Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Primer Semestre 2018
Catedráticos: Ing. Bayron López, Ing. Edgar Sabán, Ing. Erick Navarro
Tutores académicos: José Cano, Daniel Álvarez, Mike Gutiérrez y Javier Navarro.

Draco Ensamblado Web

Segundo proyecto de laboratorio

Tabla de contenido

Índice de Tablas

Índice de Ilustraciones

1 Objetivos

1.1 Objetivo general

Aplicar los conocimientos del curso de Organización de Lenguajes y Compiladores 2 en el desarrollo de una aplicación.

1.2 Objetivos específico

- Utilizar herramientas para la generación de analizadores léxicos y sintácticos.
- Realizar análisis semántico a los diferentes lenguajes a implementar.
- Introducir al estudiante a nuevas tecnologías que utilizan los conceptos de desarrollo compiladores.
- Que el estudiante aplique los conocimientos adquiridos en la carrera para el desarrollo de un intérprete de máquina de pila.

2 Descripción

2.1 Antecedentes

JavaScript, es un lenguaje de programación del tipo interpretado, permite el desarrollo web y la implementación de funcionalidades esenciales en la lógica de los sitios web. Entre las principales características con las que cuenta JavaScript, destaca su capacidad de integrarse a la perfección con el sistema operativo y con la mayoría de los navegadores web, ofreciendo una versatilidad absoluta que muy pocos lenguajes tienen y, al ser ejecutado del lado del cliente, disminuye la carga del servidor creando aplicaciones y sitios web bastante ligeros.

La mayoría de los motores JavaScript tienen el siguiente modo de compilación:

Ilustración 1: flujo de compilación del lenguaje JavaScript

A pesar de las bondades que ofrecen lenguajes como JavaScript, la demanda de aplicaciones más eficientes resulta de carácter imperativo ya que el número de usuarios en la web y las funcionalidades requeridas se vuelven cada vez más complejas, lo que impacta directamente en la eficiencia de su ejecución, causando inconvenientes a los usuarios durante el uso de las mismas.

Actualmente existen múltiples herramientas de compilación e interpretación que permiten optimizar la ejecución de lenguajes de alto nivel, este caso no es la excepción, estas herramientas podrían dar solución a la demanda de aplicaciones óptimas que se ejecuten en menor tiempo haciendo que el usuario experimente un buen rendimiento.

Un claro ejemplo de esta optimización se puede encontrar en la máquina virtual de Java (JVM) la cual tiene definido un formato binario de código ejecutable. Análogo el famoso lenguaje de bajo nivel "Assembler", pero para JVM.

El siguiente código suma dos valores enteros:

```
public void sumar() {
    int a = 2;
    int b = 3;
    int resultado = a + b;
}
```

Ilustración 2: ejemplo de un fragmento de código Java

La máquina virtual de java lo traduce al siguiente fragmento de bytecode:

```
iconst_2
istore 1
iconst_3
istore 2

iload 1
iload 2
iadd
istore 3
return

maxstack 2
maxlocals 4
```

Ilustración 3: ejemplo de Bytecode generado por la máquina virtual de Java

El Bytecode tienden a operar en números enteros simples y números de punto flotante, evitando así la complejidad de JavaScript.

Por lo que se concluye que el hecho de incluir funcionalidades en un lenguaje intermedio podría repercutir directamente en la eficiencia de una aplicación web desarrollada en JavaScript u otro equivalente al mismo como lo es PHP o ASP.NET.

2.2 Propuesta de la solución

Se solicita a los estudiantes del curso de Organización de Lenguajes y Compiladores 2 desarrollar una solución en base a los conocimientos adquiridos, dicha solución será una aplicación de software que llevará por nombre **Draco Ensamblado Web**. La aplicación deberá tener capacidad para el desarrollo de funcionalidades en los siguientes lenguajes:

- **D++:** Lenguaje de alto nivel estructurado, compuesto por un subconjunto de sentencias derivadas del famoso lenguaje C, lo que facilitará el desarrollo de funcionalidades con grado de complejidad alto.
- **DracoScript**: Lenguaje de alto nivel, interpretado, orientado al desarrollo web, compuesto por un subconjunto de sentencias semejantes a las del popular lenguaje interpretado JavaScript.
  El objetivo del mismo será la implementación de funcionalidades con grado de complejidad bajo, además de permitir la integración de funcionalidades complejas desarrolladas utilizando el lenguaje **D++.**
- **Draco Assembler (DASM):** Lenguaje de bajo nivel, análogo al Bytecode de Java, será el código destino al compilar el código **D++.**
  DASM tendrá un set de instrucciones de "0 operando", es decir que la mayoría de las instrucciones implícitamente operan valores en el tope de la pila y reemplazan esos valores por el resultado. Esto repercute directamente en la densidad de código ya que esta será mayor pero el grado de complejidad de las instrucciones se reduce, por lo que el análisis sintáctico se simplifica, lo que brindará mejoras en el desempeño y velocidad para la ejecución de las funcionalidades complejas en el navegador.
  Es decir, la integración de las funcionalidades con grado de complejidad alto se realizará importando el archivo desde el lenguaje **DracoScript** el archivo con el código **DASM** generado.

La aplicación también permitirá al desarrollador realizar pruebas con las funcionalidades implementadas, ya que la aplicación tendrá herramientas como un editor de código intuitivo y amigable, un depurador de código, un **micro-navegador** para verificar que tanto las funcionalidades básicas como las avanzadas funcionen correctamente, etc.

Para que **Draco Ensamblado Web** sea capaz de cumplir con estas expectativas dicha aplicación deberá estar compuesta como mínimo por los siguientes componentes:

1. Un **entorno integrado de desarrollo**, IDE por sus siglas en inglés. Un IDE es una aplicación de software que permite el desarrollo de software. En este caso dicho IDE será capaz de manejar los lenguajes **DracoScript, D++, DASM.**
2. Un **compilador** que llevará por nombre **Draco Compiler**, esté será el encargado de compilar el codigo **D++** y generará un archivo con código **DASM**.
3. Un **micro-navegador** el cual permitirá verificar que las funcionalidades desarrolladas utilizando el lenguaje **DracoScript.**

2.3 Flujo general de la aplicación

A continuación, se muestra el flujo de la aplicación **Draco Ensamblado Web** de forma general.

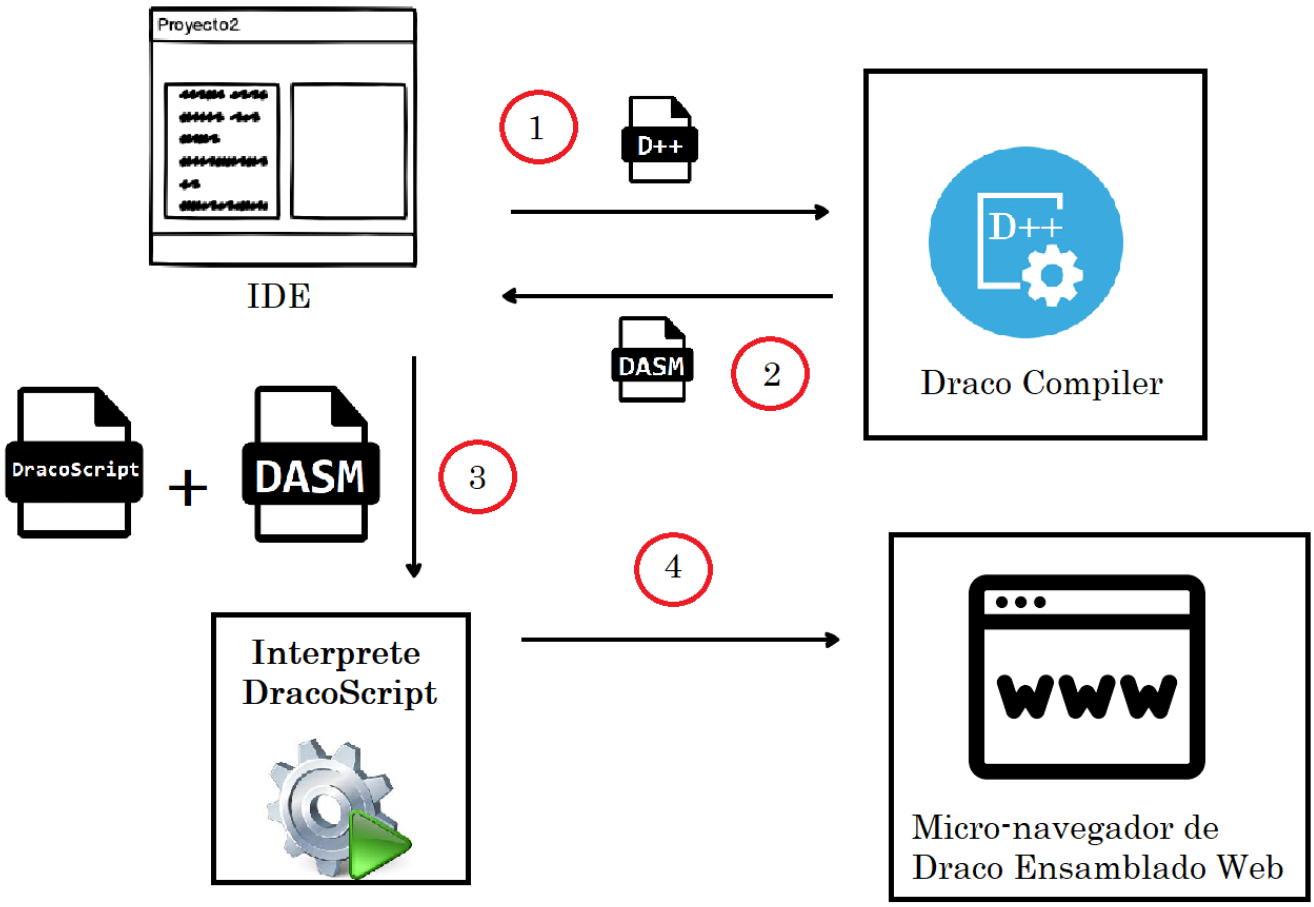


Ilustración 4: flujo general de Draco Ensamblado Web.

- Se seleccionan las funciones que representarán un grado de complejidad avanzado para desarrollarlos en el lenguaje **D++** y sea posible optimizar su ejecución.
- **Draco Compiler** genera código **DASM** en base al código **D++** de entrada.
- En el IDE de **Draco Ensamblado Web** se procederá a desarrollar la aplicación web con el lenguaje **DracoScript** y se importará el o los archivos generados por **Draco Compiler.**
  Dicho código junto con las funcionalidades contenidas en el archivo con código **DASM,** son interpretados por el intérprete de **DracoScript.**
- El intérprete de **DracoScript**, al analizar el código, permitirá definir la forma en la que se visualizará el sitio web, también permitirá la ejecución de las funcionalidades en el **Micro-navegador.**

2.4 Flujo detallado de la aplicación

2.4.1 Desarrollo de funcionalidades complejas en D++

Inicialmente se procederá a desarrollar funcionalidades de la aplicación utilizando el lenguaje **D++.**

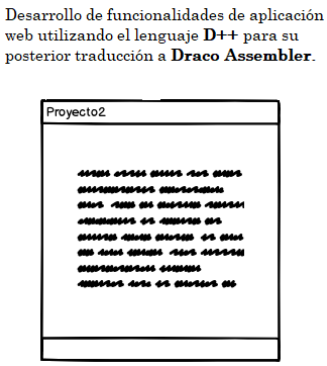


Ilustración 5: desarrollo de funcionalidades con el lenguaje D++

2.4.2 Compilar código D++

Tras haber desarrollado las funcionalidades que se desean incluir en el sitio web utilizando el lenguaje **D++** se procede a compilar dicho código y generar el código **DASM**.

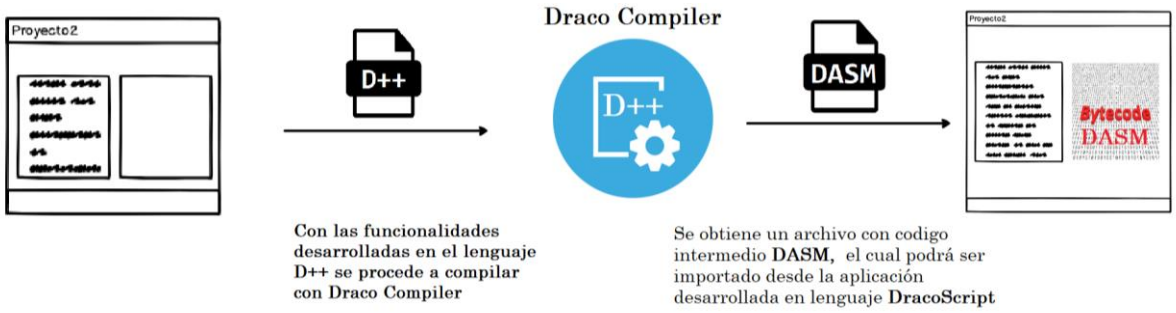


Ilustración 6: flujo de compilación de código D++

2.4.3 Importar archivo DASM generado desde DracoScript

Tras haber generado el archivo con código **DASM** y las funcionalidades que se desean integrar a la aplicación web, se procederá a importarlas con una sentencia de **DracoScript**, la cual será definida posteriormente.

