

Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Primer semestre 2023

Catedráticos: Ing. Bayron López, Ing. Edgar Saban e Ing. Luis Espino
Tutores Académicos: Damihan Morales, Andres Rodas y Marco Mazariegos



USAC
TRICENTENARIA
Universidad de San Carlos de Guatemala

mini OL v1.2

1. Competencias

1.1. Competencia General

1.2. Competencias Específicas

2. Descripción

2.1. Componentes de la aplicación

2.1.1. mini OL IDE

2.1.2. Características Básicas

2.1.3. Consola

2.2. Flujo del proyecto

2.2.1. Proceso de compilación:

2.2.2. Proceso de comprobación

2.2.3. Proceso de ejecución

2.2.4. Visualización de reportes:

2.3. Expresiones

3. Generalidades del lenguaje mini OL

3.1. Identificadores

3.2. Case Sensitive

3.3. Comentarios

3.4. Tipos estáticos

3.5. Tipos de datos primitivos

3.6. Tipos Compuestos

3.7. Secuencias de escape

4. Sintaxis del lenguaje mini OL

4.1. Bloques de Sentencias

4.2. Palabras reservadas:

4.2.1. Valor nulo (NULL)

4.2.2. Tipos de datos:

4.2.3. Palabras reservadas para funciones:

4.2.4. Palabras reservadas para el control de flujo

4.3. Signos de agrupación

4.4. Variables

4.5. Operadores Aritméticos

4.5.1. Suma

[4.5.2. Resta](#)

[4.5.3. Multiplicación](#)

[4.5.4. División](#)

[4.5.5. Módulo](#)

[4.5.6. Operador ++](#)

[4.5.7. Negación unaria](#)

[4.6. Operaciones de comparación](#)

[4.6.1. Igualdad y desigualdad](#)

[4.6.2. Relacionales](#)

[4.7. Operadores Lógicos](#)

[4.8. Sentencias de control de flujo](#)

[4.8.1. Sentencia If Else](#)

[4.8.2. Sentencia While](#)

[4.8.3. Sentencia For](#)

[4.9. Sentencias de transferencia](#)

[4.9.1. Break](#)

[4.9.2. Continue](#)

[4.9.3. Return](#)

[5. Estructuras de datos](#)

[5.1. Vectores](#)

[5.1.1. Creación de vectores](#)

[5.1.2. Función push_back\(\)](#)

[5.1.3. Función get\(int\)](#)

[5.1.4. Función remove\(int\)](#)

[5.1.5. Función size\(\)](#)

[5.1.6. Función push_front\(\)](#)

[5.1.7. Acceso convencional:](#)

[5.2. Matrices](#)

[5.2.1. Creación de matrices](#)

[6. Structs](#)

[6.1. Definición:](#)

[6.2. Creación de structs](#)

[6.3. Uso de atributos](#)

[7. Funciones](#)

[7.1. Declaración de funciones](#)

[7.1.1. Parámetros de funciones](#)

[7.2. Llamada a funciones](#)

[7.2.1. Función main\(\)](#)

[7.3. Funciones Embebidas](#)

[7.3.1. printf](#)

[7.3.2. Media](#)

[7.3.3. Mediana](#)

[7.3.4. Moda](#)

[7.3.5. atoi](#)

[7.3.6. atof](#)

[7.3.7. iota](#)

[8. Generación de código intermedio](#)

[8.1. Formato del código intermedio](#)

[8.2. Comentarios](#)

[8.3. Tipos de datos](#)

[8.4. Temporales](#)

[8.4.1. Asignación de temporales:](#)

[8.5. Etiquetas:](#)

[8.6. Saltos](#)

[8.6.1. Saltos Incondicionales](#)

[8.6.2. Saltos Condicionales](#)

[8.7. Métodos](#)

[8.7.1. Llamada a métodos:](#)

[8.8. Sentencia printf](#)

[8.9. Entorno de ejecución](#)

[8.9.1. Estructuras del entorno de ejecución](#)

[8.9.2. Stack y stack pointer](#)

[8.9.3. Heap y heap pointer](#)

[8.9.4. Acceso y asignación a las estructuras del entorno de ejecución](#)

[8.10. Encabezado](#)

[8.11. Método main](#)

[8.12. Comprobación de código tres direcciones](#)

[9. Reportes Generales](#)

[9.1. Reporte de errores](#)

[9.2. Reporte de tabla de símbolos](#)

[10. Manejo de Errores](#)

[10.1. Errores semánticos:](#)

[10.2. Comprobación dinámica](#)

[11. Apéndice A: Precedencia y asociatividad de operadores](#)

[12. Ejemplos de entrada](#)

[13. Entregables](#)

[14. Restricciones](#)

[15. Consideraciones](#)

[16. Entrega del proyecto](#)

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 a la ciencia de datos 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.

2. Descripción

mini OL[®] es un lenguaje de programación con un enfoque en el análisis estadístico pero sin perder una clara influencia por uno de los lenguajes más mediáticos, influyentes e importantes de las ciencias computacionales como lo es C. El motivo de su creación es poder brindar una sintaxis clásica al programador pero con funciones para análisis y manejo de datos. Esta vez se ha requerido la creación de un compilador que permitirá la creación de programas más rápidos y eficientes.

2.1. Componentes de la aplicación

Se requiere la implementación de un IDE (Entorno de Desarrollo Integrado, por sus siglas en inglés) que servirá para la creación de aplicaciones en lenguaje mini OL[®], este IDE a su vez será el encargado de analizar el lenguaje de entrada. La aplicación contará con los siguientes componentes:

