decimal en notación científica, puede ser una conversión explícita de tipo (float)
%g	imprime valores con puntos decimal en notación científica reducida, puede ser una conversión explícita de tipo (float).

Consideraciones:

- El estudiante es libre de utilizar los parámetros que crea conveniente para mejorar la comprensión de las salidas generadas por el programa

- Es válido que los caracteres especiales (á, é, etc.…) fuera del conjunto estándar ASCII no se impriman de forma adecuada, debido a las limitaciones del lenguaje C
- No se podrán imprimir cadenas de caracteres de forma literal.

Definición de la instrucción printf

```
<sentencia_printf> = printf ( <parámetro> , <Expr_c3d> )
```

Ejemplos de la sentencia printf:

```
printf("%d", (int)100); // Imprime 100
printf("%c", 37); // Imprime %
printf("%f", 32.2); // Imprime 32.200000
printf("%e", t3); // Imprime el valor de t3 en notación científica .
printf("%c",104); //h
printf("%c",111); //o
printf("%c",108); //l
printf("%c",97); //a
printf("%c",10); //fin linea


// sentencia inválida, no se permite imprimir posiciones de las //
estructuras del entorno de ejecución
printf("%f", heap[(int)t3]);
printf("%f", L1); // imposible imprimir etiquetas
```

8.9. Entorno de ejecución

Como ya se expuso en otras secciones de este documento, el proceso de compilación genera el código de tres direcciones y este es el único que se ejecutará por medio de su posterior compilación a lenguaje máquina, siendo indispensable para el flujo de la aplicación, en el código intermedio **no** existen cadenas, operaciones complejas, llamadas a métodos con parámetros y muchos elementos que sí existen en los lenguajes de alto nivel.

Puesto que el código intermedio busca ser una representación próxima al código máquina, por lo que todas las sentencias de las que se compone se deben basar en las estructuras que componen el entorno de ejecución.

Por lo regular se puede decir que los lenguajes de programación cuentan con dos estructuras compuestas de bytes contiguos de un tamaño definido para realizar la ejecución de sus programas en bajo nivel, la pila a la que hemos llamado `Stack` y el montículo `Heap`, en la siguiente sección se describen estas estructuras.

8.9.1. Estructuras del entorno de ejecución

Las estructuras del entorno de ejecución podemos definirlas como arreglos de bytes que son utilizados de tal forma que puedan emular las sentencias de un lenguaje de alto nivel, el compilador de T-Swift generará código de tres direcciones que emplea dichas estructuras llamadas `stack` y `heap` las cuales determinarán el flujo y la memoria de ejecución, se recomienda por facilidad y precisión utilizar arreglos de tipo **`float`**, esto con el objetivo de que se pueda tener un acceso más rápido a los datos *sin necesidad* de leer por grupos de bytes y convertir esos bytes al dato correspondiente, como normalmente se hace en otros lenguajes, por ejemplo, en Java, un int ocupa 4 bytes, un boolean ocupa 1 solo byte, un double ocupa 8 bytes, un char 2 bytes, etc.

8.9.2. Stack y stack pointer

El stack es la estructura que se encarga del manejo de los ámbitos durante las llamadas a procedimientos o funciones, permitiendo el control de variables globales, locales, el paso de parámetros de funciones y la obtención de retornos de funciones. Para llevar a cabo estas operaciones el stack maneja espacios de memoria llamados *registros de activación*, estos registros de activación son los encargados de almacenar toda la información necesaria para el manejo de ámbitos durante la ejecución.

Para el manejo de ámbitos se utilizará un apuntador llamado “Stack Pointer”, que se identifica con el nombre P, cuyo valor cambiará conforme se ejecute el programa, y su manejo debe ser cuidadoso para no corromper espacios de memoria ajenos al ámbito que esté en ejecución, su asignación se realizará de la misma manera que se realizan las asignaciones temporales.