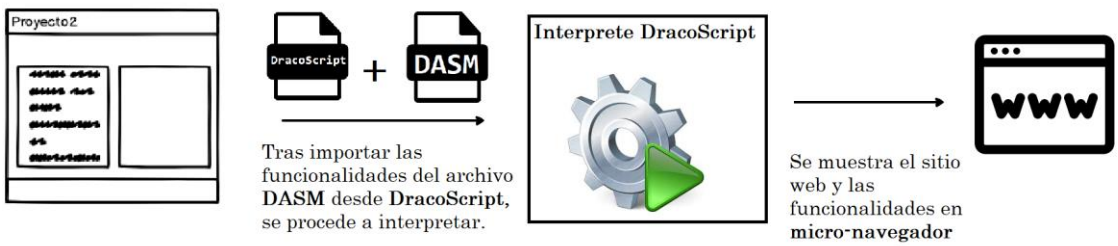


Ilustración 7: flujo de integración de código DASM con el lenguaje DracoScript

3 Componentes de la solución

3.1 Entorno Integrado de Desarrollo (IDE)

El IDE será una aplicación que proporcionará todas las herramientas necesarias para facilitarle al programador el desarrollo de software. El IDE contendrá los siguientes elementos:

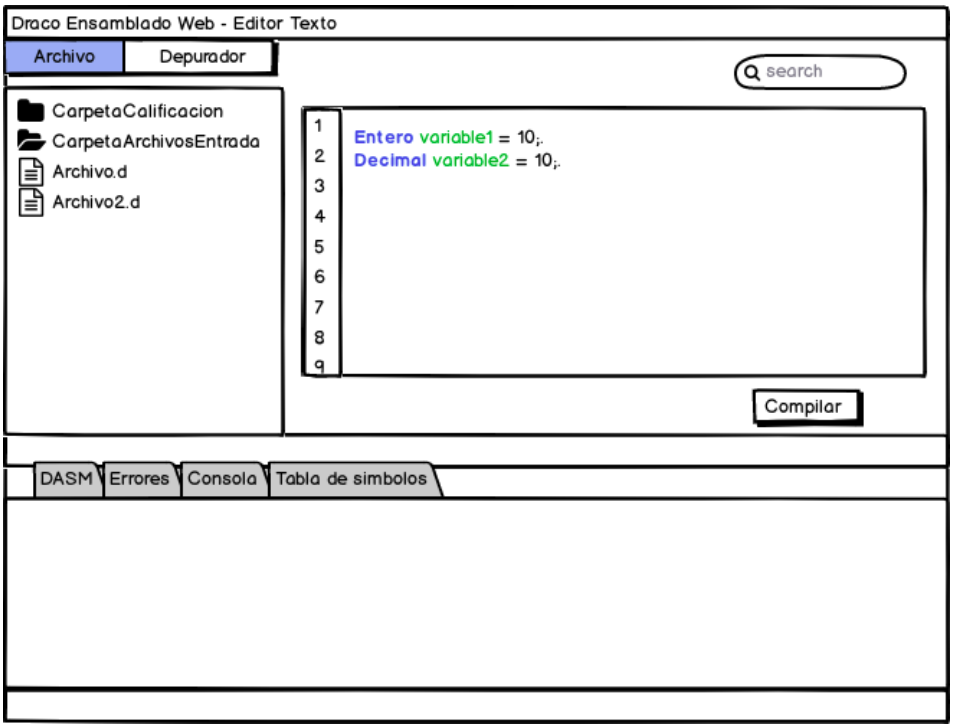


Ilustración 8: prototipo para del entorno integrado de desarrollo (IDE)

3.1.1 Editor de texto

Un editor de código fuente: El principal objetivo de este elemento será el hecho que permita el desarrollo de los lenguajes **D++** y **DracoScript**.

Funciones del editor de texto:

- Agregará sangría el contenido de texto de los archivos que se estén editando. Por ejemplo, si se añade un ciclo, el editor se encargará de colocar sangría a las instrucciones que se escriban dentro del ciclo.
- Ofrecerá un color de fuente diferente para cada componente léxico de los lenguajes **D++ y DracoScript.**
- Mostrará la línea y columna en donde se encuentre posicionado el cursor.
- Agregará el número de línea respectivo al final del documento por cada línea que se agregue al escribir el código.
- Se permitirá realizar búsquedas de palabras en todo el texto y se marcarán las coincidencias encontradas.

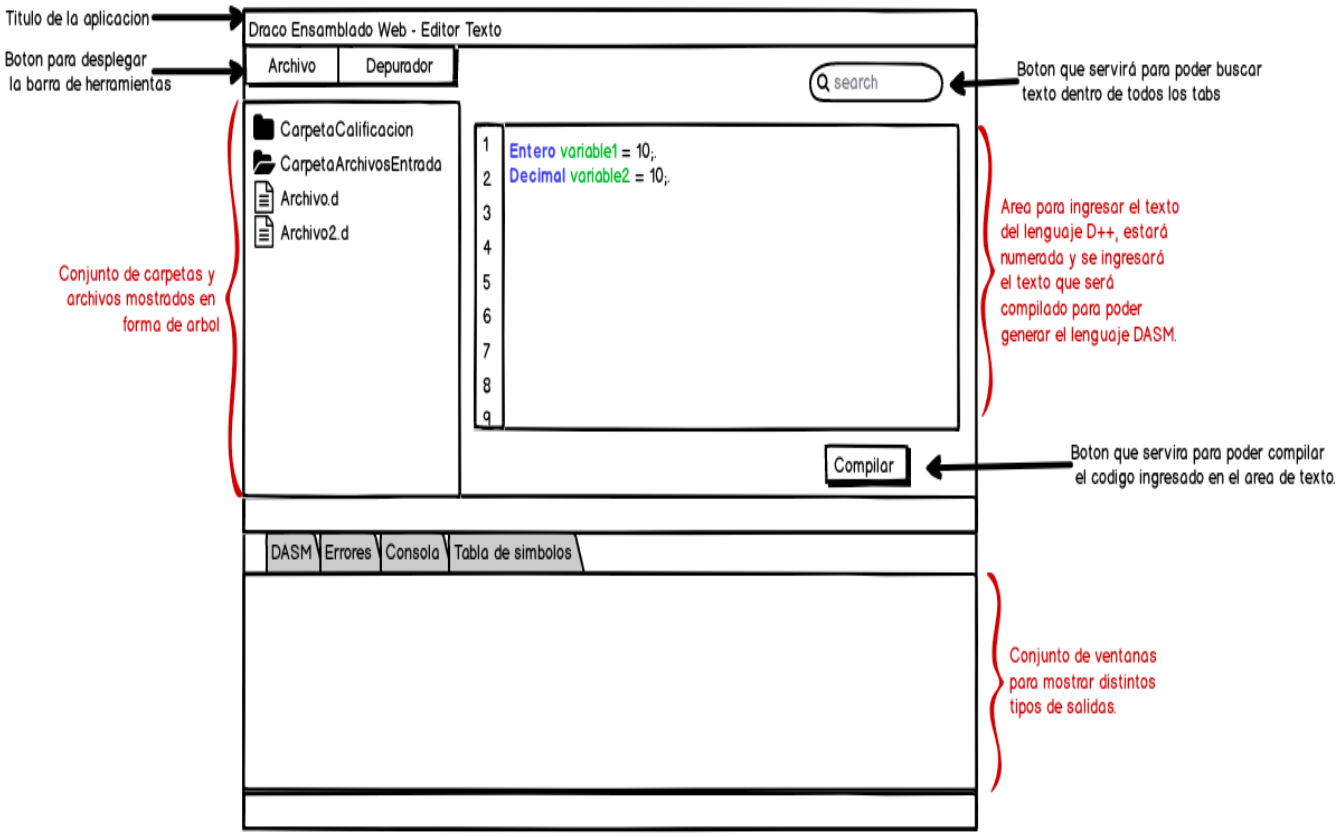


Ilustración 9: prototipo para el editor de texto

Barra de herramientas - Archivo

Al hacer clic sobre este botón mostrará el **listado de opciones** que tendrá, los cuales serán:

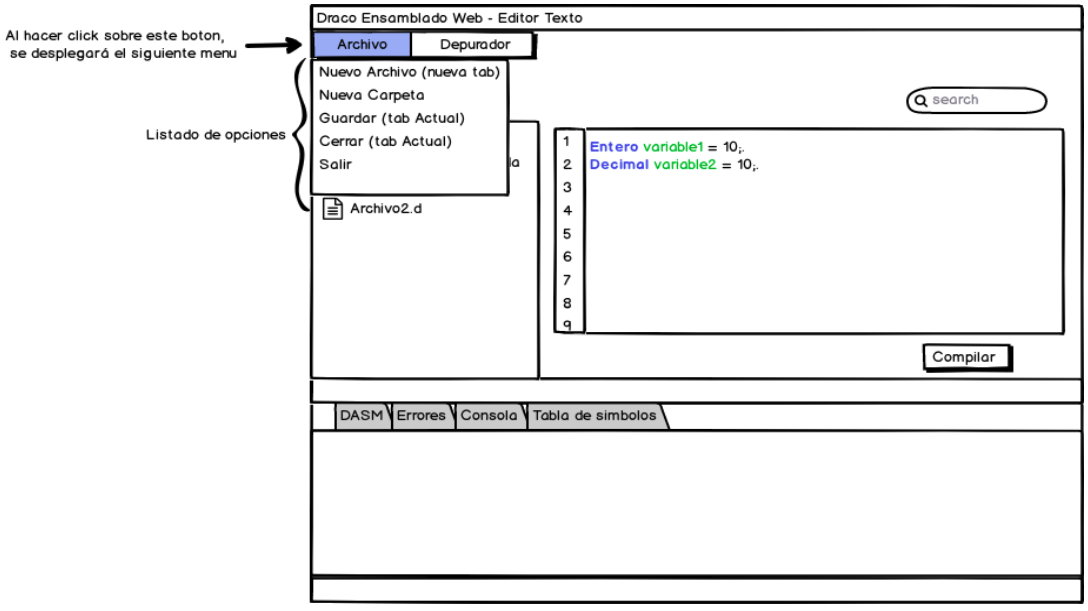


Ilustración 10: prototipo para la barra de herramientas del editor de texto

Nuevo Archivo: Creará una nueva pestaña para poder ingresar código de alto nivel y al mismo tiempo generará un archivo en la carpeta indicada, el cual se deberá mostrar en el listado de el lado izquierdo.

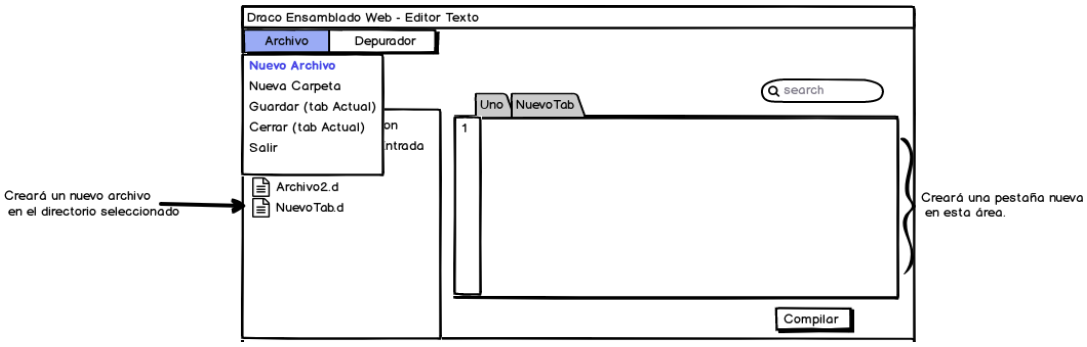


Ilustración 11: prototipo para la opción "Nuevo archivo" de editor de texto

Nueva Carpeta: Creará una nueva carpeta en la ubicación indicada y se mostrará en el listado de el lado izquierdo.

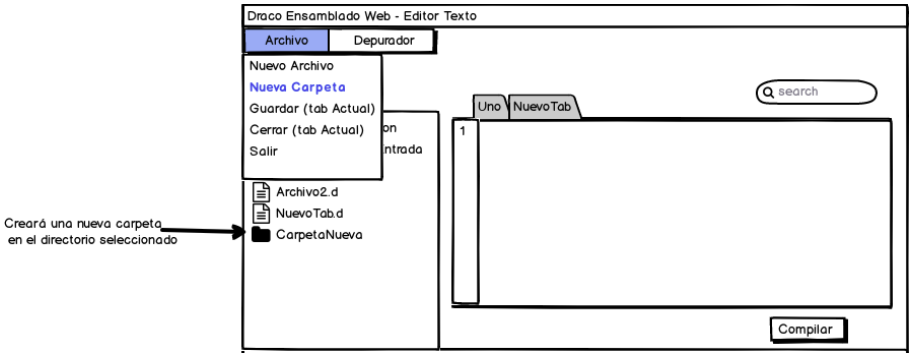


Ilustración 12: prototipo para la opción "Nuevo carpeta" de editor de texto

Guardar: Almacena la información que tiene la pestaña actual.

Cerrar: Cerrará la pestaña actual, y guardará automáticamente el contenido que tengan las pestañas las cuales no se hayan guardado después de un cambio, esto para evitar perder información.

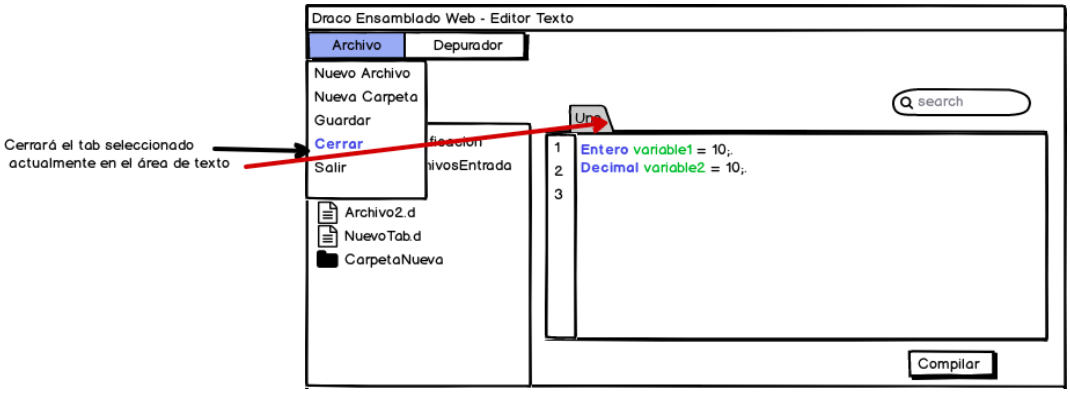


Ilustración 13: prototipo para la opción "Cerrar" de editor de texto

Salir: Cerrará la aplicación.