2.1.1. mini OL[®] IDE

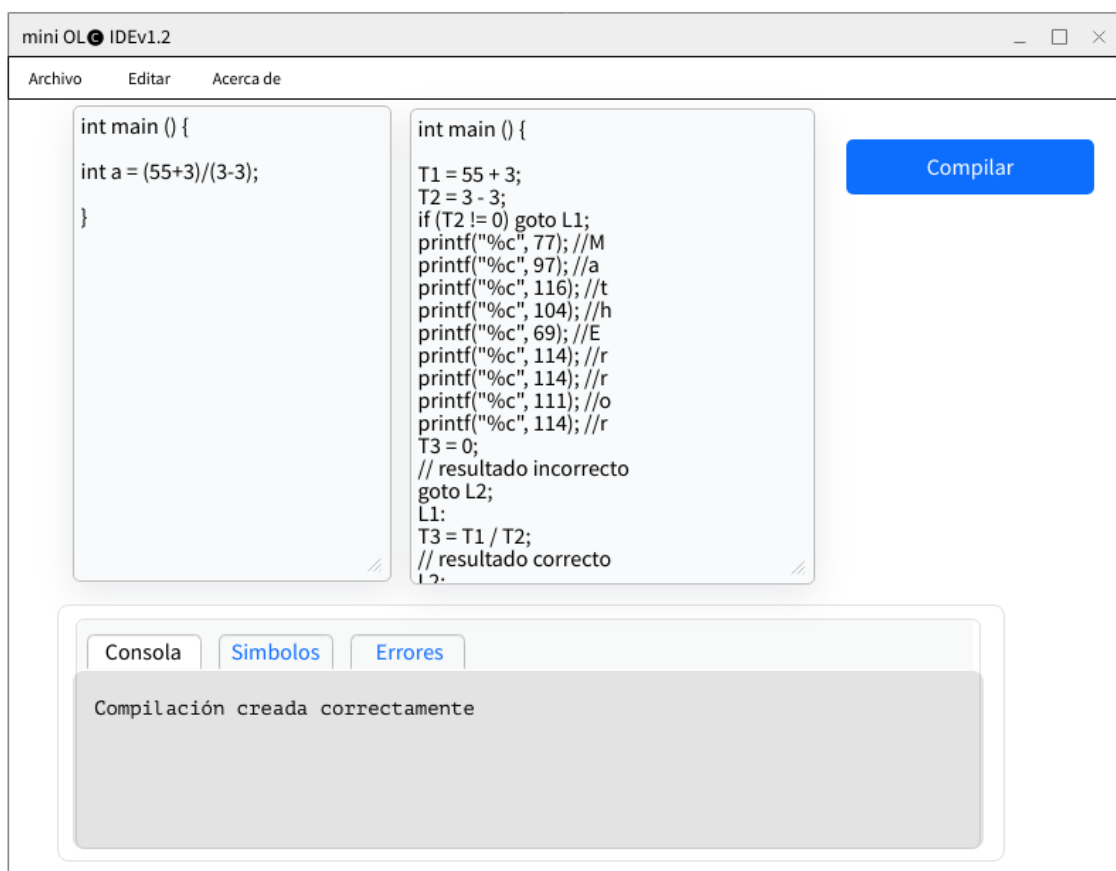
mini OL[®] IDE es un entorno de desarrollo que provee las herramientas para la escritura de programas en lenguaje mini OL[®]. Este IDE nos da la posibilidad de visualizar tanto la salida en consola de la ejecución del archivo fuente como los diversos reportes de la aplicación que se explican más adelante. La interfaz y el framework para el desarrollo de GUI queda a elección por parte del estudiante, siempre y cuando se utilice el lenguaje indicado por los tutores.

2.1.2. Características Básicas

- Abrir, guardar y guardar como
- Editor de código
- Botón para compilar archivo
- Reporte de errores en tiempo de compilación
- Reporte de tabla de símbolos
- 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.

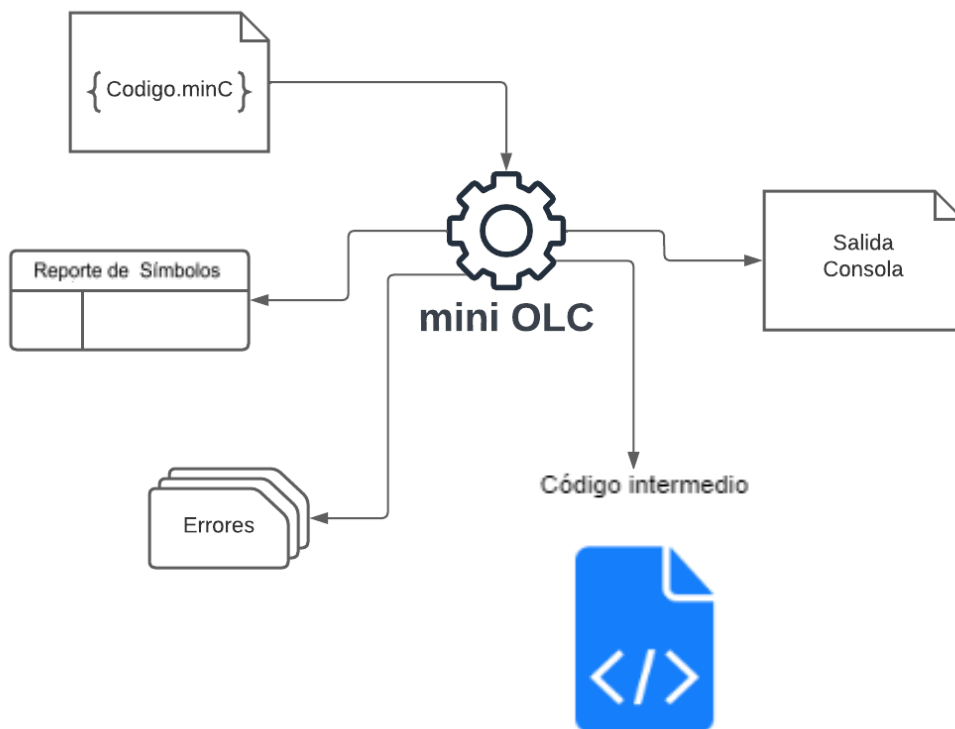


2.2. Flujo del proyecto

El flujo principal de la aplicación comienza con un archivo en lenguaje mini OL y concluye con la presentación de resultados en la consola o como una gráfica según sea el caso, como se detalla a continuación:

- El programador crea su programa en lenguaje OL a través del entorno de desarrollo.
- El programador solicita la traducción de su programa fuente.
- La aplicación analiza y ejecuta el archivo de entrada.