Ejemplo del uso del stack:

```
float stack[10000];     //stack
float P;               // Stack pointer
int main() {
     P = P + 1;
     stack[(int)P] = 10;
     t1 = stack[(int)P];
     return 0;
}
```

8.9.3. Heap y heap pointer

El heap (montículo) es la estructura encargada de guardar datos de larga duración, es decir todos aquellos datos que perdurarán independientemente de las llamadas a procedimientos o funciones que se realicen, en esta se almacenan los datos referentes a cadenas, arreglos, matrices y structs.

Para el manejo del heap se utilizará un puntero al que llamaremos H, a diferencia del puntero P, que aumenta y disminuye dependiendo del ámbito en el cual se encuentre la

ejecución, el punteo H ***solo aumenta***, su función principal es brindar siempre la próxima posición libre de memoria del montículo.

Ejemplo del uso de heap:

```
float heap[10000];          // Heap
float H;                    // Heap pointer
int main() {
    H = H + 1;              // aumento
    heap[(int)H] = 10;      // asignación
    t1 = heap[(int)H];      // acceso
    return 0;
}
```

Observaciones:

- Al guardarse los caracteres pertenecientes a cadenas, se **debe** guardar en cada espacio de memoria únicamente un carácter representado por su código ASCII.
- El puntero del heap solamente crece, nunca reutiliza ni compacta espacios de memoria
- Para el manejo de las *variables globales* queda a criterio el estudiante almacenarlas al inicio del stack ó en el heap.

8.9.4. Acceso y asignación a las estructuras del entorno de ejecución

Para asignar un valor a una estructura del entorno de ejecución, es necesario colocar el identificador de la estructura (`stack` ó `heap`), la posición en forma de expresión de código de tres direcciones seguido del valor a asignar en formato de expresión de código de tres direcciones.

Para el acceso de datos pertenecientes a las estructuras del entorno de ejecución, se podrán asignar a temporales o los punteros H ó P.

```
// formato de asignación
Asig _EE = ID [ <Expr_c3d> ] "=" <Expr_c3d> ;

// formato de acceso
Acceso _EE =  Temp "=" (int | float)?  ID [ <Expr_c3d> ]
                       |  H "=" (int | float )?  ID [ <Expr_c3d> ]
                       |  P "=" (int | float )?  ID [ <Expr_c3d> ]
```

Consideraciones:

- La asignación a las estructuras se debe realizar mediante su apuntador, temporales o valores constantes, no es permitido el uso de operaciones aritméticas o lógicas para la asignación a estas estructuras.
- No se permite la asignación a una estructura mediante el acceso a otra, por ejemplo "`Stack[0] = Heap[100]`".
- Si el acceso a las estructuras se hace por medio del apuntador o temporales, se debe realizar un casteo, debido que los temporales pueden ser de tipo float.

Ejemplos

```
//Asignación
heap[(int)H] = t1;
stock[(int)H] = (float) t11;
stack[(int)t16] = (int) t15;
stack[(int)t2] = 150;
stack[10] = 250;

//Acceso
t19 = (float) stack[(int)t18];
t18 = (int) stack[(int)t18];
H = (int) stack[(int)t18];
```

8.10. Encabezado

En esta sección se definirán todas las variables y estructuras a utilizar. Solo en esta sección se permite el uso de declaraciones, no se permite las declaraciones dentro de los métodos. El encabezado debe ser generado junto con el código de tres direcciones para hacer uso de los temporales y las estructuras. La estructura del encabezado es la siguiente:

```
#include <stdio.h>

float stack[10000];     // Stack
float heap[10000];      // Heap
float P;                // Puntero Stack
float H;                // Puntero Heap
float t1, t2, t3, t4;   // Temporales
```

Consideraciones:

- No está permitido el uso de otras librerías diferentes a `<stdio.h>`, en esta librería únicamente se usará la función de imprimir *`printf`*.
- Todas las declaraciones de temporales **se deben** encontrar en el encabezado

- El tamaño que se le asigne al stack y heap queda a discreción del estudiante. Tomar en cuenta que el heap únicamente aumenta, por lo que el tamaño de este debe ser grande.