3.1.2 Search

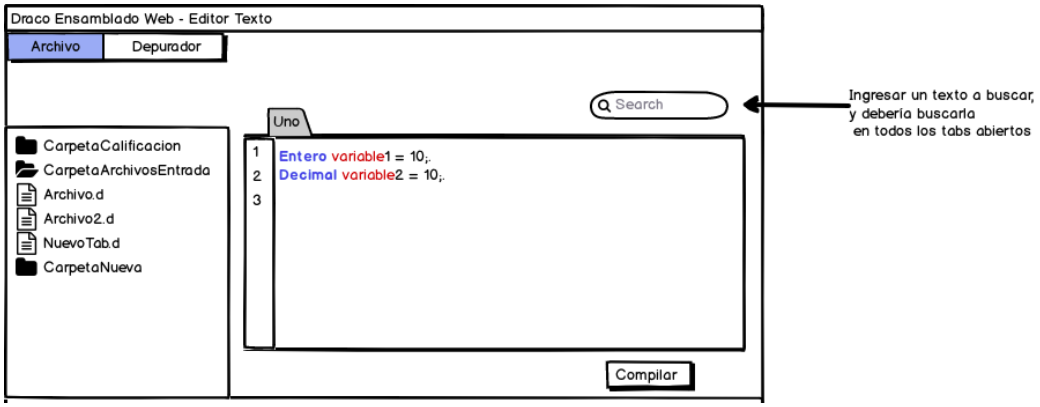


Ilustración 14: prototipo para la opción "Search" de editor de texto

3.1.3 Errores

Ventana en la cual se mostrará un listado detallado de los errores encontrados durante el análisis del código.

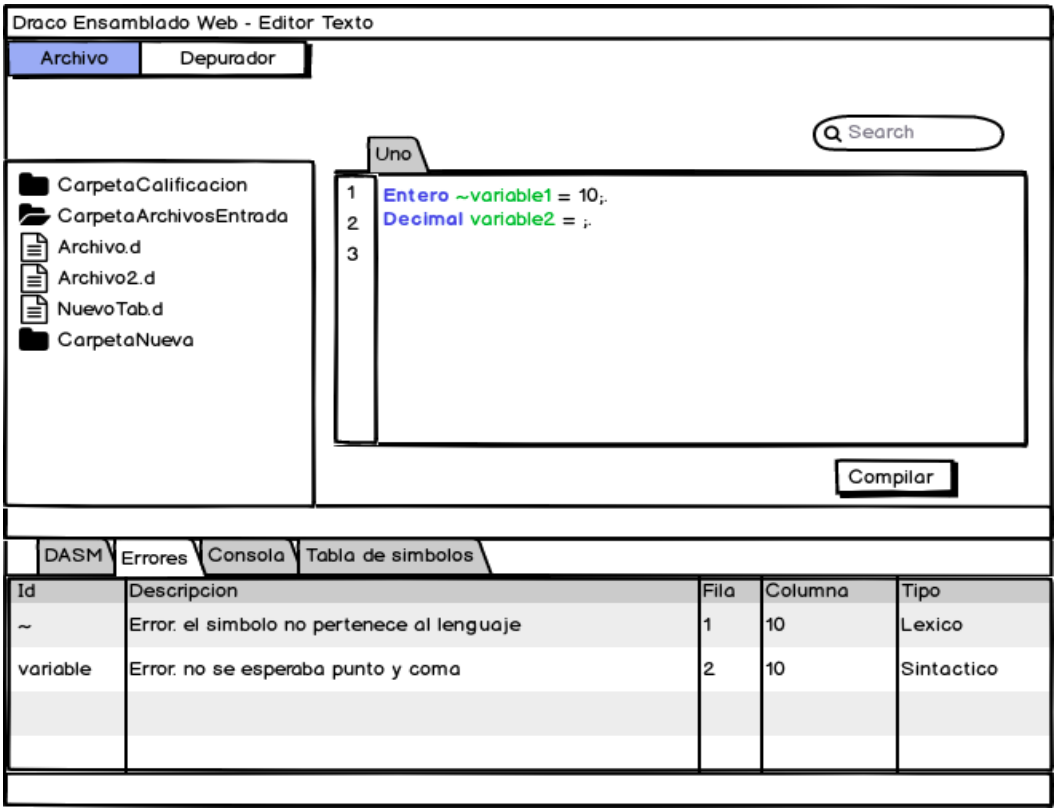


Ilustración 15: prototipo para módulo de errores de editor de texto

3.1.4 Consola

Ventana donde se mostrará el código que se haya mandado a imprimir en alto nivel, luego de ser ejecutado.

Ilustración 16: prototipo para la consola de editor de texto

3.1.5 Tabla de símbolos

Ventana donde se mostrará detalladamente todos los símbolos encontrados en el análisis del código.

Ilustración 17: prototipo para módulo de tabla de símbolos de editor de texto

3.2 Draco Compiler

Se basará en una máquina de pila, la cual es un modelo computacional en el que la memoria de la computadora toma la forma de una o más pilas.

El código deberá ser enviado al analizador, este recibirá el texto (código D++) y generar un archivo con el lenguaje DASM, si este código tiene errores entonces deberá retornar un listado de errores detalladamente, de lo contrario retornará un mensaje “éxito”.

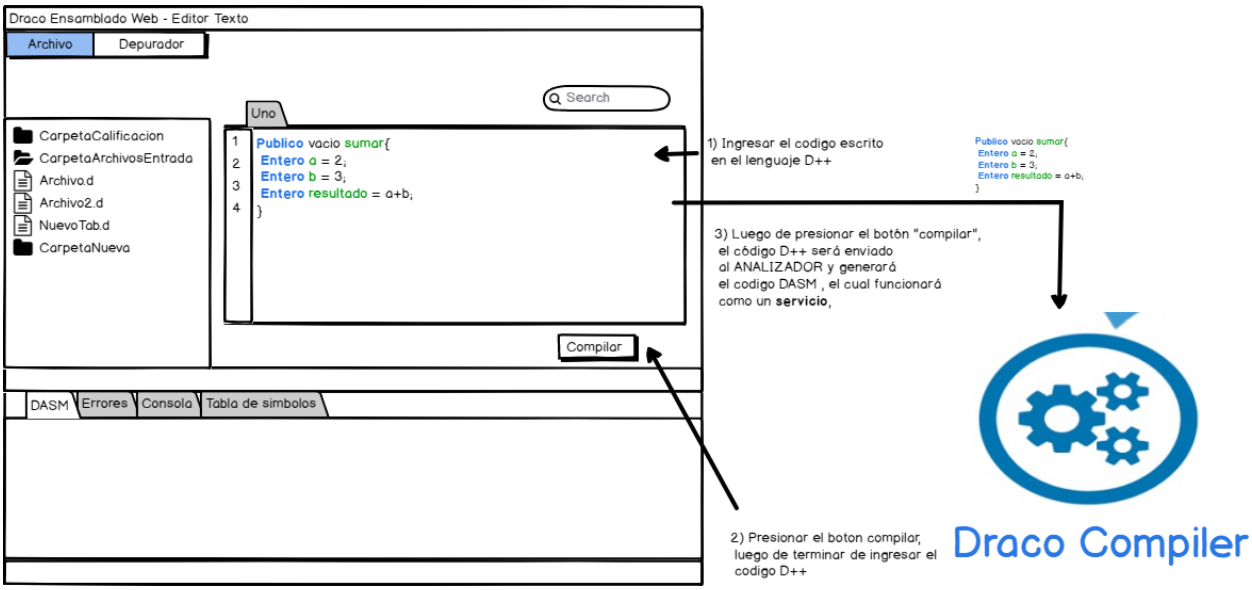


Ilustración 18: flujo de compilación para un archivo D++ desde el editor de texto

3.3 Micro-Navegador

Un **micro navegador** que será capaz de desplegar sitios web sencillos, pero con todas las funcionalidades desarrolladas tanto en el lenguaje **DracoScript** como las generadas por **Draco Compiler** en lenguaje **DASM.**

- Un intérprete que recibirá como entrada el código fuente en lenguaje **DracoScript** y será el encargado de ejecutar dicho código, así como el código **DASM** importado que generará el compilador **Draco Compiler**.
- Un entorno gráfico en el que se observaran las funcionalidades desarrolladas y sitios web sencillos.

Luego de haber generado con éxito el archivo DASM, se procederá a su respectiva ejecución y generará las imágenes que se hayan ingresado como se muestra a continuación:

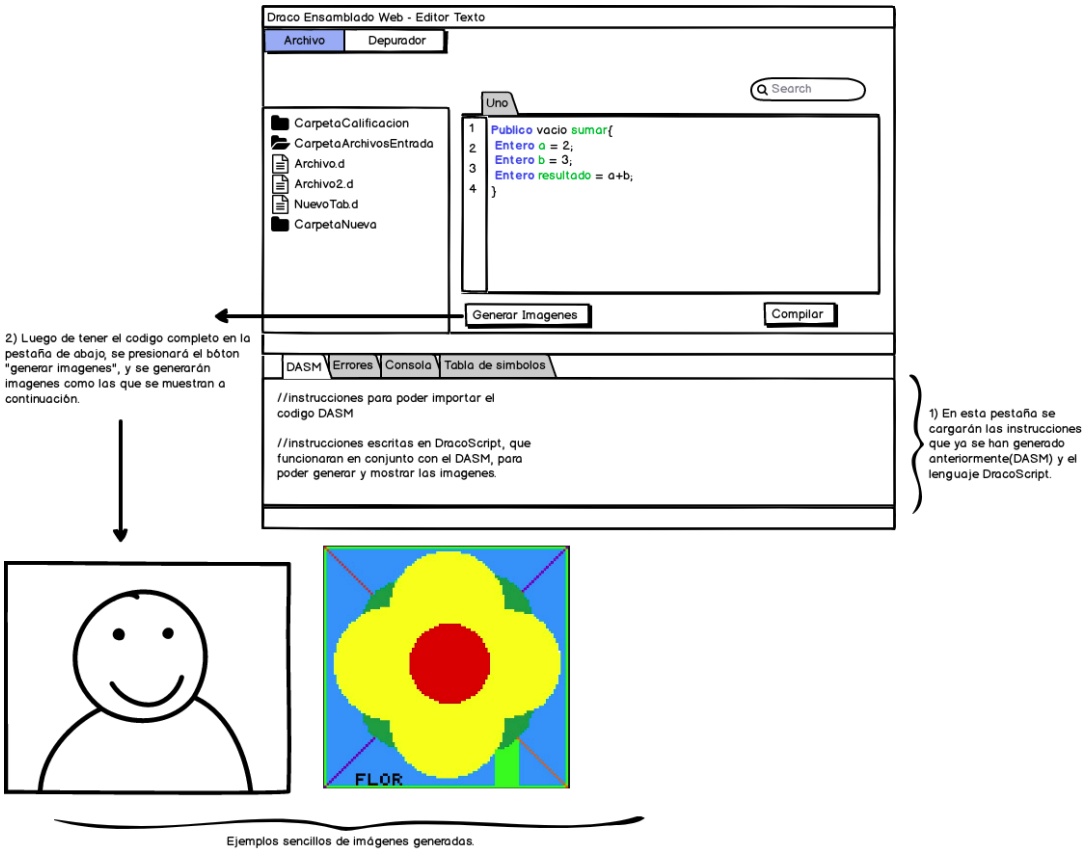


Ilustración 19: flujo de generación de dibujos.

A continuación, se mostrará otra imagen que será posible generar por medio del sistema, ingresando código de alto nivel y generando el respectivo bytecode para posteriormente mostrar estas imágenes:

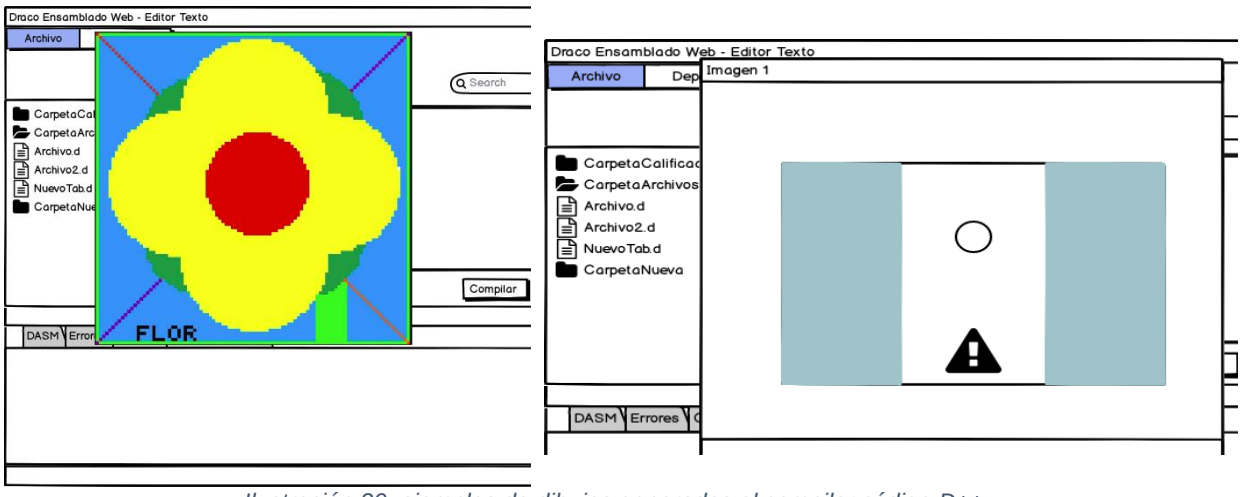


Ilustración 20: ejemplos de dibujos generadas al compilar código D++

Depurador de código

Este elemento servirá para hacer pruebas y detectar errores en los programas escritos en el editor de texto. El mismo tendrá dos modalidades:

- Depuración en la generación de código intermedio destino **DASM** a partir del código origen **D++** escrito en el editor de texto. Lo cual permitirá observar en tiempo real las estructuras necesarias, por ejemplo: La pila, el heap.
- Depuración en la interpretación del código **DracoScript.** El cual permitirá la detección de errores y pruebas en el desarrollo de funcionalidades básicas del lenguaje, así como la importación de código **DASM** que represente funcionalidades de complejidad avanzada.

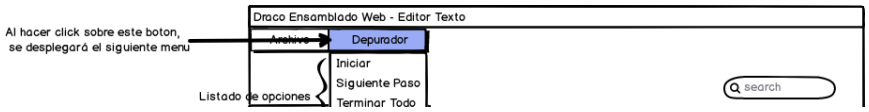


Ilustración 21: prototipo de opción "Depurador" de editor de texto.

El depurador iniciará su funcionamiento cuando el usuario presione el botón “iniciar”. A continuación, se muestran las acciones que podrá realizar.