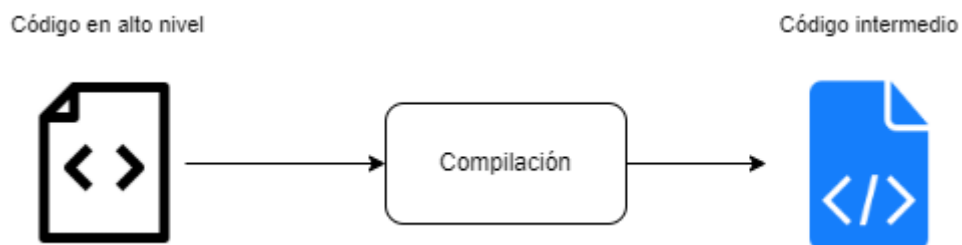
- Se despliegan en consola mensajes acerca de la compilación
- Se muestran los reportes según sea el caso (Errores, tablas, gráficas).
- Se muestra en un área de salida el resultado de la traducción.



jj

2.2.1. Proceso de compilación:

El proceso de compilación recibe una entrada de código fuente en alto nivel generada por parte del usuario y así 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 comprobación

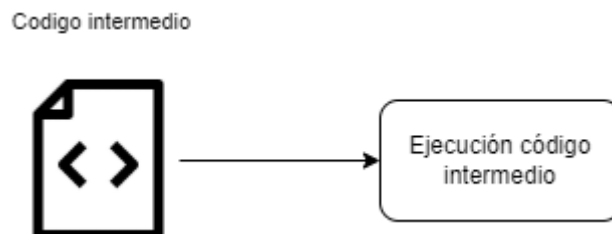
Como el código de tres direcciones utiliza sentencias del lenguaje C, se debe asegurar que el compilador genere el formato correcto antes de su ejecución, por lo cual, se debe de realizar el proceso de comprobación después de generar el código de tres direcciones. Así para cumplir con los objetivos del proyecto, por este motivo estará disponible un analizador

de código de tres direcciones desarrollado por los tutores para constatar que el código generado tenga la sintaxis correcta. Dicha herramienta se puede encontrar en el siguiente [enlace](#).



2.2.3. Proceso de ejecución

En el proceso de ejecución recibe como entrada el código de tres direcciones y será ejecutado en un [compilador en línea](#) del lenguaje de programación C, este proceso se puede realizar después del proceso de compilación, se ejecutará el código únicamente si el código de tres direcciones tiene el formato correcto.



```

1 #include <stdio.h>
2 int P;
3 int H;
4 float heap[15000];
5 float stack[15000];
6 float t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13, t14, t15;
7
8 void InsertFirst() {
9     t2 = P + 2;
10    t3 = stack[(int)t2];
11    t4 = stack[(int)t3];
12    t5 = t4 + 0;
13    t4 = heap[(int)t5];
14    t6 = -1;
15    t6 = t6 + 1.0;
16    if (t4 != t6) goto L1;
17    goto L2;
18 L1:
19    t1 = P + 1;
20    t2 = stack[(int)t1];
21    t3 = stack[(int)t2];
22    t3 = t3;
23    t7 = P + 2;
24    t8 = stack[(int)t7];
25    t9 = stack[(int)t8];
26    t10 = t9 + 0;
27    t9 = heap[(int)t10];
28    t4 = t9;
29    if ( t4 < 0 ) goto L12;
30    t6 = t3 + 2;
31    t5 = -1;
32 L13:
  
```

```

STDIN
Input for the program ( Optional )

---Insertar al inicio---
---Insertar en posicion 4---
Primero: 6, Ultimo: 5, Tamaño: 8
val: 133 indice: 0 right: 7 left: 2
val: 1 indice: 1 right: 3 left: 7
val: 3 indice: 2 right: 0 left: 4
val: 35 indice: 3 right: 5 left: 1
val: 66 indice: 4 right: 2 left: 6
val: 53 indice: 5 right: -1 left: 3
val: 78 indice: 6 right: 4 left: -1
val: 100 indice: 7 right: 1 left: 0

Imprimir desde inicio
Val|| de nodo: 78
Val|| de nodo: 66
Val|| de nodo: 3
Val|| de nodo: 133
  
```

Ejecución y compilación del código de tres direcciones en línea

2.2.4. Visualización de reportes:




Luego de haber finalizado el proceso de compilación el usuario podrá consultar el reporte de errores, advertencias y tabla de símbolos.

2.3. Expresiones

Cuando se haga referencia a una 'expresión', se hará referencia a cualquier sentencia 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. Generalidades del lenguaje mini OL

En lenguaje mini OL  está inspirado en la sintaxis del lenguaje C, por lo tanto se conforma a por un subconjunto de instrucciones de este, pero con la diferencia de que mini OL  va más allá y agrega más funcionalidades a estas para facilitar la experiencia de usuario cuando se está haciendo análisis de datos, esta vez se le solicita que se realice un compilador compatible con el lenguaje mini OL  para acelerar la ejecución de los programas realizados con el proyecto anterior.

3.1. 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 _
- El identificador no puede contener caracteres especiales (.\$,-, etc)
- El identificador no puede comenzar con un número.

3.2. Case Sensitive

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

3.3. 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
*/
```

3.4. Tipos estáticos

El lenguaje mini OL[©] no soportará múltiples asignaciones de diferentes tipos para una variable, por lo tanto si una variable es asignada con un tipo solo podrán asignarles valores de ese tipo.

3.5. Tipos de datos primitivos

Se utilizan los siguientes tipos de datos primitivos:


Tipo primitivo	Definición	Rango	Valor por defecto
int	Acepta valores numéricos enteros	-32,768 a +32767	0
float	Acepta valores numéricos de punto flotante	3.4E-38 a 3.4E+38	0.0
string	Acepta cadenas de	[0, 65535]	""

Tipo primitivo	Definición	Rango	Valor por defecto
	caracteres	caracteres	
bool	Acepta valores lógicos de verdadero y falso	true false	false

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**
- Los valores **true** y **false** cuando se combinen con tipos numéricos tomarán los valores de **1** y **0** respectivamente.
- Cuando se declare una **variable** y no se defina su valor, automáticamente tomará el valor por defecto del tipo, esto para evitar la lectura de basura durante la ejecución.

3.6. 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 mini OL .

- Vectores
- Matrices
- Structs

Estos tipos especiales se explicarán más adelante.

3.7. 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

Secuencia	Definición
\r	Retorno de carro
\r	Tabulación

4. Sintaxis del lenguaje mini OL

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


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
}

int i = 10; // variable global es accesible desde este ámbito
{
    int j = 10 + i; // i es accesible desde este ámbito
    {
        int k = ++j + i ; // i y j son accesibles desde este ámbito
    }
    j = k; // error k ya no es accesible en este ámbito
}
```

4.2. Palabras reservadas:

El lenguaje mini OL  consta de varias palabras reservadas, que se utilizan para utilizar algunas funciones o denotar algunas propiedades existentes dentro del lenguaje.