8.11. Método main

Este es el método donde iniciará la ejecución del código traducido. Este método debe ser de tipo `int` por convención y debe terminar con la sentencia `return 0`, su estructura es la siguiente:

```
int main() {
    return 0;
}
```

8.12. Comprobación de código tres direcciones

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

La herramienta que utilizarán los estudiantes para comprobar que estén generando el código de tres direcciones con la sintaxis correcta seŕa la misma utilizada durante semestres anteriores, creada por el tutor académico Manuel Miranda ubicada en el siguiente enlace.

9. Optimización:

El compilador de T-Swift tendrá la capacidad de aplicar transformaciones de optimización al código de tres direcciones como parte de su compilación para producir código más eficiente, estas optimizaciones se realizarán mediante la optimización por mirilla:

9.1. Optimización por mirilla:

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

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

La mirilla deberá iniciar con un tamaño de **25 líneas**. Y se debe de revisar lo siguiente:

- Si en una pasada se encuentra más de una optimización, entonces *el tamaño de la mirilla debe ser la misma para la siguiente pasada,*
- En caso contrario si en la pasada no existe ninguna optimización *el tamaño de la mirilla deberá aumentar otras 25 líneas más para la siguiente pasada.* Esto se deberá de realizar hasta llegar a la última pasada.

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

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

9.1.1. Eliminación de instrucciones redundantes de carga y almacenamiento.

9.1.1.1. Regla 1

Si existe una asignación de valor de la forma `a = b` y posteriormente existe una asignación de forma `b = a`, se eliminará la segunda asignación siempre que a no haya cambiado su valor. Se deberá tener la seguridad de que no exista el cambio de valor y no existan etiquetas entre las 2 asignaciones.

Ejemplo	Optimización
`T3 = T2;` `<instrucciones>` `T2 = T3;`	`T3 = T2;` `<instrucciones>`

9.1.2. Eliminación de código inalcanzable.

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

9.1.2.1. Regla 2

Esta regla sostiene que si existe un salto condicional **Lx** y una etiqueta **Lx**; todo el código que se encuentre entre ellos podrá ser eliminado siempre y cuando no exista una etiqueta en dicho código.

Ejemplo	Optimización
goto L1; <instrucciones> L1: T3 = T1 + T3;	goto L1; L1: T3 = T1 + T3

9.1.2.2. Regla 3

Si se utilizan valores constantes dentro de las condiciones y el resultado de la condición es una constante verdadera, se podrá transformar en un salto incondicional y eliminarse el salto hacia la etiqueta falsa Lf.

Ejemplo	Optimización
if(1==1) goto L1; goto L2;	goto L1;

9.1.2.3. Regla 4

Si se utilizan valores constantes dentro de las condiciones y el resultado de la condición es una constante falsa, se podrá transformar en un salto incondicional y eliminarse el salto hacia la etiqueta verdadera **Lv**.

Ejemplo	Optimización
if(4==1) goto L1; goto L2;	goto L2;

9.1.3. Optimizaciones de flujo de control.

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

9.1.3.1. Regla 5

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

Ejemplo	Optimización
goto L1; **<instrucciones>** **L1;** **goto L2;**	goto L2; <instrucciones> *L1:* goto L2;

9.1.3.2. Regla 6

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

Ejemplo	Optimización
if (a < **b)** goto L1; <**instrucciones**> *L1:* goto L2;	if (a < **b)** goto L2; <**instrucciones**> *L1:* goto L2;

9.1.4. Simplificación algebraica y reducción por fuerza.

La optimización podrá utilizar identidades algebraicas para eliminar las instrucciones ineficientes. Para las reglas 7,8,9 se eliminan las expresiones algebraicas que no afectan el valor de una variable y que se asigna a ella misma, por ejemplo, las sumas/restas con 0 y la multiplicación/división por 1.