- Permitirá correr el programa que se esté **depurando paso a paso con la opción siguiente línea**, esto significará que se ejecutará una instrucción a la vez.
- La opción **continuar**, será capaz de navegar de punto de interrupción en punto de interrupción, en el flujo del código que se esté depurando. Si no se añade ningún punto de interrupción, la ejecución continuará hasta el final de misma.
- Tendrá la opción **Iniciar** que iniciará el proceso de depuración.
- Tendrá la opción **Terminar todo** que permitirá salir del proceso de depuración.

Permitirá el manejo **puntos de ruptura** (breakpoints), un punto de ruptura facilitará el análisis de una instrucción de código especifica.

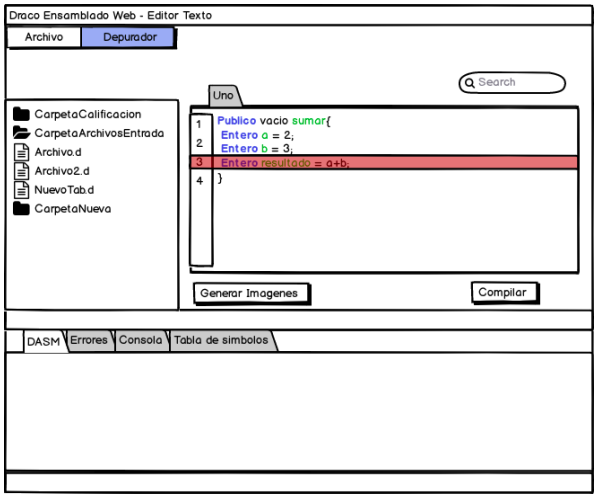


Ilustración 22: prototipo de depurador

4 Descripción del lenguaje D++

El lenguaje **D++** como cualquier lenguaje de programación, tiene su propia estructura, reglas de sintaxis y paradigma de programación, es un derivado del lenguaje C/C++ y Java, por lo que sus reglas de sintaxis son parecidas a dichos lenguajes.

4.1 Notación utilizada

Se utilizarán las siguientes notaciones cuando se haga referencia a sentencias del lenguaje **D++**.

Sintaxis y ejemplos

Para definir la sintaxis y ejemplos de sentencias de código de alto nivel se utilizará un rectángulo gris.

Asignación de colores

Dentro del enunciado, para el código de alto nivel, se seguirá el formato de colores establecido en la Tabla 1. Estos colores también deberán de ser implementados en el editor de texto del **IDE**.

*Tabla 1: colores en el código **D++** .*

COLOR	TOKEN
AZUL	Palabras reservadas
NARANJA	Cadenas, caracteres
MORADO	Números
GRIS	Comentario

Palabras reservadas

Son palabras que no podrán ser utilizadas como identificadores ya que representan una instrucción específica y necesaria dentro del lenguaje.

Expresiones

Cuando se haga referencia a una 'expresión', se hará referencia a cualquier sentencia que devuelve un valor.

Cerradura positiva

Esta será usada cuando se desee definir que un elemento del lenguaje **D++** podrá venir una o más veces: (elemento)+

Cerradura de Kleene

Esta será usada cuando se desee definir que un elemento del lenguaje **D++** podrá venir cero o más veces: (elemento)*

Opcionalidad

Esta será usada cuando se desee definir que un elemento del lenguaje **D++** podrá o no venir (cero o una vez): (elemento)?

4.2 Características generales

Esta sección describirá las características generales que definen el lenguaje de alto nivel **D++.**

4.2.1 Paradigma de programación

El paradigma de programación del lenguaje **D++** será orientado a la estructura.

Un lenguaje de programación orientado a la estructura tiene como objetivo la claridad del código, reduce la complejidad del mismo, el uso de este enfoque se divide en subprograma/subrutinas. Este paradigma de programación tiene una rica estructura de control.

4.2.2 Sensibilidad a mayúsculas

D++ será un lenguaje case sensitive, es decir, dicho lenguaje **si** será sensible a las mayúsculas y minúsculas, esto aplicará tanto para palabras reservadas propias del lenguaje como para identificadores.

4.2.3 Sobrecarga de métodos

Será posible definir dos o más métodos, que compartan el mismo nombre, mientras que las declaraciones de sus parámetros sean diferentes.

Formas de diferenciar las declaraciones de parámetros:

- Cantidad de parámetros
- Tipo de parámetros

En este caso los métodos se dice que están sobrecargados y el proceso se conoce como sobrecarga de métodos.

4.2.4 Recursividad

Los métodos y funciones de **D++** deberán soportar llamadas recursivas.

4.2.5 Identificadores

Un identificador será utilizado para dar un nombre a variables, métodos o estructuras. Un identificador es una secuencia de caracteres alfabéticos **[A-Z a-z]** incluyendo el guion bajo **[_]** o dígitos **[0-9]** que comienzan con un carácter alfabético o guion bajo.

Ejemplos de identificadores validos

```
este_es_un_identificador_valido_09
_este_tambien_09
Y_este_2018
```

Ejemplos de identificadores no válidos

```
0id
id-5
```

4.2.6 Valor nulo

Se reservará la palabra "nulo" en el lenguaje **D++** y está se utilizará para hacer referencia a la nada, indicará la ausencia de valor.

4.2.7 Variables

Las variables serán unidades básicas de almacenamiento en **D++**. Una variable se definirá por la combinación de un identificador, un tipo y un inicializador opcional. Además, la variable tiene un entorno que define su usabilidad.

4.2.8 Tipos primitivos de datos

Se utilizarán los siguientes tipos de dato:

- **Entero**: este tipo de dato, aceptará valores numéricos enteros.
- **Decimal**: este tipo de dato, aceptará números de coma flotante de doble precisión[1].
- **Cadena**: este tipo de dato aceptará cadenas de caracteres, que deberán ser encerradas dentro de comillas dobles ("caracteres").
- **Caracter**: este tipo de dato aceptará un único carácter, que deberá ser encerrado en comillas simples ('caracter').
- **Booleano**: este tipo de dato aceptará valores de verdadero y falso que serán representados con palabras reservadas que serán sus respectivos nombres ("verdadero", "falso").

*Tabla 2: rangos y requisitos de almacenamiento de cada tipo de dato de **D++***

TIPO DE DATO	RANGO	REQUISITOS DE ALMACENAMIENTO
ENTERO	[-2.147.483.648, 2.147.483.647]	4 bytes
DECIMAL	[-922.337.203.685.477,5800, 922.337.203.685.477,5800]	8 bytes
CARACTER	[0,255] (ASCII)	1 byte
BOOLEANO	verdadero, falso, 0, 1	1 byte

NOTAS

Cuando se efectúen operaciones con datos primitivos y el resultado de dicha operación sobrepase el rango establecido se deberá mostrar un error en tiempo de ejecución.

[1] El formato en coma flotante de doble precisión es un formato de número de computador u ordenador que ocupa 64 bits en su memoria y representa un amplio y dinámico rango de valores mediante el uso de la coma flotante. Este formato suele ser conocido como binary64 tal como se especifica en el estándar IEEE 754.

4.3 Sintaxis

A continuación, se detalla la sintaxis de todas las sentencias del lenguaje **D++.**

4.3.1 Bloque de sentencias

Sera un conjunto de sentencias delimitado por llaves “{}”, estas definirán un entorno local, es decir las variables declaradas en dicho entorno, únicamente podrán ser utilizadas en dicho entorno o bien en entornos hijos.

SINTAXIS

```
{ LISTA_SENTENCIAS }
```

4.3.2 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.

SINTAXIS

```
55 * ((85 - 3) / 8)
```

4.3.3 Comentarios

Los comentarios permiten añadir al código fuente notas o comentarios de texto que son ignorados por el compilador. Existirán dos tipos de comentarios:

Comentarios de una línea

Iniciarán con los símbolos “//” y finalizarán con salto de línea.

SINTAXIS

```
// SECUENCIA_CARACTERES \n
```

EJEMPLO

```
//este es un comentario de una sola línea
```

Comentarios de múltiples líneas

Iniciarán con los símbolos “/*” y finalizarán con los símbolos “*/”.

SINTAXIS

```
/* SECUENCIA_CARACTERES */
```

EJEMPLO

```
/*
 * este es un comentario
 * de múltiples líneas
  */
```

4.3.4 Operaciones aritméticas

Una operación aritmética permitirá obtener expresiones a partir de otras expresiones utilizando operadores aritméticos.

Suma

Operación aritmética que consistirá en adicionar un sumando a otro sumando. El operador de la suma es el signo más (+).

A continuación, se define el sistema de tipos para la suma.

*Tabla 3: sistema de tipos para la suma en **D++***

OPERANDOS	TIPO RESULTANTE	EJEMPLOS
entero + decimal decimal + entero decimal + caracter caracter + decimal booleano + decimal decimal + booleano decimal + decimal	decimal	5 + 4.5 = 9.5 7.8 + 3 = 10.8 15.3 + 'a' = 112.3 'b'+ 2.7 = 100.7 verdadero + 1.2 = 2.2 4.5 + falso = 4.5 3.56 + 2.3 = 5.86
entero + caracter caracter + entero booleano + entero entero + booleano entero + entero	entero	7 + 'c' = 106 'C' + 7 = 74 4 + verdadero = 5 4 + falso = 4 4 + 5 = 9
cadena + entero cadena + decimal decimal + cadena entero + cadena	cadena	"hola" + 2 = "hola2" "hola" + 3.5 = "hola3.5" 4.5 + "hola" = "4.5hola" 8 + "hola" = "8hola"
cadena + caracter caracter + cadena cadena + cadena	cadena	"hola" + 't' = "holat" 'u' + "hola" = "uhola" "hola" + "mundo" = "holamundo"
booleano + booleano	booleano	1 + falso = verdadero verdadero + verdadero = verdadero 0 + 1 = 1 = verdadero falso + 0 = falso

Cualquier otra combinación será inválida y se deberá reportar un error de semántica.

Resta

Operación aritmética que consistirá en quitar una cantidad (sustraendo) de otra (minuendo) para averiguar la diferencia entre las dos. El operador de la resta es el signo menos (-).

A continuación, se define el sistema de tipos para la operación aritmética de resta.

*Tabla 4: sistema de tipos para la resta en **D++***

OPERANDOS	TIPO RESULTANTE	EJEMPLOS
entero - decimal	decimal	5 - 4.5 = 0.5
decimal - entero		7.8 - 3 = 4.8
decimal - caracter		15.3 - ‘a’ = 81.7
caracter - decimal		‘b’- 2.7 = 95.3
booleano - decimal		verdadero - 1.2 = 0.2
decimal - booleano		4.5 - falso = 4.5
decimal - decimal		3.56 - 2.3 = 1.26
entero + caracter	entero	7 - ‘c’ = 92
caracter + entero		‘C’ – 7 = 60
booleano + entero		4 – verdadero = 3
entero + booleano		4 - falso = 4
entero + entero		4 - 5 = -1

Cualquier otra combinación será inválida y se deberá reportar un error de semántica.

Multiplicación

Operación aritmética que consistirá en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). El operador de la multiplicación es el asterisco (*).

A continuación, se define el sistema de tipos para la multiplicación.

*Tabla 5: sistema de tipos para la multiplicación en **D++***

OPERANDOS	TIPO RESULTANTE	EJEMPLOS
entero * decimal	decimal	5 * 4.5 = 22.5
decimal * entero		7.8 * 3 = 23.4
decimal * caracter		15.3 * ‘a’ = 1484.1
caracter * decimal		‘b’ * 2.7 = 264.6
booleano * decimal		verdadero * 1.2 = 1.2
decimal * booleano		4.5 * falso = 0
decimal * decimal		3.56 * 2.3 = 8.188
entero * caracter	entero	7 * ‘c’ = 693
caracter * entero		‘C’ * 7 = 469
booleano * entero		4 * verdadero = 4
entero * booleano		4 * falso = 0
entero * entero		4 * 5 = 20
booleano * booleano	booleano	verdadero * 0 = 0 = falso verdadero * 1= 1 = verdadero 0 * falso = 0 = falso 1 * falso = 0 = falso

Cualquier otra combinación será inválida y se deberá reportar un error de semántica.

División

Operación aritmética que consistirá en partir un todo en varias partes, al todo se le conoce como dividendo, al total de partes se le llama divisor y el resultado recibe el nombre de cociente. El operador de la división es la diagonal (/).

A continuación, se define el sistema de tipos para la operación aritmética de división.

*Tabla 6: sistema de tipos para la división en **D++***

OPERANDOS	TIPO RESULTANTE	EJEMPLOS
entero / decimal	decimal	5 / 4.5 = 1.11111
decimal / entero		7.8 / 3 = 2.6
decimal / caracter		15.3 / 'a' =0.1577
caracter / decimal		'b' / 2.7 = 28.8889
booleano / decimal		verdadero / 1.2 = 0.8333
decimal / booleano		4.5 / falso = error
decimal / decimal		3.56 / 2.3 = 1.5478
entero / caracter		7 / 'c' = 0.7070
caracter / entero		'C' / 7 = 9.5714
booleano / entero		4 / verdadero = 4.0
entero / booleano		4 / falso = error
entero / entero		4 / 5 = 0.8

Cualquier otra combinación será inválida y se deberá reportar un error de semántica.

Potencia

Operación aritmética que consistirá en multiplicar varias veces un mismo factor.

A continuación, se define el sistema de tipos para la operación aritmética de división.

*Tabla 7: sistema de tipos para la potencia en **D++***

OPERANDOS	TIPO RESULTANTE	EJEMPLOS
entero ^ decimal	decimal	5 ^ 4.5 = 1397.54
decimal ^ entero		7.8 ^ 3 = 474.55
decimal ^ caracter		15.3 ^ 'a' = fuera de rango
caracter ^ decimal		'b' ^ 2.7 = 237853.96
booleano ^ decimal		verdadero ^ 1.2 = 1.0
decimal ^ booleano		4.5 ^ falso = 1.0
decimal ^ decimal		3.56 ^ 2.3 = 0.0539
entero ^ caracter	entero	7 ^ 'c' = fuera de rango
caracter ^ entero		'C' ^ 7 = 6060711605323
booleano ^ entero		4 ^ verdadero = 4
entero ^ booleano		4 ^ falso = 1
entero ^ entero		4 ^ 5 = 1024

Cualquier otra combinación será inválida y se deberá reportar un error de semántica.