4.2.1. Valor nulo (NULL)

En el lenguaje mini OL  se utiliza la palabra reservada **NULL** para hacer referencia a la nada, esto indicará la ausencia de valor, solo podrá ser asignable a variables de tipos **primitivos**.

Tipo primitivo	valor que devuelve
int	0
float	0.0
string	"" (cadena vacía)
bool	false

Consideraciones:

- En esta ocasión para la versión v1.2 se ha limitado el uso de NULL únicamente para los siguientes casos:
 - **Definición y asignación** de variables con valor NULL para que tome el valor por defecto
 - Para la **asignación y comparación** de atributos de structs de tipo struct
- El valor NULL *no estará* presente en operaciones operaciones relacionales ni aritméticas únicamente en operaciones == y != para atributos de structs de tipo struct

4.2.2. Tipos de datos:

Las palabras reservadas para los tipos de datos son:

- **int**
- **float**
- **string**
- **bool**
- **struct**
- **vector**

Dichos tipos de datos se explicarán más adelante a lo largo de este documento.

4.2.3. Palabras reservadas para funciones:

- **return** especifica la expresión que la función devolverá como valor
- **void** especifica que una función no devolverá un tipo de dato específico.

4.2.4. Palabras reservadas para el control de flujo

El control de flujo se especificará más adelante en este enunciado, la palabra reservadas para este son las siguientes:

- **if** indica el inicio de una sentencia if
- **for** indica el inicio de una sentencia for
- **while** indica el inicio de una sentencia while
- **break** indica la terminación de la ejecución de un ciclo while
- **continue** indica el fin de una iteración en un ciclo

4.3. 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
```

4.4. 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.

sintaxis:

```
//declaración  
<tipo> <identificador> = <Expresión> ;  
<tipo> <identificador> ;  
//asignación  
<identificador> = <Expresión> ;
```

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 a una variable de tipo **int** se le asigna un valor de tipo **float**, entonces existirá un truncamiento implícito donde se descartará la parte decimal y solo se guardará la parte entera.
- Si una variable de tipo **float** se le asigna un valor de tipo **int**, entonces esta tomará el valor: <valorIntAsignado>.000
- Si a una variable de tipo **bool** se le asigna una expresión de tipo numérico, tomará el valor de **false** si se le asigna **0** o **0.0** y como **true** con cualquier otro número

Ejemplo:

```
// declaracion de variables

string valor; //correcto, declaración sin valor

int valor1 = 10; // correcto

int valor1 = 10.001; // correcto se guarda 10

float valor2 = 10.2; // correcto

float valor20 = 10; // correcto se guarda 10.00

string valor3 = "esto es una variable"; //correcto

bool valor4 = true ; //correcto

//debe ser un error ya que los tipos no son compatibles
string valor4 = true;

// debe ser un error ya que existe otra variable valor 3 definida
previamente
int valor3 = 10;

string valor10 = "10" //debe ser un error porque falta un ;

int .58 = NULL; // debe ser error porque .58 no es un nombre válido

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

// ejemplo de asignaciones

valor1 = 200; // correcto

valor3 = "otra cadena"; //correcto


valor4 = 10; //correcto, toma el valor de true

valor4 = false //incorrecto falta ;
```

4.5. 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 la potencia ^ y el módulo %.

4.5.1. Suma

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

Operandos	Tipo resultante	Ejemplo
<code>int + int</code> <code>int + float</code> <code>int + bool</code> <code>int + string</code>	<code>int</code> <code>float</code> <code>int</code> <code>string</code>	<code>1 + 1 = 2</code> <code>1 + 1.0 = 2.0</code> <code>1 + true = 2</code> <code>3 + "3" = "33"</code>
<code>float + float</code> <code>float + int</code> <code>float + bool</code> <code>float + string</code>	<code>float</code> <code>float</code> <code>float</code> <code>string</code>	<code>1.0 + 13.0 = 14.0</code> <code>1.0 + 1 = 2.0</code> <code>1.0 + false = 1.0</code> <code>1.0 + "1" = "1.01"</code>
<code>string + string</code> <code>string + int</code> <code>string + float</code> <code>string + bool</code>	<code>string</code> <code>string</code> <code>string</code> <code>string</code>	<code>"ho" + "la" = "hola"</code> <code>"1" + 1 = "11"</code> <code>"1" + 1.0 = "11.0"</code> <code>"c" + true = "ctrue"</code>
<code>bool + bool</code> <code>bool + int</code> <code>bool + float</code> <code>bool + string</code>	<code>int</code> <code>int</code> <code>float</code> <code>string</code>	<code>true + true = 2</code> <code>true + 0 = 1</code> <code>true + 1.0 = 2.0</code> <code>true + "cadena" = "truecadena"</code>

Consideraciones:

- Cualquier otra combinación será inválida y se deberá reportar el error.
- Los valores booleanos `true` y `false` se convertirán en `"true"` y `"false"` cuando sean convertidos en cadenas

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
<code>int - int</code> <code>int - float</code>	<code>int</code> <code>float</code>	<code>1 - 1 = 0</code> <code>1 - 1.0 = 0.0</code>

Operandos	Tipo resultante	Ejemplo
int - bool	int	1 - true = 0
float - float float - int float - bool	float float float	1.0 - 13.0 = -12.0 1.0 - 1 = 0.0 1.0 - false = 1.0
bool - bool bool - int bool - float	int int float	true - true = 0 true - 0 = 1 true - 1.0 = 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 * bool	int float int	1 * 10 = 10 1 * 1.0 = 1.0 1 * false = 0
float * float float * int float * bool	float float float	1.0 * 13.0 = 13.0 1.0 * 1 = 1.0 1.0 * true = 1.0
bool * bool bool * int bool * float	int int float	true * false = 0 false * 1 = 0 false * 1.0 = 0.0

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 / bool	int float int	10 / 3 = 3 1 / 3.0 = 0.3333 1 / true = 1
float / float	float	13.0 / 13.0 = 1.0

Operandos	Tipo resultante	Ejemplo
<code>float / int</code> <code>float / bool</code>	<code>float</code> <code>float</code>	<code>1.0 / 1 = 1.0</code> <code>1.0 / true = 1.0</code>
<code>bool / bool</code> <code>bool / int</code> <code>bool / float</code>	<code>int</code> <code>int</code> <code>float</code>	<code>true / true = 1</code> <code>false / 1 = 0</code> <code>false / 1.0 = 0.0</code>

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 error.

4.5.5. Módulo

El módulo produce el residuo entre la división 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
<code>int % int</code> <code>int % bool</code>	<code>int</code> <code>int</code>	<code>10 % 3 = 1</code> <code>1 % true = 0</code>
<code>bool % bool</code> <code>bool % int</code>	<code>int</code> <code>int</code>	<code>true % true = 0</code> <code>false % 1 = 0</code>

Consideraciones:

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

4.5.6. Operador ++

El operador ++ indica el incremento de `1` en una **variable** de tipo `int` y de `1.0` en una variable de tipo `float` el operador puede estar en modo **prefijo** y **postfijo** siendo estas las diferencias:

- **Prefijo:** el valor de la variable se incrementa +1 *antes* que el valor de la variable se utilice.
- **Postfijo:** el valor de la variable se incrementa +1 después que el valor de la variable se utilice.

Ejemplos:

```
int var1 = 10;
```

```
int var2 = 10;

int var3 = 1 + ++var1; //el valor de la var3 es de 12

int var4 = 1 + var2++; //el valor de la var4 es de 11

int var5 = var2; // el valor de var5 es de 11

float x = 2.0;

x++; //el valor de x se vuelve 3.0
```

Consideraciones:

- El operador ++ puede ser utilizado como una expresión o como una sentencia.
- En el caso de ser sentencia no habrá diferencia entre la notación postfija o infija, al final la variables tendrá el incremento correspondiente.

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
- bool	int	-true = -1

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 operadores pueden ser numéricos, strings o lógicos, en caso de la comparación entre tipos **bool**, **int** y **float** habrá una conversión implícita donde los valores se convertirán en tipo **float** y serán comparados.

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
<code>int [==,!=] int</code> <code>int [==,!=] float</code> <code>int [==,!=] bool</code>	<code>bool</code>	<code>1 == 1 = true</code> <code>1 != 1 = false</code> <code>1 != 0.001 = true</code> <code>1 == 1.000 = true</code> <code>1 == true = true</code> <code>0 == false = true</code>
<code>float [==,!=] float</code> <code>float [==,!=] int</code> <code>float [==,!=] bool</code>	<code>bool</code>	<code>13.0 == 13.0 = true</code> <code>0.001 != 0.001 = false</code> <code>1.0 != 1 = false</code> <code>2.0001 == 2 = false</code> <code>1.000 == true = true</code> <code>0.0 != false = false</code>
<code>bool [==,!=] bool</code> <code>bool [==,!=] int</code> <code>bool [==,!=] float</code>	<code>bool</code>	<code>true == false = false</code> <code>false != true = true</code> <code>false != 1 = true</code> <code>false == 0 = true</code> <code>true == 1.00 = true</code> <code>false != 1.00 = true</code>
<code>string [==,!=] string</code>	<code>bool</code>	<code>"ho" == "Ha" = false</code> <code>"Ho" != "Ho" = false</code>

Consideraciones

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

4.6.2. Relacionales

Las operaciones relacionales que soporta el lenguaje mini OL  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>int [>,<,>=,<=] float</code> <code>int [>,<,>=,<=] bool</code>	<code>bool</code>	<code>1 < 1 = false</code> <code>0 > 0.001 = false</code> <code>1 >= true = true</code>
<code>float [>,<,>=,<=] float</code> <code>float [>,<,>=,<=] int</code> <code>float [>,<,>=,<=] bool</code>	<code>bool</code>	<code>13.0 >= 13.0 = true</code> <code>0.999 <= 1 = true</code> <code>1.000 <= false = false</code>
<code>bool [>,<,>=,<=] bool</code> <code>bool [>,<,>=,<=] int</code> <code>bool [>,<,>=,<=] float</code>	<code>bool</code>	<code>true <= false = false</code> <code>false < -1 = false</code> <code>false > 0.00 = false</code>

Consideraciones

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

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
<code>true</code>	<code>true</code>	<code>true</code>	<code>true</code>	<code>false</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>true</code>
<code>false</code>	<code>false</code>	<code>false</code>	<code>false</code>	<code>true</code>

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 ser un valor tipo **bool** en caso contrario debe tomarse como error y reportarlo.

4.8.2. 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  
}
```

Consideraciones:

- A diferencia del lenguaje C, el ciclo while recibirá una expresión de tipo **bool**, en caso contrario deberá mostrar un error.

4.8.3. Sentencia For

Un bucle **for** en el lenguaje mini OL  itera en base a 2 *sentencias* y una expresión de *comparación*, el flujo es el siguiente:

- Antes de la primera iteración se ejecuta la *sentencia 1*
- Al inicio de cada iteración se evalúa la *expresión de comparación* donde finaliza el ciclo si esta devuelve un valor falso (**false**), en caso contrario se procede con iniciar la iteración.
- Luego ya finalizada la iteración se ejecuta la *sentencia 2* y se procede nuevamente con la evaluación de la expresión de comparación.

Sintaxis:

```
for ( <SENTENCIA 1> ; <EXPRESIÓN COMPARACIÓN>; <SENTENCIA 2> ) {  
    <BLOQUE SENTENCIAS>  
}
```

Ejemplo

```
int k = 0;  
for ( int i = 0; i < 3; i++ ) {  
    k = i + k;  
}  
  
// k al final del ciclo adoptará el valor de 3  
  
//ciclo inválido, no se permiten una declaración en la sentencia 2  
for ( int i = 0; i <= 3; int j = 0 ) {  
    k = i + k;  
}  
  
// la variable i solo corresponde al ámbito dentro del ciclo
```

```
for ( int i = 0; i <=3; i++) {  
    k = i + k;  
}  
k = i + k; //error  
  
//error son necesarios los 3 elementos que conforman el ciclo  
for (int i = 0; ; i++) {  
  
}
```

Consideraciones:

- Antes de comenzar la primera iteración se verifica la *expresión de comparación*
- Las 2 sentencias y la expresión de comparación son de carácter **obligatorio**, si falta alguna se debe reportar como un error.
- Si la expresión de comparación devuelve otro valor **diferente** a uno de tipo **bool** se debe considerar como un error.
- Se podrán declarar variables únicamente en la sentencia 1 y estas estarán asignadas al ámbito local del ciclo, si se declara una variable en la sentencia 2 se considerará como 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, sentencia switch y transfiere el control del programa a la siguiente sentencia a la sentencia de terminación de estos elementos.