9.1.4.1. Regla 7

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

Ejemplo	Optimización
t1= t1 + 0 t1= t1 - 0 t1= t1 * 1 t1= t1 / 1	//Se elimina la instrucción //Se elimina la instrucción //Se elimina la instrucción //Se elimina la instrucción

9.1.4.2. Regla 8

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

Ejemplo	Optimización
`t1= t2 + 0` `t1= t2 - 0` `t1= t2 * 1` `t1= t2 / 1`	`t1= t2` `t1= t2` `t1= t2` `t1= t2`

9.1.4.3. Regla 9

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

`t1= t1 * 2` `t1= t2 * 0` `t1= 0 / y`	`t1= t1 + t1` `t1= 0` `t1= 0`

10. Reportes Generales

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

10.1. 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	Ámbito	Línea	Columna
1	El Struct “Persona” no fue encontrado.	Global	5	1
2	No se puede dividir entre cero.	Global	19	6
3	El símbolo “¬” no es aceptado en el lenguaje.	Ackerman	55	2

10.2. 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 tradujo correctamente el código de entrada.

ID	Tipo símbolo	Tipo dato	Ámbito	Línea	Columna
x	Variable	Int	Global	2	5
Ackerman	Función	Float	Global	5	1
vector1	Variable	vec	Ackerman	10	5

10.3. Reporte de optimización:

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

- Número de pasada
- Regla de optimización aplicada
- Expresión original
- Expresión optimizada
- Fila

10.4. Manejo de Errores

10.4.1. Errores semánticos:

El compilador del lenguaje T-Swift deberá ser capaz de detectar todos los errores semánticos que se encuentren durante el proceso de compilación, Todos los errores se deberán de recolectar y mostrar en el reporte de errores antes mencionado.

Un error semántico es cuando la sintaxis es la correcta, pero la lógica no es la que se pretendía, por eso, la recuperación de errores semánticos será de ignorar y reportar la instrucción en donde se generó el error. En la siguiente tabla se muestra algunos errores que puede encontrar el estudiante en todo el análisis:

Validación	Ejemplo	Error
Validar que al hacer referencia a un ID en cualquier tipo de expresión, ese debe estar definido como variable, función, etc	`sumar( z, x)`	Variable x no definida
Validar que las instrucciones break continue estén dentro de un bloque de repetición	`continue;` `while (i > 0) {` `print(i);` `}`	Instrucciones de transferencia en lugares incorrectos
Validar que la instrucción return estre dentro de una función y el retorno sea correspondiente al tipo que la función	`func calc() {` `return 1;` `}`	Valor de retorno de tipo int no esperado en una función void.
Validar tipos de parámetros en las llamadas a funciones	`func sumar(_ a: Int,_ b: Int) -> Int {` `return a + b;` `}` `sumar("hola");`	La función sumar esperaba 2 parámetros de tipo int
Validar declaración existente de variables y funciones	`var a = 10;` `var a = "hola"`	La variable a ya estaba definida previamente
Validar asignaciones de diferentes tipos , en variables de vectores o arreglos	`int b = "cadena"`	La variable a esperaba una expresión de tipo int
Validar acceso a arreglo y vectores fuera de rango	`array[-1] = 0;`	El índice es incorrecto, se esperaba un valor mayor a 0
Validar errores aritméticos de constantes	`var a = 10 / 0;`	No es posible dividir un número entre 0
Validar existencia de métodos o funciones	`sumar_alt("valor", "valor2)`	La función sumar_alt no existe

10.5. Comprobación dinámica

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

10.5.1. División entre cero

Se deberá de realizar la comprobación de división entre cero siempre y cuando se realicen expresiones aritméticas con el operador de división (/) o el operador de módulo (%). Se deberá realizar la verificación en código de tres direcciones mostrando como mensaje “MathError”. Por ejemplo:

Entrada	Salida
`var a = (55+3)/(3-3);`	`T1 = 55 + 3;` `T2 = 3 - 3;` `if (T2 != 0) goto L1;` `printf("%c", 77); //M` `printf("%c", 97); //a` `printf("%c", 116); //t` `printf("%c", 104); //h` `printf("%c", 69); //E` `printf("%c", 114); //r` `printf("%c", 114); //r` `printf("%c", 111); //o` `printf("%c", 114); //r` `T3 = 0;` `// resultado incorrecto` `goto L2;` `L1:` `T3 = T1 / T2;` `// resultado correcto` `L2:`