Aumento

Operación aritmética que consistirá en añadir una unidad a un dato numérico. El aumento será una operación de un solo operando. El aumento sólo podrá venir del lado derecho de un dato. El operador del aumento es el doble signo más (++). Especificaciones sobre el aumento:

- Al aumentar un tipo de dato numérico (entero, decimal, caracter) el resultado será numérico.
- No será posible aumentar tipos de datos `carácter o cadena.`
- No será posible aumentar tipos de datos lógicos (booleano).
- El aumento podrá realizarse sobre números o sobre identificadores de tipo numérico (variables, parámetros o atributos).

SINTAXIS

```
nombre_variable++;
```

EJEMPLO

```
contador++;
```

Decremento

Operación aritmética que consistirá en disminuir una unidad a un dato numérico. El decremento será una operación de un solo operando. El decremento sólo podrá venir del lado derecho de un dato. El operador del decremento es el doble signo menos (--). Especificaciones sobre el decremento:

- Al decrementar un tipo de dato numérico (entero, decimal, caracter) el resultado será numérico.
- No será posible decrementar tipos de datos `carácter o cadena`.
- No será posible decrementar tipos de datos lógicos (booleano).
- El decremento podrá realizarse sobre números o sobre identificadores de tipo numérico (variables, parámetros o atributos).

SINTAXIS

```
nombre_variable--;
```

EJEMPLO

```
contador--;
```

Precedencia de operadores aritméticos

Para saber el orden jerárquico de las operaciones aritméticas se define la siguiente precedencia de operadores. La precedencia de los operadores irá de menor a mayor según su aparición en la tabla 8.

*Tabla 8: precedencia y asociatividad de operadores aritméticos de **D++***

NIVEL	OPERADOR	ASOCIATIVIDAD
0	+ -	Izquierda
1	* /	Izquierda
2	^	Derecha
3	++ --	Derecha

4.3.5 Operaciones relacionales

Una operación relacional será una operación de comparación entre dos valores, siempre devolverá un valor de tipo lógico (`booleano`) según el resultado de la comparación. Una operación relacional es binaria. La jerarquía de los operadores relacionales es la misma. Las especificaciones sobre las operaciones relacionales son las siguientes:

- Será válido comparar datos numéricos (entero, decimal) entre sí, la comparación se realizará utilizando el valor numérico con signo de cada dato.
- Será válido comparar cadenas de caracteres (caracter, cadena) entre sí, la comparación se realizará sobre el resultado de sumar el código ASCII de cada uno de los caracteres que forman la cadena.
- Será válido comparar datos numéricos con datos de caracter (caracter) en este caso se hará la comparación del valor numérico con signo del primero con el valor del código ASCII del segundo.
- No será válido comparar cadenas de caracteres (cadena) con datos numéricos.
- No será válido comparar cadenas de caracteres (cadena) o datos numéricos (entero, decimal, caracter) con valores lógicos (booleano).
- No será válido comparar valores lógicos (booleano) entre sí.

*Tabla 9: operadores relacionales de **D++***

OPERADOR	USO	SIGNIFICADO	VALOR RESULTANTE
`==`	`A == B`	A es igual que B	verdadero, si A es igual que B falso, si A no es igual que B
`<>`	`A <> B`	A es diferente que B	verdadero, si A es diferente que B falso, si A no es diferente que B
`<`	`A < B`	A es menor que B	verdadero, si A es menor que B falso, si A no es menor que B
`>`	`A > B`	A es mayor que B	verdadero, si A es mayor que B falso, si A no es mayor que B
`<=`	`A <= B`	A es menor o igual que B	verdadero, si A es menor o igual que B falso, si A no es menor o igual que B
`>=`	`A >= B`	A es mayor o igual que B	verdadero, si A es mayor o igual que B falso, si A no es mayor o igual que B

4.3.6 Operaciones lógicas

Las operaciones lógicas son expresiones matemáticas cuyo resultado será valor lógico (booleano). Las operaciones lógicas se basan en el álgebra de Boole.

*Tabla 10: tabla de verdad para operadores booleanos de **D++***

OPERADOR A	OPERADOR B	A | | B	A &&B	!A
falso	falso	falso	falso	verdadero
falso	verdadero	verdadero	falso	verdadero
verdadero	falso	verdadero	falso	falso
verdadero	verdadero	verdadero	verdadero	falso

A continuación, se presenta la jerarquía de operadores lógicos.

*Tabla 11: precedencia y asociatividad de operadores lógicos en **D++***

NIVEL	OPERADOR	ASOCIATIVIDAD
0	Or "||"	Izquierda
1	And "&&"	Izquierda
2	Not "!"	Derecha

4.3.7 Declaración de variables

Una variable deberá ser declarada antes de poder ser utilizada. Para la declaración de variables será necesario empezar con el tipo de dato de la variable seguido del identificador que esta tendrá. La variable podrá o no tener un valor de inicialización. Es posible declarar dos o más variables a la vez, en tal caso los identificadores estarán separados por comas.

SINTAXIS

```
TIPO LISTA_ID (= EXPRESION)?;
```

EJEMPLO

```
// Declaración de una variable
entero var1;
// Declaración e inicialización de varias variables
caracter var1, var2, var3 = 0;
```

Declarar es el proceso de asignar suficiente espacio de memoria para los datos en términos del tipo de la variable. A continuación, se muestra un ejemplo de ese proceso.

```
entero b = 20;
```

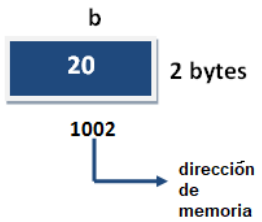


Ilustración 23: proceso de declarar una variable

Tabla 12: valores por defecto en tipos primitivos

TIPO DE DATO	VALOR POR DEFECTO
caracter	'\u0000' caracter nulo
entero decimal	0
booleano	falso

NOTAS

- No será posible declarar variables con el mismo identificador si pertenecen al mismo contexto o entorno, si esto sucede debe reportarse un error semántico.
- Deberá verificarse que el tipo de la expresión de inicialización sea el mismo que el declarado, de lo contrario, debe reportarse un error semántico.

4.3.8 Asignación de variables

Una asignación de variable consistirá en asignar un valor a una variable. La asignación será llevada a cabo con el signo igual (=).

SINTAXIS

```
identificador = EXPRESION;
```

EJEMPLO

```
entero a = 20;
// asignación directa
b = 25;
// asignación en términos de otra variable
b = a;
// asignación en términos de una expresión
var3 = 15 * 2 + 10;
```

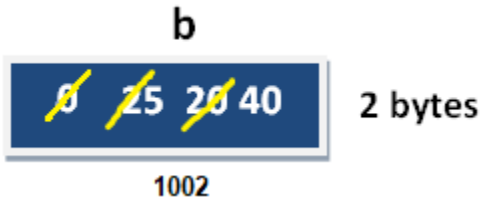


Ilustración 24: proceso de asignar una variable

NOTAS

- Deberá verificarse que el tipo de la expresión sea el mismo que el declarado, de lo contrario, debe reportarse un error semántico.
- Deberá verificarse que la variable a las que se le está asignando la expresión, exista en el contexto o entorno que contiene dicha sentencia de asignación.

4.3.9 Arreglos de una dimensión

Un arreglo será un conjunto de datos de tipos similares que se identifican con un nombre común.

Los diferentes elementos contenidos en un arreglo se definen por un índice y se acceden a ellos utilizando su índice; los índices arrancan en 0.

SINTAXIS

```
// Declaración de un arreglo
TIPO identificador [EXPRESION] (= {dato1, ..., datoN})?;
// EXPRESION define el tamaño N del arreglo.
// dato1,…,datoN representan una lista de EXPRESION
```

EJEMPLO

```
// Declaración de un arreglo de enteros de tamaño N = 4.
entero factorial5[4] = {1, 2, 6, 24, 120};
```

NOTAS

- Deberá verificarse que la variable que representa al arreglo, no exista en el contexto o entorno que contiene dicha sentencia de declaración.
- Deberá verificarse que las dimensiones declaradas del arreglo coincidan con las dimensiones que se definan con la expresión de inicialización, de no ser así, debe reportarse un error semántico.
- Deberá verificarse que el tipo de los elementos de inicialización sea el mismo que el declarado, de lo contrario, debe reportarse un error semántico.

4.3.10 Acceso a posiciones dentro de un arreglo

Se podrá acceder a posiciones dentro de un arreglo, especificando la posición a la cual se desea acceder.

Existen dos posibles funcionalidades de un acceso:

Acceder a una posición de un arreglo para utilizar su valor

En este caso se accede al valor de una posición específica para su utilización en alguna otra sentencia.

SINTAXIS

```
identificador [EXPRESION];
```

EJEMPLO

```
var1 = arreglo[0];
```

Acceder a una posición de un arreglo para asignarle un valor

En este caso se accede una posición específica para modificar el valor actual con el valor implícito de otra expresión.

SINTAXIS

```
identificador [EXPRESION] = EXPRESION;
```

EJEMPLO

```
factorial5[0] = 0;
```

NOTAS

- Deberá verificarse que la variable que representa al arreglo, exista en el contexto o entorno que contiene dicha sentencia de asignación.
- Deberá verificarse que las expresiones que representan los índices para acceder a una posición especifica del arreglo, estén dentro del rango definido previamente en la declaración del mismo.
- Deberá verificarse que, si se asignara un valor a una posición especifica del arreglo, el tipo de la expresión sea el mismo que el declarado.

4.3.11 Arreglos multidimensionales

Estos arreglos serán arreglos especiales ya que sus elementos pueden ser otros arreglos. Se conforman por **expresiones subíndice**, dichas expresiones definen el contenido de cada elemento interno.

SINTAXIS

```
// Declaración de un arreglo de una o más dimensiones
TIPO identificador [EXPRESION1] [EXPRESION2] … [EXPRESIONn];
// Declaración e inicialización de un arreglo de una o más dimensiones
TIPO identificador [EXPRESION] [EXPRESION2] … [EXPRESIONn]= {{dato1,
..., datoN}, {dato1, ..., datoN} .... {dato1, ..., datoN}};
```

Las expresiones de subíndice se asocian de izquierda a derecha. La expresión de subíndice del extremo izquierdo, `identificador [EXPRESION1]` se evalúa primero. La dirección resultante forma una expresión de puntero; entonces, se agrega `EXPRESION2` a esta expresión de puntero para formar una nueva expresión de puntero, y así sucesivamente, hasta que se haya agregado la última expresión de subíndice, la cual será un arreglo de una dimensión con elementos de tipo primitivo (carácter, entero, decimal o booleano).

Las expresiones con varios subíndices hacen referencia a elementos de “arreglos multidimensionales”.

Un arreglo multidimensional es un arreglo cuyos elementos son arreglos. Por ejemplo, el primer elemento de un arreglo tridimensional es un arreglo con dos dimensiones.

EJEMPLO

```
// Declaración de un arreglo de tres dimensiones.
decimal arreglo[3][6][2];
```

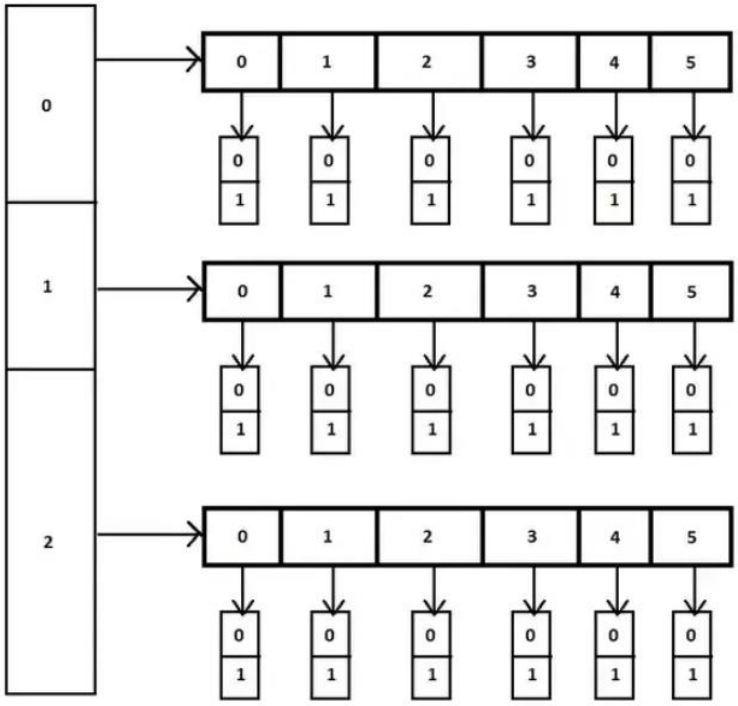


Ilustración 25: visualización de un arreglo multidimensional

NOTAS

- Deberá verificarse que la variable que representa al arreglo, no exista en el contexto o entorno que contiene dicha sentencia de declaración.
- Deberá verificarse que las dimensiones declaradas del arreglo coincidan con las dimensiones que se definan con la expresión de inicialización, de no ser así, debe reportarse un error semántico.
- Deberá verificarse que el tipo de los elementos de inicialización sea el mismo que el declarado, de lo contrario, debe reportarse un error semántico.

4.3.12 Acceso a posiciones dentro de un arreglo multidimensional

Se podrá acceder a posiciones dentro del arreglo, especificando la posición a la cual se desea acceder. Existen dos posibles funcionalidades de un acceso:

Acceder a una posición de un arreglo para utilizar su valor

En este caso se accede al valor de una posición específica para su utilización en alguna otra sentencia.

SINTAXIS

```
identificador [EXPRESION1] [EXPRESION2] … [EXPRESIONn] = EXPRESION;
```

EJEMPLO

```
decimal var = 96 + arreglo[3][6][2] * 1.5 ^ 2;
```

Acceder a una posición de un arreglo para asignarle un valor

En este caso se accede una posición específica para modificar el valor actual con el valor implícito de otra expresión.