Ejemplo:

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

Consideraciones:

- Si se encuentra un **break** fuera de un ciclo 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;
}

int i = 0;

for (int j = i; i < 2; i++) {
    if ( j == 0){
        i = 1;
        continue;
    }
}

// i posee el valor de 2 al finalizar el ciclo
```

Consideraciones:

- Cuando **continue** se encuentra dentro de un ciclo **for** esta ejecuta la *sentencia 2* y luego procede con la evaluación de la expresión de comparación para la ejecución de una nueva iteración.
- 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.

```
int funcion1() {
    return 1; // retorna un valor int
}

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


5. Estructuras de datos

Las estructuras de datos en el lenguaje mini OL[®] son herramientas que nos permiten almacenar distintos valores, las estructuras básicas que incluye el lenguaje son las siguientes: vectores, matrices y arreglos.

5.1. Vectores

Los vectores son la estructura compuesta más básica del lenguaje mini OL[®], los tipos de vectores que existen son con base a los tipos **primitivos** del lenguaje. Su notación de posiciones por convención comienza con 0. Una de las particularidades de los vectores es que su tamaño puede *aumentar* durante la ejecución.

5.1.1. Creación de vectores

Para crear vectores se utiliza la siguiente sintaxis.

Sintaxis:

```
<Declaracion_vector> ->  
    vector< <TIPO> > Id ( = [ <LISTA EXPRESIONES> ] )? ;
```

Consideraciones:

- Un vector al declararse sin elementos se convertirá en un vector vacío
- La lista expresiones deben ser del mismo tipo que el tipo del vector

5.1.2. Función push_back()

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

5.1.3. Función get(int)

Esta función se encarga de retornar un valor en cierta posición del vector, si esta posición no existiese o supera el tamaño actual del vector entonces devolverá un valor por defecto dependiendo del tipo de dato que sea el vector

5.1.4. Función remove(int)

Esta función remueve un elemento del vector en cierta posición, si la posición es inválida se debe de notificar el error, no retorna nada.

5.1.5. Función size()

Esta función devuelve un valor tipo **int** representando el tamaño actual del vector.

5.1.6. Función push_front()

Esta función añade un valor a principio del vector desplazando a los demás elementos.

5.1.7. Acceso convencional:

Los vectores también soportan la notación [] para la asignación acceso de valores, únicamente con los valores existentes

Ejemplo:

```
//vector con valores
vector<int> vec1 = [10,20,30,40,50];
//vector vacío
vector<float> vec2; //[ ]
//vector vacío
vector<string> vec3 = [ ];

//imprime 5
printf(iota(vec1.size( )))

//imprime 0
printf(iota(vec2.size( )))

//inserta 1 al principio
vec1.push_front(1); // [1,10,20,30,40,50 ]

//inserta 100 al final
vec2.push_back(100) //[1,10,20,30,40,50,100]

//inserciones en vacíos
vec2.push_front(1.0) // [1.0]
vec3.push_back("cadena") // {"cadena"}

//elimina la primera posición
vec1.remove(0); //[10,20,30,40,50,100]

//elimina la última posición
vec1.remove(vec1.size() - 1); //[10,20,30,40,50]

//función get
int val = vec1.get(4) // val = 50

//asignación con []
vec1[1] = vec1[0]; //[10,10,30,40,50]

//error get fuera de rango
```

```

int val2 = vec1.get(100) // val = 0 (valor por defecto de int)

//error: inserción de tipos incompatible
vec3.push_back(10)

//error: posición inválida
vec3.remove(100)

// error: asignación fuera de rango
vec1[1000] = 20;

```

5.2. Matrices

Las matrices en mini OL🟡 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 []

5.2.1. Creación de matrices

Las matrices en mini OL🟡 pueden ser de 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 sin asignación de valor, la matriz debe estar llena con los valores por defecto en la inicialización de variables con el tipo de la matriz.
- Si la declaración es explícita pero su definición no es acorde a esta declaración se debe marcar como un error.
- 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 el valor por defecto del tipo de la matriz

Sintaxis:

```

Decl_Mat -> <Tipo_Matriz> Id [num] ( [ num ] )* ( = <Defincion_ Mat> )?

Defincion_Mat -> { <Lista_Expresion> | <Definicion_Mat> ( , <Definicion_Mat>)* ) }

```

Ejemplo:

```
// declaración
int matx1 [3][2][2] ;

/*
esta matriz sería:
{
    {
        {
            0,0
        },
        {
            0,0
        }
    },
    {
        {
            0,0
        },
        {
            0,0
        }
    },
    {
        {
            0,0
        },
        {
            0,0
        }
    }
}
*/

bool mtvx2[2][5];
/*
mtvx2 = { {false, false, false, false, false},
           {false, false, false, false, false}
        }
*/
```

```
// declaración mediante definición

int matx2  [][] = int daytab[2][13] = {
{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
{0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
};

// declaración con definición
int daytab[2][13] = {
{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
{0, 32, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
};

//asignacion de valores

daytab[1][1] = 10; //cambia 10 por 32
printf(  iota( daytab[0][0] ) ) ; //imprime 0
printf(  iota( daytab [1][3] ) ); //imprime 31

//error índices fuera de rango
daytab[100][100] = 10; //devuelve 0 valor por defecto de int
```

6. Structs

El lenguaje mini OL[®] 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 incluso por sí mismos por medio de referencia.

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:

- Si el atributo de un struct es de tipo struct, este struct debe estar declarado previamente excepto si es el mismo tipo que se está definiendo .
- Si un struct se declara como vacío y tiene atributos de tipo struct estos pasarán a tener el valor **NULL** hasta que se les asigne la referencia a otro struct, los demás atributos de tipo primitivo tendrán valores por *defecto*.

- Los struct **solo** pueden ser *declarados* en el ámbito global
- Los structs deben tener al menos 1 atributo.