10.5.2. Indice fuera de los límites

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

Entrada	Salida
`var vec1: [Int] = [10,20,30,40,50]`	`// índice al que desea acceder` `T2 = 10;`

```
vec[10] = 44;
```

```
// posición en heap del tamaño  del vector
H = P + 1;
// T3 = tamaño del vector
T3 = heap[H];
T3 = T3 - 1;
// comparación índice - tamaño
if (T3 > T2) goto L1;
// comparación índice mayor a 0
if (T3 < 0) goto L1;
// 3 es el límite superior del arreglo
goto L2;
L1:
printf("%c", 66); //B
printf("%c", 111); //o
printf("%c", 117); //u
printf("%c", 110); //n
printf("%c", 100); //d
printf("%c", 115); //s
printf("%c", 69); //E
printf("%c", 114); //r
printf("%c", 114); //r
printf("%c", 111); //o
printf("%c", 114); //r
// No continúa con la instrucción
goto L3;
L2:
// Continúa con la instrucción
L3:
```

10.5.3. Manejo de nil

Se debe hacer manejo del **nil** de forma apropiada, se sugiere que sea de un valor reservado grande o pequeño para su manejo.

```
var valor: Int?

if (z == 1) {
  valor = 10
}
print(valor)

//posibles salidas:
//10

//nil
```

```
// posición en stack de valor
t0 = P + 10;
//valor por defecto de nil
stack[t0] = 9999999827968.00;

t1 = P + 1; //posición de z
t2 = stack[t1]; //se accede a z
//if
if ( t2 == 1 ) goto L1;
goto L2;
```

	`` L1: t3 = P + 10; //posición de valor stack[t3] = 10; L2: //print t4 = P + 10 //posición de valor t5 = stack[t4] //se accede a valor //verificación de nil if (t5 == 9999999827968.00) goto L3; goto L4; // imprime nil si valor no tiene valor definido L3: printf("%c", 110); //n printf("%c", 105); // printf("%c", 108); // goto L5; L4: // imprime 10 si valor es 10 print("%d", (int)t5) L5: ```

Consideraciones:

- En caso se produzca un error al intentar ejecutar una instrucción, queda a criterio del estudiante detener por completo la ejecución del programa ó seguir con la siguiente instrucción, deberá especificar el comportamiento del programa durante la calificación.
- La implementación de los ejemplos mostrados en este enunciado no es obligatoria, el estudiante es libre de crear el código que crea oportuno para el manejo de errores, siempre y cuando respete los lineamientos del código de tres direcciones.

11. Apéndice A: 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

12. Ejemplos de entrada

A continuación se muestran algunos ejemplos de entradas para el lenguaje T-Swift:
https://github.com/AndresRodas/OLC2_ArchivosDeEntrada_2023/tree/main

13. 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:

13.1. Código fuente de la aplicación
13.2. Gramáticas utilizadas
13.3. Manual técnico y manual de usuario (formato Markdown ó pdf)

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

- AndresRodas
- DamC502-2
- marckomatic

14. Restricciones

14.1. El proyecto deberá realizarse como una aplicación de escritorio ó web utilizando el lenguaje Golang

14.2. Es válido el uso de cualquier herramienta y/o librería en Golang para el desarrollo de la interfaz gráfica. Se permite la creación de un Frontend con Angula, Vue.js, React, etc. Si el estudiante lo considera necesario.

14.3. Para el analizador léxico y sintáctico se debe implementar una gramática con la herramienta ANTLR4

14.4. 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.

14.5. El desarrollo y la entrega del proyecto son de manera **individual**.

14.6. El sistema operativo queda a elección del estudiante.

15. Consideraciones

15.1. 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*.

15.2. Se necesita que el estudiante al momento de la calificación tenga el entorno de desarrollo y las herramientas necesarias para realizar compilaciones, ya que se pedirán 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*.

15.3. 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.

15.4. 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.

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

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

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

15.8. 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 T-Swift.

16. Entrega del proyecto

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

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

16.3. La entrega de cada uno de los proyectos es individual

16.4. Fecha límite de entrega del proyecto: **2 de Noviembre de 2023 a las 8:00 am**