SINTAXIS

```
identificador [EXPRESION1] [EXPRESION2] … [EXPRESIONn] = EXPRESION;
```

EJEMPLO

```
arreglo[3][6][2]= 1.5;
```

NOTAS

- Deberá verificarse que la variable que representa al arreglo, exista en el contexto o entorno que contiene dicha sentencia de asignación.
- Deberá verificarse que las expresiones que representan los índices para acceder a una posición especifica del arreglo, estén dentro del rango definido previamente en la declaración del mismo y el número de dimensiones sea correcta.
- Deberá verificarse que, si se asignara un valor a una posición especifica del arreglo, el tipo de la expresión sea el mismo que el declarado.

4.3.13 Sentencias de selección

Operador ternario

Esta sentencia **retornará** una expresión en función de la veracidad de una condición lógica.

SINTAXIS

```
CONDICION? EXPRESION1: EXPRESION2 ;
```

Si la condición es verdadera entonces el valor resultante será EXPRESION1, de lo contrario si la condición fuera falsa el valor retornado será EXPRESION2.

EJEMPLO

```
entero var1 = 1;
cadena cadenita = var1==0? “var1 es 0”: “var1 NO es 0”;
// Ya que var1 no es igual a 0 entonces cadenita = “var1 es 0”
```

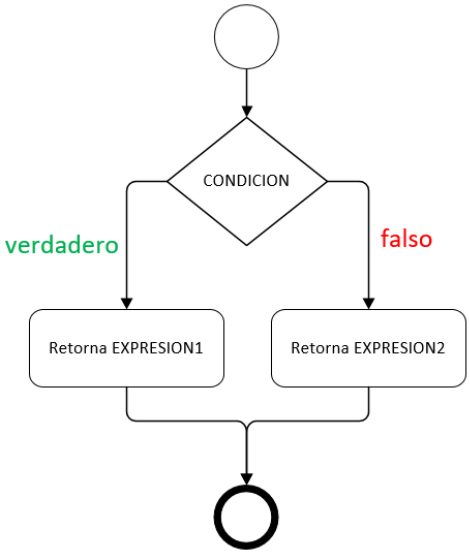


*Ilustración 26: diagrama de flujo para un **operador ternario***

Si

Esta sentencia de selección bifurcará el flujo de ejecución ya que proporciona control sobre dos alternativas basadas en el valor lógico de una expresión (condición).

Sintaxis

```
si(CONDICION) BLOQUE_SENTENCIAS
(sino si (CONDICION) BLOQUE_SENTENCIAS) *
[sino BLOQUE_SENTENCIAS]
```

Ejemplo

```
si (variable > 0) {
     var1 = 0;
} sino si (variable < 0) {
     var1 = 1;
} sino {
     var1 = 2;
}
```

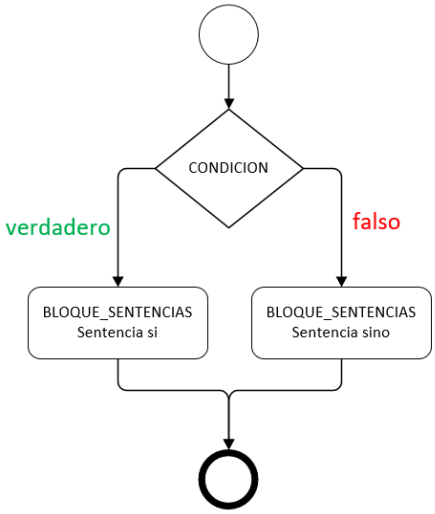


*Ilustración 27: diagrama de flujo para la sentencia de selección **si***

4.3.14 Sentencias cíclicas o bucles

Mientras

La sentencia cíclica **mientras** se utilizará para crear repeticiones de sentencias en el flujo del programa. El bloque de sentencias se ejecutará mientras la condición sea verdadera $(0\ a\ n\ veces)$, de lo contrario el programa continuará con su flujo de ejecución.

SINTAXIS

```
mientras(CONDICION) BLOQUE_SENTENCIAS
```

EJEMPLO

```
mientras (cuenta <= 10) {
      cuenta++;
}
```

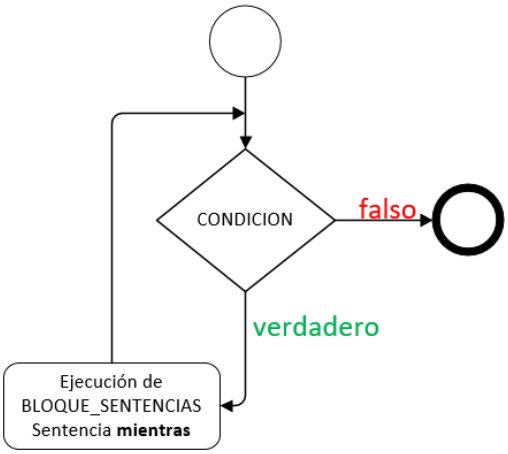


*Ilustración 28: diagrama de flujo para la sentencia cíclica **mientras***

Para

La sentencia cíclica **para** permitirá inicializar una variable como variable de control, el ciclo tendrá una condición que se verificará en cada iteración, luego se deberá definir una operación que actualice la variable de control cada vez que se ejecuta un ciclo para luego verificar si la condición se cumple.

SINTAXIS

```
para(INICIALIZACION; CONDICION ; ACTUALIZACION) BLOQUE_SENTENCIAS
```

EJEMPLO

```
para (entero cuenta = 0; cuenta <= 10; cuenta++) {
     a[cuenta] = cuenta;
}
```

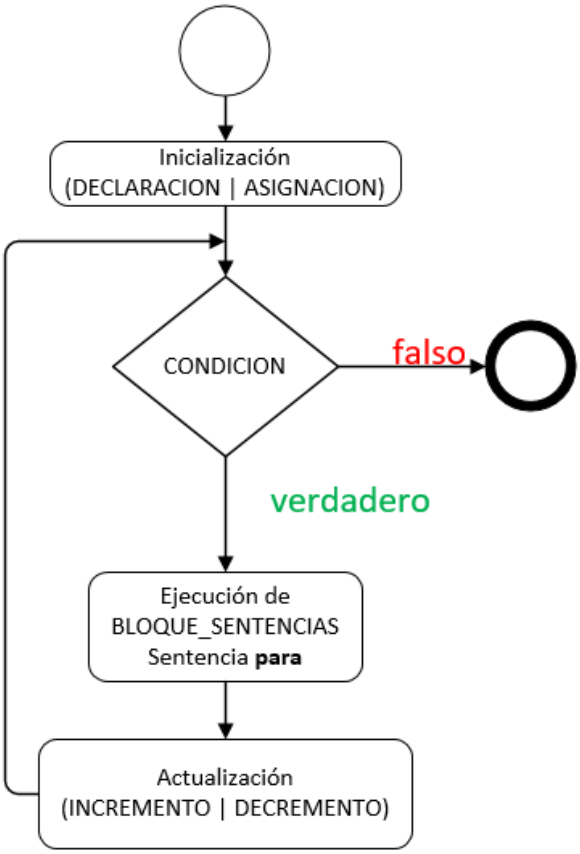


*Ilustración 29: diagrama de flujo de sentencia cíclica **para***

4.3.15 Sentencias de transferencia

D++ soportará tres sentencias de salto o transferencia: **detener**, **continuar** y **retornar**. Estas sentencias transferirán el control a otras partes del programa.

Detener

La sentencia **detener** tendrá dos usos:

- Terminar un bloque de sentencias en una sentencia **selecciona**
- Terminar la ejecución de un ciclo.

SINTAXIS

```
detener;
```

EJEMPLO

```
mientras (cuenta <= 10) {
      si (cuenta == 5) {
            detener;
      }
      cuenta++;
}
```

NOTAS

- Deberá verificarse que la sentencia detener aparezca únicamente dentro de un ciclo o dentro de una sentencia seleccionar.
- En el caso de sentencias cíclicas, la sentencia **detener** únicamente detendrá la ejecución del bloque de sentencias asociado a la sentencia cíclica que la contiene.
- En el caso de sentencia seleccionar, la sentencia **detener** únicamente detendrá la ejecución del bloque de sentencias asociado a la sentencia case que la contiene.

Continuar

La sentencia **continuar** se utilizará para transferir el control al principio del ciclo, es decir, continuar con la siguiente iteración.

SINTAXIS

```
continuar;
```

EJEMPLO

```
mientras (cuenta <= 10) {
      si (cuenta == 5) {
            continuar;
      }
      cuenta++;
}
```

NOTAS

- Deberá verificarse que la sentencia **continuar** aparezca únicamente dentro de un ciclo y afecte solamente las iteraciones de dicho ciclo.

Retornar

La sentencia **retornar** se empleará para salir de la ejecución de sentencias de un método y, opcionalmente, devolver un valor. Tras la salida del método se vuelve a la secuencia de ejecución del programa al lugar de llamada de dicho método**.**

SINTAXIS

```
retornar [EXPRESION];
```

EJEMPLO

```
retornar 0;
```

NOTAS

- Deberá verificarse que la sentencia **retornar** aparezca únicamente dentro de un método o función.
- Debe verificarse que en funciones la sentencia **retornar**, retorne una expresión del mismo tipo del que fue declarado en dicha función.
- Debe verificarse que la sentencia **retornar**, sin una expresión asociada este contenida únicamente en métodos (funciones de tipo vacío).

4.3.16 Sentencia de imprimir

Esta sentencia recibirá una expresión y la mostrará en la consola de salida.

SINTAXIS

```
imprimir(EXPRESION);
```

EJEMPLO

```
mientras (cuenta <= 10) {
      imprimir ("La cuenta es: " + cuenta);
      cuenta++;
}
```

4.3.17 Sentencia importar

La sentencia **importar** permitirá hacer referencia a código del lenguaje **D++**. La sentencia recibirá una cadena que contenga la ruta del archivo. La ruta del archivo deberá ser la dirección física del archivo.

SINTAXIS

```
importar("cadena");
```

EJEMPLO

```
importar("/home/javiern/prueba.dpp");
```

4.3.18 Declaración de una estructura

La declaración de una **estructura** designa un tipo y especifica una secuencia de valores variables (denominados “miembros” o “campos” de la estructura) que pueden tener diferentes tipos. Un identificador denominado “etiqueta”, proporciona el nombre del tipo de estructura y se puede usar en referencias posteriores al tipo de estructura. Una variable de este tipo de estructura contiene la secuencia completa definida por el tipo.

La declaración de un tipo de estructura no reserva espacio para una estructura. Es solo una plantilla para declaraciones posteriores de variables de estructura.

Diferencia entre un arreglo y una variable de estructura

En **D++** un arreglo es un tipo de dato definido por el usuario, pero un arreglo solo es capaz de almacenar elementos del mismo tipo, a diferencia de una **estructura** que tiene capacidad de manejar elementos de diferente tipo.

SINTAXIS

```
estructura identificador // representa la etiqueta de la estructura
{LISTA_MIEMBROS} // una lista de declaraciones separadas por “;”
```

`LISTA_MIEMBROS` especifica los tipos y los nombres de los miembros de la estructura.

Miembros de una estructura

Cada variable declarada en `LISTA_MIEMBROS` se define como un miembro del tipo de estructura. Las declaraciones de variable dentro de `LISTA_MIEMBROS` tienen el mismo formato que otras declaraciones de variable, con la salvedad de que no pueden contener inicializadores. Los miembros de la estructura pueden tener cualquier tipo de variable excepto el tipo vacio.

EJEMPLO

```
estructura Estudiante {
      entero carne;
      cadena nombre = “cadena de inicio”;
      decimal arreglo[3][6][2];
}
```

4.3.19 Métodos

Las declaraciones de métodos están compuestas por un identificador, una lista de parámetros y un bloque de sentencias. El tipo de retorno, si el mismo existiera debe ser vacío. La lista de parámetros consta de declaraciones de tipo (sin valores iniciales) separados por comas. La lista de parámetros podría estar vacía.

SINTAXIS

```
vacio identificador(LISTA_PARAMETROS) BLOQUE_SENTENCIAS
```

EJEMPLO

```
vacio metodo(entero carne){
carne = 201513630;
}
```

4.3.20 Funciones

Las declaraciones de funciones están compuestas por un tipo, un identificador, una lista de parámetros y un bloque de sentencias. El tipo de retorno debe ser no vacío. La lista de parámetros consta de declaraciones de tipo (sin valores iniciales) separados por comas. La lista de parámetros podría estar vacía.

SINTAXIS

```
TIPO identificador(LISTA_PARAMETROS) BLOQUE_SENTENCIAS
```

EJEMPLO

```
entero getCarne(){
      retornar 201513630;
}
```

4.3.21 Llamada de métodos o funciones

Cuando se llama a un método, se deben proporcionar los argumentos del tipo adecuado y la cantidad.

SINTAXIS

```
identificador(LISTA_EXPRESION);
```

EJEMPLO

```
getCarne();
```

4.3.22 Método principal

Este método dará inicio a la ejecución de las acciones. Dentro del método principal se indicará el orden en que se ejecutarán las sentencias ingresadas en alto nivel.

SINTAXIS

```
vacio principal() BLOQUE_SENTENCIAS
```

EJEMPLO

```
principal () {
     imprimir ("/****inicializando variables****/");

     //se colocan los valores de op1 y op2
     inicia_vars ();
     imprimir ("suma:" + suma (este.op1, este.op2));
}
```

4.3.23 Creación de variables de estructura

Para la utilización de variables de tipo **estructura** se deberá contar con la definición de la **estructura** previamente creada.

SINTAXIS

```
identificador1 identificador2 ;
```

Donde `identificador1` de tipo **estructura** se deberá contar con la definición de la **estructura** previamente creada.

EJEMPLO

```
Estudiante Jorge;
```

4.3.24 Acceso a atributos de una estructura

Para acceder a un miembro de una variable de estructura se deberá de escribir el identificador de la variable de estructura, seguido de un punto (.) y el identificador del miembro que se quiera acceder.

SINTAXIS

```
identificador_VariableEstructura.identificador_Miembro;
```

EJEMPLO

```
Persona Carlos;
Carlos.nombre = Jorge.nombre;
Carlos.edad = 23;
```

4.3.25 Funciones nativas de dibujo

El lenguaje de alto nivel permitirá poder ejecutar funciones nativas, estas funciones son las misma que pertenecen al lenguaje interpretado de DracoScript.

Pintar punto

Este método tomará una serie de parámetros con los cuales podrá dibujar un punto en una posición en específica del área del micro navegador. El punto a dibujar la posición [X,Y] deberá ser el centro del punto.