Sintaxis:

```
<Def_Struct> -> struct Id { <Lista_Atributos>+ }
<Lista_Atributos> -> <Tipo_Atrib> <Lista_ID> ;
```

Ejemplos:

```
struct Proveedor {
    string nombre, direccion;
    int id;
}

struct Producto {
    int id, cantidad ;
    string nombre ;
    bool disponible;
    Proveedor proveedor;
    float peso;
}

struct Nodo{
    string valor;
    Nodo anterior;
    Nodo siguiente;
}
```

```
struct Producto {
    int id, cantidad ;
    string nombre ;
    bool disponible;
    Proveedor proveedor; // declaración errónea, Proveedor no ha sido declarado
    float peso;
}

struct Proveedor {
    string nombre, direccion;
    int id;
}
```

6.2. Creación de structs

El lenguaje mini OL[©] ofrece diversas formas de declarar structs las cuales son las siguientes:

- Creación de structs vacíos
- Creación de structs con expresión.
 - Con expresiones en orden de declaración: Es una lista de expresiones que se asignan a los atributos del struct siguiendo el orden en el cual fueron declarados, no puede mezclarse con la notación Dupla - Expresión
 - Dupla - Expresión: Es una lista duplas donde consta de dos valores *ID* y *Expresión*, 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 se les asigna el valor por defecto.
 - Mediante llamada de función: es posible que una función regrese un struct del tipo que se está declarando con ciertos valores, dicha operación debe estar plenamente soportada.

Sintaxis

```
Struct_Expr -> struct ID ID ( = { <L_Expresiones> | <L_Dupla> | <Fn> } ) ? ;

<L_Expresiones> -> <Expresión> ( , <Expresión> ) *

<L_Dupla > -> . ID = <Expresión> ( , ID = <Expresión> ) *

<FN> -> <LLamada_Funcion>
```

Ejemplo:

```
// si solo son expresiones deben seguir el orden
// declarado en este caso
// { id, cantidad, nombre, disponible, proveedor, peso }
struct Producto pr = { 1, 10, "Lapicero", true,
                      { .nombre = "ISA", .id = 10 },
                      10.03 } ;

struct Producto pr1 ; // se crea un struct vacío
/* { id = 0, cantidad = 0, nombre = "",
    disponible = false, proveedor = NULL, peso = 0.0 } */

//La función nuevo_proveedor() devuelve un struct
// de tipo proveedor
```

```
struct Proveedor prov = nuevo_proveedor();

// asignación alternativa
struct Producto pr1 = { .proveedor = prov, .id = 10 }
```

6.3. Uso de atributos

El lenguaje mini OL[©] 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:

```
// acceso
printf( prov.nombre ) // imprime el contenido del atributo nombre
nodo.siguiete = raiz.anterior;

printf( producto.proveedor.nombre )

//asignación
//asigna un nuevo valor al atributo nombre
prov.nombre = "La bodega S.A.";

producto.proveedor.nombre = "Bodegas EF";

nodo.siguiete.siguiete.siguiete = NULL;
```


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.

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 deben ser declaradas como tipo **void**
- 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
- 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> = <Tipo_Retorno> Id ( <Lista_Parámetros>? ) {  
    <Bloque_Sentencias>  
}
```

Ejemplo:

```
int func1 () {  
    return 1;  
}  
  
string fn2() {  
    return "cadena";  
}  
  
void func() {  
    return;  
}
```

```

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

string func () {
    return "valor";
}

// función inválida
// nombre inválido

float if() {
    return 21.0;
}

string valor() {
    // error: valor de retorno incompatible con el tipo de retorno
    return 10;
}

void 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.

Consideraciones:

- Los parámetros de tipos primitivos se pasan por **valor** mientras que los tipos compuestos se pasan por **referencia**.
- No pueden existir parámetros con el mismo nombre.
- Pueden existir funciones sin parámetros.

Sintaxis:

```
<Lista_Parámetros> = <Tipo_Parámetro> Id ( , <Tipo_Parámetro> Id )*
```

Ejemplos:

```

int suma (int num1, int num2) {
    return num1 + num2;
}

```

```

float resta(int val1, float val2) {
    return val1 - val2;
}

void duplicar(int[] mtx, int size){
    for (int i = 0; i<size; i++) {
        mtx[i] = mtx[i]*2;
    }
}

// los struct siempre son por referencia
void editar_id(Producto prod, int id) {
    prod.id = id; //el id se ha editado
}

// retorno por referencia
Producto nuevo_producto() {
    struct Producto nuevoP = { .id = 10, .nombre = "Batería AA" };
    return nuevoP;
}

```

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** cuando son tipos primitivos y por **referencia** cuando son tipos compuestos (Arreglos, vectores ó structs). Si la función cambia el valor de un argumento, este cambio no se refleja globalmente ni en la llamada de la función si es un tipo primitivo, al cambio si es un tipo compuesto sí se verá reflejado.

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

Consideraciones:

- Si se realiza una llamada de una función de tipo **void** dentro de una expresión, esto se debe marcar como un error.
- Se deben verificar que los parámetros sean del mismo tipo y cantidad esperada que se especificaron en la declaración.
- Una llamada solo se puede realizar *si y solo si* la función ha sido declarada **antes**.

Sintaxis:

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

Ejemplos:

```

float funcion(int num1, string cadena, float num2) {
    printf( cadena );
    return num1 * num2;
}

void funcion2(vector<int> lista1, int[][] lista2){
    lista[1][1]= 100;
    lista1.push_back(5);
    return;
}

int main() {
    funcion(10, "hola", 20.001);
    // error, envio de parámetros inválidos
    funcion(10, 20);

    vector<int> vec1 = [10,20,30,40,50];
    int mat[][] = { { 0, 0}, { 0, 0} };
    //Los parámetros al ser de tipo vector y matriz
    // se envían por referencia
    // por lo tanto son modificados
    funcion2(vec1, mat);
}

```

```

int main() {
    // error debido a que la definición de funcion se encuentra abajo
    funcion(10, "hola", 20.001);
}

float funcion(int num1, string cadena, float num2) {
    printf( cadena );
    return num1 * num2;
}


```

7.2.1. Función main()

Es la función principal que posee el programa, esta debe estar declarada en el ámbito global y es de carácter **obligatorio**, esta función marcará el comienzo de la ejecución del programa, su valor de retorno puede ser **int** o **void** y no recibirá parámetros.

7.3. Funciones Embebidas

7.3.1. printf

En esta ocasión la función para la nueva versión de mini OL  v1.2 printf permitirá imprimir en consola **cualquier** expresión de *tipo primitivo*.

Consideraciones


- Puede venir cualquier cantidad de expresiones
- Si la lista contiene un valor **NULL**, se debe imprimir NULL
- Debido a la naturaleza del proyecto se considerarán algunos errores de redondeo o magnitud, debido a algunos comportamientos inesperados del lenguaje objetivo.
- Se debe considerar el manejo de errores por medio de *comprobaciones dinámicas*.

Ejemplo:

```
printf("cadena1", " cadena2") //mostraría: cadena1 cadena2
printf("valor = ", 1 ) // mostraría valor = 1
printf("cadena1 \n cadena2") // mostraría cadena1
                             //                  cadena2

printf("valor", 10+1) // mostraría: valor11
printf("valor 2: ", true) // mostraría: valor 2: true
printf("valor 3: ", 3.5) // mostraría: valor 3: 3.5
/*
Es valida la salida: 3.5000000 ó 3.5000f , etc
*/
printf("div : ", 3 / 0) // mostraría: div : MathError
printf(NULL) // imprime NULL
```

7.3.2. Media


La media es el valor que obtenemos al dividir la suma de un conjunto de valores numéricos entre la cantidad de esos mismos números. En mini OL  calculamos la media llamando a la función “mean” y mandando como parámetro un vector numérico.

Ejemplo:

```
vector<int> vec1 = [10,8,16,94,1,2,16,8,16,23];
float media1 = mean(vec1); // la función mean nos retorna 19.4
printf(media1) // se imprime 19.4
```

7.3.3. Mediana


La mediana es el valor que se encuentra justo a la mitad de un conjunto de valores numéricos, es decir, cuando se ordena el conjunto de números de menor a mayor la

mediana se localiza justamente en medio. En mini OL  calculamos la mediana llamando a la función “median” y mandando como parámetro un vector numérico.

Ejemplo:

```
vector<int> vec1 = [10,8,16,94,1,2,16,8,16,23];  
int mediana1 = median(vec1); // La función median nos retorna 13  
printf(mediana1) // se imprime 13
```

7.3.4. Moda

La moda es el valor que aparece mayor cantidad de veces en un conjunto de valores numéricos. En mini OL  calculamos la moda llamando a la función “mode” y mandando como parámetro un vector numérico.

Ejemplo:

```
vector<int> vec1 = [10,8,16,94,1,2,16,8,16,23];  
int moda1 = mode(vec1); // La función median nos retorna 16  
printf(modal) // se imprime 16
```

7.3.5. atoi

Esta función permite convertir una cadena de caracteres en una variable 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.

Ejemplo:

```
int w = atoi("10") // w obtiene el valor de 10  
  
int x = atoi("10.001") //trunca el valor y asigna el valor de 10  
  
int y = atoi("Q10.00") //error no pude convertirse a int
```

7.3.6. atof

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.

```
float w = atof("10") // w obtiene el valor de 10.00
```

```
float x = atoi("10.001") //x adopta el valor de 10.001

float y = atoi("Q10.00") //error no puede convertirse a float
```

7.3.7. iota


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 si recibe un valor bool lo convierte en **"true"** o **"false"**


```
printf( iota( 10) + iota(3.5)) //imprime 103.5
printf( iota( true )) //true

string cadena = iota(true) + " -> " + iota(3.504) //

printf(cadena); // imprime true->3.504
```

8. Generación de código intermedio

El código intermedio es una representación intermedia del programa fuente que se ingresó en mini OL . 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 mini OL .

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 $x = y \text{ op } z$, 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 booleanos.*
- 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> opere1 <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 ( ) { <Sentencias_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 la 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 convirtiéndolo en un programa funcional, siendo indispensable para el flujo de la aplicación, que 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 mini OL☉ 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 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

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 será la misma utilizada durante semestres anteriores, creada por el tutor académico Manuel Miranda ubicada en el siguiente [enlace](#).

9. Reportes Generales

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

9.1. 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

9.2. Reporte de tabla de símbolos

Este reporte mostrará *el estado final* de 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. Manejo de Errores

10.1. Errores semánticos:

El compilador del lenguaje mini OL🍷 deberá ser capaz de detectar todos los errores semánticos que se encuentren durante el proceso de compilación, Todos los errores se deberán de recolecta y mostrarse 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	<code>continue;</code> <code>while (i > 0) {</code> <code>printf(i);</code> <code>}</code>	Instrucciones de transferencia en lugares incorrectos
Validar que la instrucción return este dentro de una función y el retorno sea correspondiente al tipo que la función	<code>void calc() {</code> <code>return 1;</code> <code>}</code>	Valor de retorno de tipo int no esperado en una función void.
Validar tipos de parámetros en las llamadas a funciones	<code>int sumar(int a, int b)</code> <code>{</code> <code> return a + b;</code> <code>}</code> <code>sumar("hola");</code>	La función sumar esperaba 2 parámetros de tipo int
Validar declaración existente de variables y funciones	<code>int a = 10;</code> <code>string a = "hola"</code>	La variable a ya estaba definida previamente
Validar asignaciones de diferentes tipos , en variables de vectores o arreglos	<code>int b = "cadena"</code>	La variable a esperaba una expresión de tipo int
Validar acceso a arreglo y vectores fuera de rango	<code>array[-1] = 0;</code>	El índice es incorrecto, se esperaba un valor mayor a 0
Validar errores aritméticos de constantes	<code>int a = 10 / 0;</code>	No es posible dividir un número entre 0
Validar existencia de métodos o funciones	<code>sumar_alt("valor",</code> <code>"valor2")</code>	La función sumar_alt no existe

10.2. 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 no se tomará en cuenta este tipo de validaciones, si se produce una división por cero, ó si el acceso a un vector/matriz está fuera de sus límites ó si existe un acceso a atributos de tipo NULL el programa podrá parar su ejecución de forma abrupta debido a que no existirá este tipo de validaciones.

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
++ - (negación)	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 Mini OL .

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)


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 utilizando el lenguaje C++.
- 14.2. Es válido el uso de cualquier herramienta y/o librería en C++ para el desarrollo de la interfaz gráfica.
- 14.3. Para el analizador léxico se debe utilizar la herramienta **Flex**.
- 14.4. Para el analizador sintáctico se debe implementar una gramática utilizando la herramienta **Bison**.
- 14.5. 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.
- 14.6. El desarrollo y la entrega del proyecto son de manera **individual**.
- 14.7. 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 se reportará la copia.
- 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 entonces recibirá una penalización del 30% sobre la nota final.
- 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 mini OL .

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: **Lunes 1 de mayo a las 23:00.**