SINTAXIS

```
punto (posición en X, posición en Y, color, Diámetro);
```

- El primer parámetro deberá ser de tipo entero e indicará la posición en X del área del micro navegador donde se pintará el puto.
- El segundo parámetro deberá ser de tipo entero e indicará la posición en Y del área del micro navegador donde se pintará el punto.
- El tercer parámetro deberá ser de tipo cadena e indicará el color del punto. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.
- El cuarto parámetro deberá ser de tipo entero e indicará el diámetro medido en pixeles de punto.

EJEMPLO

```
punto (100,100, “#000000”, 50);
punto (50,105, “#FFFF”, 50);
```

Pintar Cuadrado

Este método tomará una serie de parámetros con los cuales podrá dibujar un cuadrado es decir con altura y anchura con tamaños distintos en una posición en específica del área del micro navegador. La posición [X, Y] será la esquina superior izquierda del cuadrado y dibujará hacia la derecha y hacia abajo, siendo estas su ancho y altura.

SINTAXIS

```
cuadrado (posición en X, posición en Y, color, Ancho, Alto)
```

- El primer parámetro deberá ser de tipo entero e indicará la posición en X del área del micro navegador donde se pintará el rectángulo.
- El segundo parámetro deberá ser de tipo entero e indicará la posición en Y del área del micro navegador donde se pintará el rectángulo.

- El tercer parámetro deberá ser de tipo cadena he indicará el color del rectángulo. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.
- El cuarto parámetro deberá ser de tipo entero e indicará el ancho medido en pixeles del rectángulo.
- El quinto parámetro deberá ser de tipo entero e indicará el alto medido en pixeles del rectángulo.

EJEMPLO

```
cuadrado (100, 500, “#F44000”, 150, 200)
```

Pintar Ovalo

Este método tomará una serie de parámetros con los cuales podrá dibujar un ovalo es decir con altura y anchura con tamaños distintos en una posición en específica del área del micro navegador.

SINTAXIS

```
ovalo (posición en X, posición en Y, color, Ancho, Alto)
```

- El primer parámetro deberá ser de tipo entero e indicará la posición en X del área del micro navegador donde se pintará el óvalo.
- El segundo parámetro deberá ser de tipo entero e indicará la posición en Y del área del micro navegador donde se pintará el óvalo.
- El tercer parámetro deberá ser de tipo cadena e indicará el color del óvalo. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.
- El cuarto parámetro deberá ser de tipo entero e indicará el ancho medido en pixeles del óvalo.
- El quinto parámetro deberá ser de tipo entero e indicará el alto medido en pixeles del óvalo.

EJEMPLO

```
ovalo (100, 500, “#F44000”, 150, 200)
```

Pintar cadena

Este método tomará una serie de parámetros con los cuales podrá dibujar una cadena en una posición en específica del área del micro navegador.

SINTAXIS

```
cadena (posición en X, posición en Y, color, Cadena)
```

- El primer parámetro deberá ser de tipo entero he indicará la posición en X del área del micro navegador donde se pintará el puto.
- El segundo parámetro deberá ser de tipo entero he indicará la posición en Y del área del micro navegador donde se pintará el punto.
- El tercer parámetro deberá ser de tipo cadena he indicará el color del punto. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.
- El cuarto parámetro deberá ser de tipo cadena y será el texto a desplegar en el área del micro navegador.

Nota: El grosor del texto queda a discreción del estudiante, pero deberá se legible.

EJEMPLO

```
cadena (100, 500, “#F44000”, “Hola mundo”);
```

Pintar línea

Este método tomará una serie de parámetros con los cuales podrá dibujar una línea en una posición en específica del área del micro navegador.

SINTAXIS

```
linea (posición en Xi, posición en Yi, posición Xf, posición Yf, color, grosor )
```

- El primer parámetro deberá ser de tipo entero e indicará la posición en Xi del área del micro navegador donde iniciar a pintar la línea.
- El segundo parámetro deberá ser de tipo entero e indicará la posición en Yi del área del micro navegador donde iniciar a pintar la línea.
- El tercer parámetro deberá ser de tipo entero e indicará la posición en Xf del área del micro navegador donde finalizará de pintar la línea.
- El cuarto parámetro deberá ser de tipo entero e indicará la posición en Yi del área del micro navegador donde finalizará de pintar la línea.
- El quinto parámetro deberá ser de tipo cadena e indicará el color de la línea. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.
- El sexto parámetro deberá ser de tipo entero e indicará el grosor de la línea medido en pixeles.

EJEMPLO

```
linea  (100, 500, 150, 200, “#F44000”, 15);
```

Nota: El color de usado en todas las funciones nativas es el color de relleno de la figura.

5 Lenguaje interpretado DracoScript

Lenguaje de alto nivel, interpretado, orientado al desarrollo web, compuesto por un subconjunto de sentencias semejantes a las del popular lenguaje interpretado JavaScript.

El objetivo del mismo será la implementación de funcionalidades con grado de complejidad bajo, además de permitir la integración de funcionalidades complejas desarrolladas utilizando el lenguaje D++ y traducidas por Draco Compiler.

El lenguaje DracoScript será el encargado de realizar gran parte de la lógica y manejo de datos que el micro navegador deberá de realizar. Este lenguaje se encontrará en archivos con extensión “djs”.

A continuación, se describen los componentes que conformarán la aplicación.

5.1 Notación dentro del enunciado

Dentro del enunciado, se utilizarán las siguientes notaciones cuando se haga referencia al código de alto nivel.

Sintaxis y ejemplos

Para definir la sintaxis y ejemplos de sentencias de código de alto nivel se utilizará un rectángulo gris.

```
```

Asignación de colores

Dentro del enunciado y en el editor de texto de la aplicación, para el código de alto nivel, seguirá el formato de colores establecido en la siguiente tabla. Estos colores también deberán de ser implementados en el editor de texto.

Tabla 13: código de colores.

Color	Token
Azul	Palabras reservadas
Naranja	Cadenas, caracteres
Morado	Números
Gris	Comentario
Negro	Otro

Palabras reservadas

Son palabras que no podrán ser utilizadas como identificadores ya que representan una instrucción específica y necesaria dentro del lenguaje.

Expresiones

Cuando se haga referencia a una ‘expresión’, se hará referencia a cualquier sentencia que devuelve un valor, ya sea una operación aritmética, una operación relacional, una operación lógica, un atributo, variable, parámetro, función.

Los tipos de datos para cada expresión vendrán dados por los sistemas de tipos definidos para las operaciones y del tipo de datos asociados al resto de elementos del lenguaje.

5.2 Características generales

Esta sección describirá las características que contiene el lenguaje DracoScript.

5.2.1 Case sensitive

El lenguaje es case insensitive, esto significa que no existirá diferencia entre mayúsculas y minúsculas, esto aplicará tanto para palabras reservadas propias del lenguaje como para identificadores. Por ejemplo, si se declara un identificador de nombre “Contador” este no será distinto a definir un identificador “contador”.

5.2.2 Tipos de Dato

Para este lenguaje se utilizará los tipos de dato entero, decimal, cadena, carácter y booleano.

A continuación, se definen los tipos de dato a utilizar:

- Entero: este tipo de dato, como su nombre lo indica, aceptará valores numéricos enteros.
- Decimal: este tipo de dato, como su nombre lo indica, aceptará valores numéricos decimales.
- Booleano: este tipo de dato aceptará valores de verdadero (true) y falso (false) que serán representados con palabras reservadas que serán sus respectivos nombres.
- Carácter: este tipo de dato aceptará un único carácter, que deberá ser encerrado en comillas simples (‘carácter’).
- Cadena: este tipo de dato aceptará cadenas de caracteres, que deberán ser encerradas dentro de comillas dobles (“caracteres”).

5.2.3 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.

Ejemplo

```
55 * ((85 - 3) / 8)
```

5.2.4 Comentarios

El lenguaje DacroScript, permitirá agregar comentarios los cuales podrán ser simples o múltiples líneas. Esto con la finalidad que el programador pueda dejar información respecto al código que está generando.

Comentario Simple: este utilizará el carácter **"$$"**, para dar inicio a un comentario.

Comentario con múltiples líneas: el comentario doble iniciará con **"$*"** y terminará con **"*$"**.

Ejemplo

```
$$ Comentario simple
$* Se agregó un comentario
de
tres líneas *$
```

5.2.5 Variables

Las variables serán contenedores en que se podrán almacenar valores, una variable podrá tener cualquier tipo de dato asociado. Las variables podrán ser llamadas desde cualquier lugar del archivo de DracoScript.

5.3 Sintaxis

En este punto se definirá la sintaxis del lenguaje DracoScript y la manera correcta de funcionamiento.

5.3.1 Declaración de Variables.

Para la declaración de variables primero se deberá declarar la variable con la palabra reservada **var**, seguida del nombre que le quiera dar. Las variables que se declaren afuera de los métodos serán variables globales.

Sintaxis

```
Var Nombre_de_Variable;
Var var1, var2, var3, varN…;
Var Nombre_de_Variable :: Expresion;
Var var1 :=: expresión, var2, var3 :=: expresión;
```

Ejemplo

```
Var variable1;
Var var1, var2, var3;
Var var1 :=: 15, var2, var3 :=: 20;
```

5.3.2 Asignación de valores

Las variables podrán tomar cualquier tipo de valor, por lo que no es necesario especificar un tipo de valor en específico. El signo para la asignación de variables será el "**:=:**".

Sintaxis

```
Var Nombre_de_Variable :: valor;
Nombre_de_Variable :: valor;

Var var1 :=: 15, var2, var3 :=: 20;
```

Ejemplo

```
Var NOMBRE :=: "Juan";
Nombre :=: "Pedro";
```

5.3.3 Tipos de datos

La declaración de tipos de variables será implícita en el lenguaje CJS, este tomará el tipo dependiendo del valor asignado a esta variable, los tipos de variable soportados serán los descritos en la siguiente tabla.

Tipo	Descripción	Formato
Cadena	Las variables que se le asigne los valores de tipos texto, deberán ir entre comillas dobles **" ".**	Var mivariable :=: "Hola mundo";
Carácter	Las variables que se le asigne los valores de tipos texto, deberán ir entre comillas simples ' '**.**	Var mivariable :=: 'J';
Booleano	Son valores de verdadero y falsos, las palabras reservadas para estos serán 'True' y 'False	Var mivariable :=: true;
Numérico	Estas son los valores numéricos que puede tomar las variables, pueden ser tanto enteros como decimales.	Var mivariable :=: 100; Var mivariable :=: 90.90;

5.3.4 Operadores

Las operaciones que soportará el lenguaje DracoScript, se encuentran en la siguiente tabla.

Operador	Nombre	Función	Ejemplo
+	Suma	Operación aritmética que consiste en sumar varias cantidades en una sola.	A :=: 5+5+10+6 B :=: 10.5+10.5

			C :=: 95+0.5+1.5
-	Resta	Operación aritmética que consiste en restar o quitar cantidades y obtener la diferencia	D :=: 100-0.5-95 E :=: 50-1.5-5
*****	Multiplicación	Operación aritmética que consiste en multiplicar es decir sumar las cantidades las veces indicadas a otro número.	F :=: 10*15*10 G :=: 30*0.5*2
/	División	Operación aritmética que devolver el resultado de una división.	H :=: 10/2 I :=: 200/5
^	Potencia	Operación aritmética de potencia, toma la primera expresión como base y la segunda como exponente.	a :=: b^2;
%	Modulo	Operación aritmética que devolver el residuo de una división.	a :=: b%2;
++	Adición	Suma 1 a la expresión.	a++; (10 + 15) ++
--	Sustracción	Resta 1 a la expresión.	(b+2) -- B--

Los operadores aritméticos deberán de cumplir con las reglas de operatoria que se encuentran en la siguiente tabla.

Suma(+)	**Booleano**	**Numérico**	**Cadena**	**Carácter**
Booleano	OR	Sumar	Concatenar	Concatenar
Numérico	Sumar	Sumar	Concatenar	Concatenar
Cadena	Concatenar	Concatenar	Concatenar	Concatenar
Carácter	Sumar (Ascii)	Sumar (Ascii)	Concatenar	Concatenar
Resta(-)				
Booleano	Error	Restar	Error	Error
Numérico	Restar	Restar	Error	Error
Cadena	Error	Restar (Ascii)	Error	Error
Carácter	Restar (Ascii)	Restar (Ascii)	Error	Error
Multiplicación(*)				
Booleano	AND	Multiplicar	Error	Error
Numérico	Multiplicar	Multiplicar	Error	Error
Cadena	Error	Error	Error	Error

Carácter	Multiplicar (Ascii)	Multiplicar (Ascii)	Error	Error
División (/)				
Booleano	Error	Dividir	Error	Error
Numérico	Dividir	Dividir	Error	Error
Cadena	Error	Error	Error	Error
Carácter	Dividir (Ascii)	Dividir (Ascii)	Error	Error
Potencia(^)				
Booleano	Error	Elevar	Error	Error
Numérico	Elevar	Elevar	Error	Error
Cadena	Error	Error	Error	Error
Carácter	Elevar (Ascii)	Elevar (Ascii)	Error	Error

5.3.5 *Condiciones*

Las condiciones son operaciones que servirán para poder determinar la veracidad o falsedad de la misma. Para ello se hace uso de operaciones aritméticas, relacionales y lógicas.

5.3.6 Expresiones relacionales

Las operaciones relacionales servirán para comparar dos expresiones, el sistema deberá de manejar las operaciones de igualdad, diferente, menor, mayor, menor o igual y mayor o igual. Las expresiones relacionales darán como resultado un valor booleano el cual podrá ser verdadero (true) o falso (false).

Sintaxis

```
Expresion Operador Expresion;
```

En la siguiente tabla se describirá el uso de las expresiones relacionales.

Nombre	Símbolo	Descripción
Igual	**==**	Esta operación comprobará si dos expresiones tienen el mismo valor, de ser así retorna verdadero (true) de lo contrario retorna falso (false).
Diferente	**!=**	Esta operación comprobará si dos expresiones tienen distinto valor, de ser así retorna verdadero (true) de lo contrario retorna falso (false).
Menor que	**<**	Esta operación comprobará si la primera expresión es menor a la segunda expresión, de ser así retorna verdadero (true) de lo contrario retorna falso (false).

Mayor que	**>**	Esta operación comprobará si la primera expresión es mayor a la segunda expresión, de ser así retorna verdadero (true) de lo contrario retorna falso (false).
Menor o igual	**<=**	Esta operación comprobará si la primera expresión es menor o igual a la segunda expresión, de ser así retorna verdadero (true) de lo contrario retorna falso (false).
Mayor o igual	**>=**	Esta operación comprobará si la primera expresión es mayor o igual a la segunda expresión, de ser así retorna verdadero (true) de lo contrario retorna falso (false).

5.3.7 Expresiones lógicas

Estas expresiones utilizarán los conectores lógicos para unir dos condiciones. Sus operandos tienen que ser de tipo booleano, de lo contrario se deberá reportar error. El resultado de la operación será de tipo booleano. En la siguiente tabla se describen los operadores lógicos a implementar.

Nombre	Símbolo	Descripción
AND	&&	Este operador comprobará que tanto la primera expresión como la segunda expresión tengan valor verdadero. Si esto se cumple retorna 1 (verdadero) de lo contrario retorna 0(falso).
OR	||	Este operador comprobará que la primera expresión o la segunda expresión tenga valor verdadero. Si esto se cumple retorna 1 (verdadero) de lo contrario retorna 0(falso).
NOT	**!**	Este operador negará el valor de la expresión con la que viene acompañada. La expresión que negara, si es verdadera retornara falso y si falsa retornara verdadera.

5.3.8 Precedencia de operadores y asociatividad

En la tabla que se presenta a continuación se definen la precedencia y asociatividad de cada uno de los operadores definidos anteriormente, ordenados de mayor precedencia a menor precedencia. Los paréntesis son los únicos símbolos de agrupación a utilizar.

Símbolo	Precedencia	Asociatividad
()	9	Izquierda a derecha
++ -- ! +(unario) –(unario)	8	Derecha a izquierda
/ * %	7	Izquierda a derecha
+ -	6	Izquierda a derecha
< <= > >=	5	Izquierda a derecha
== !=	4	Izquierda a derecha
&&	3	Izquierda a derecha
||	2	Izquierda a derecha

5.3.9 Sentencias de control

Sentencia If

Sentencia cuya función determinará el flujo que el programa deberá seguir entre una acción u otra dependiendo de la condición establecidas por el programador.

Sintaxis:

```
if (expresion_logica) {
   $$ instrucciones
}
```

Ejemplo

```
If (a < b) {
   $$ instrucciones
}
```

Sentencia Si Sino

Se deberá utilizar cuando se desea tener varias instrucciones en el caso que la condición sea verdadera y otras instrucciones cuando la condición sea falsa.

Sintaxis

```
if (a < b) {
   $$ instrucciones
} if not {
  $$ instrucciones
}
```

Sentencia Si Sino Si

Cuando se necesitan diferentes acciones para diferentes casos (condiciones), será posible utilizar esta sentencia, esta sentencia se podrá agregar varias instrucciones que se ejecuten en el caso que no se cumpla ninguna condición.

Sintaxis

```
If (a < b) {
   $$ instrucciones
} elif (a < b) {
  $$ instrucciones
} elif (a < b) {
  $$ instrucciones
} if not {
  $$ instrucciones
}
```

5.3.10 Bucles

Un bucle o ciclo, es una instrucción que deberá ejecutar cada sentencia contenida en ella repetidas veces hasta que una condición asignada deje de cumplirse.

- **Sentencia Romper:** Esta sentencia se utilizará para salir de cualquier bucle, romper solo puede ser utilizada siempre que se encuentre dentro de un bucle.

Sintaxis

```
Smash;
```

Sentencia Mientras

Es un ciclo que se ejecutará mientras una condición sea verdadera.

Sintaxis

```
while (condicion)
{
   $$instrucciones
}
```

Ejemplo

```
while (a < b)
{
   a = a + 1;
}
```

Sentencia Para

Es un bucle que permitirá inicializar o establecer una variable como variable de control, el ciclo contendrá una condición que se verificará en cada iteración, luego se deberá definir una operación que afecte directamente a la variable de control cada vez que se ejecuta un ciclo para volver a verificar si se cumple la condición. Su uso es ideal para los casos en donde se conocerá el número de repeticiones o se necesitará hacer un barrido a vectores o matrices.

Sintaxis

```
for (variable de control; condición; operación) {
 $$instrucciones
}
```

Ejemplo

```
for (var a :=: 0; a < 10; a++) {
 $$instrucciones
}
Var a;
for (a :=: 0; a < 10; a--) {
 $$instrucciones
}
```

5.3.11 Imprimir

Esta sentencia recibirá un valor de cualquier tipo de dato y lo mostrará en la consola del micro navegador como una salida.

Sintaxis

```
Print ( Expresion );
```

Ejemplo

```
Print ( “el texto que quiero mostrar”);
Print (‘a’);
Print (5);
Print (Variable);
Print (true);
```

5.3.12 Funciones nativas de integración DASM

El lenguaje de dracoScript admite un subconjunto de funciones las cuales realizarán la acción al momento de hacer una referencia o un llamado hacia la instancia.

Al momento de realizar importaciones de archivos ‘dasm’, deberá de encontrarse dentro del mismo proyecto, las importaciones se realizarán por el nombre del archivo.

Importar DASM

La función principal es poder hacer el llamado al archivo DASM (código intermedio, generado del lenguaje D++) e interpretar el código intermedio dentro del archivo DASM.

Esta función importará el o los archivos DASM e interpretará, se ejecutará un archivo a la vez, es decir se ejecutar el primer archivo al finalizar se ejecutará el siguiente y así sucesivamente con los demás archivos.

Sintaxis

```
RunMultDasm (<<archivo1>>,<<archivo2>>,<<archivo3>>, <<archivoN>> …);
```

Ejemplo

```
RunMultDasm ('archivo.dasm', 'archivo2.dasm', 'archivo3.dasm'); $$Ejecutara el código de cada archivo.
```

5.3.13 Funciones nativas de dibujo

Pintar punto

Este método tomará una serie de parámetros con los cuales podrá dibujar un punto en una posición en específica del área del micro navegador. El punto a dibujar la posición [X,Y] deberá ser el centro del punto.

Sintaxis

```
Point (posición en X, posición en Y, color, Diámetro);
```

- El primer parámetro deberá ser de tipo entero e indicará la posición en X del área del micro navegador donde se pintará el puto.
- El segundo parámetro deberá ser de tipo entero e indicará la posición en Y del área del micro navegador donde se pintará el punto.
- El tercer parámetro deberá ser de tipo cadena e indicará el color del punto. dicho parámetro será ingresado en hexadecimal, ejemplo "#FFFFFF", "#FFCCEE", etc.
- El cuarto parámetro deberá ser de tipo entero e indicará el diámetro medido en pixeles de punto.

Ejemplo

```
point (100,100, "#000000", 50);
point (50,105, "#FFFF", 50);
```

Pintar Cuadrado

Este método tomará una serie de parámetros con los cuales podrá dibujar un cuadrado es decir con altura y anchura con tamaños distintos en una posición en específica del área del micro navegador. La posición [X, Y] será la esquina superior izquierda del cuadrado y dibujará hacia la derecha y hacia abajo, siendo estas su ancho y altura.

Sintaxis

```
Quadrate (posición en X, posición en Y, color, Ancho, Alto)
```

- El primer parámetro deberá ser de tipo entero e indicará la posición en X del área del micro navegador donde se pintará el rectángulo.
- El segundo parámetro deberá ser de tipo entero e indicará la posición en Y del área del micro navegador donde se pintará el rectángulo.

- El tercer parámetro deberá ser de tipo cadena he indicará el color del rectángulo. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.
- El cuarto parámetro deberá ser de tipo entero e indicará el ancho medido en pixeles del rectángulo.
- El quinto parámetro deberá ser de tipo entero e indicará el alto medido en pixeles del rectángulo.

Ejemplo

```
Quadrate (100, 500, “#F44000”, 150, 200)
```

Pintar Ovalo

Este método tomará una serie de parámetros con los cuales podrá dibujar un ovalo es decir con altura y anchura con tamaños distintos en una posición en específica del área del micro navegador.

Sintaxis

```
Oval (posición en X, posición en Y, color, Ancho, Alto)
```

- El primer parámetro deberá ser de tipo entero e indicará la posición en X del área del micro navegador donde se pintará el óvalo.
- El segundo parámetro deberá ser de tipo entero e indicará la posición en Y del área del micro navegador donde se pintará el óvalo.
- El tercer parámetro deberá ser de tipo cadena e indicará el color del óvalo. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.
- El cuarto parámetro deberá ser de tipo entero e indicará el ancho medido en pixeles del óvalo.
- El quinto parámetro deberá ser de tipo entero e indicará el alto medido en pixeles del óvalo.

Ejemplo

```
Oval (100, 500, “#F44000”, 150, 200)
```

Pintar cadena

Este método tomará una serie de parámetros con los cuales podrá dibujar una cadena en una posición en específica del área del micro navegador.

Sintaxis

```
String (posición en X, posición en Y, color, Cadena)
```

- El primer parámetro deberá ser de tipo entero he indicará la posición en X del área del micro navegador donde se pintará el puto.
- El segundo parámetro deberá ser de tipo entero he indicará la posición en Y del área del micro navegador donde se pintará el punto.
- El tercer parámetro deberá ser de tipo cadena he indicará el color del punto. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.
- El cuarto parámetro deberá ser de tipo cadena y será el texto a desplegar en el área del micro navegador.

Nota: El grosor del texto queda a discreción del estudiante, pero deberá se legible.

Ejemplo

```
String (100, 500, “#F44000”, “Hola mundo”);
```

Pintar línea

Este método tomará una serie de parámetros con los cuales podrá dibujar una línea en una posición en específica del área del micro navegador.

Sintaxis

```
Line (posición en Xi, posición en Yi, posición Xf, posición Yf, color, grosor )
```

- El primer parámetro deberá ser de tipo entero e indicará la posición en Xi del área del micro navegador donde iniciar a pintar la línea.
- El segundo parámetro deberá ser de tipo entero e indicará la posición en Yi del área del micro navegador donde iniciar a pintar la línea.
- El tercer parámetro deberá ser de tipo entero e indicará la posición en Xf del área del micro navegador donde finalizará de pintar la línea.
- El cuarto parámetro deberá ser de tipo entero e indicará la posición en Yi del área del micro navegador donde finalizará de pintar la línea.
- El quinto parámetro deberá ser de tipo cadena e indicará el color de la línea. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.
- El sexto parámetro deberá ser de tipo entero e indicará el grosor de la línea medido en pixeles.

Ejemplo

```
Line  (100, 500, 150, 200, “#F44000”, 15);
```

Nota: El color de usado en todas las funciones nativas es el color de relleno de la figura.

6 Generación código intermedio Draco Assembler

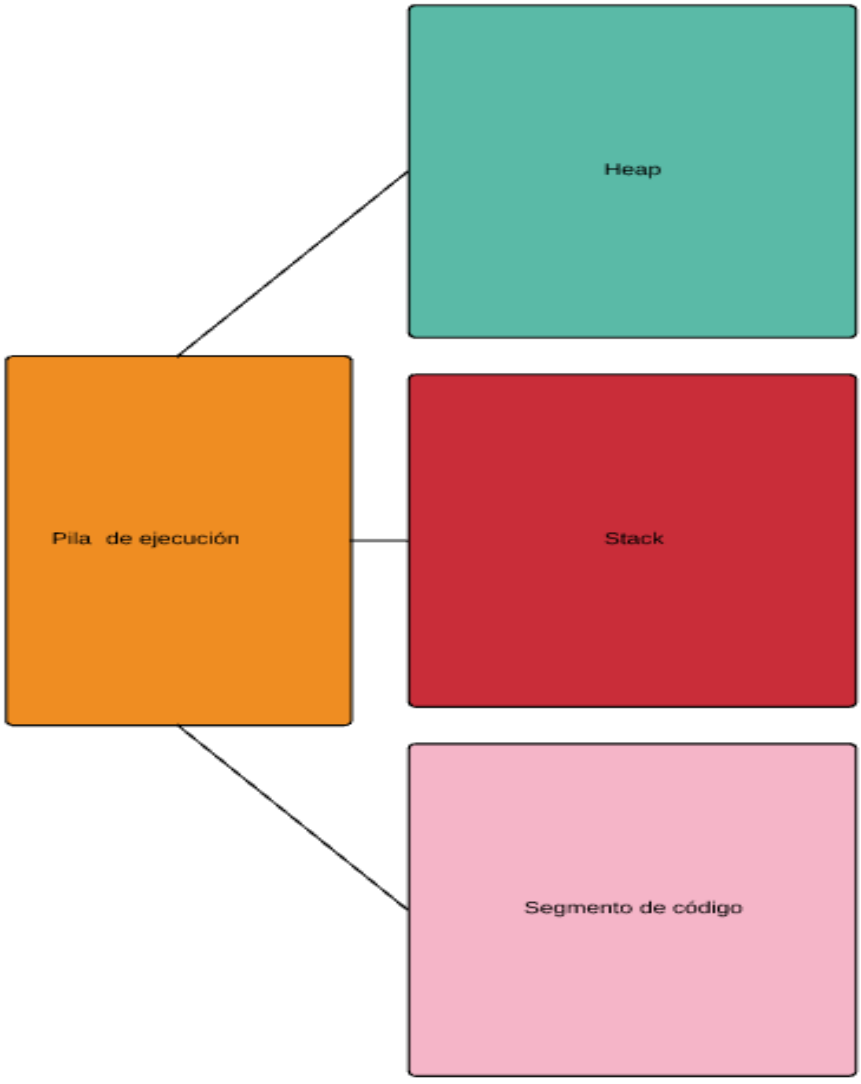
DASM será un formato de código seguro, portátil y de bajo nivel diseñado para una ejecución eficiente y una representación compacta. Su objetivo principal es habilitar aplicaciones de alto rendimiento en la Web.

6.1 Caracteristicas de DASM

6.2 Estructuras en tiempo de ejecución

Para ejecutar el código intermedio que se generará se contará con una pila que tendrá una sección de código, una sección para variables locales (stack) y una sección para almacenar cadenas y estructuras de datos con tamaños dinámicos (heap).

En la siguiente imagen se define la estructura de la pila:

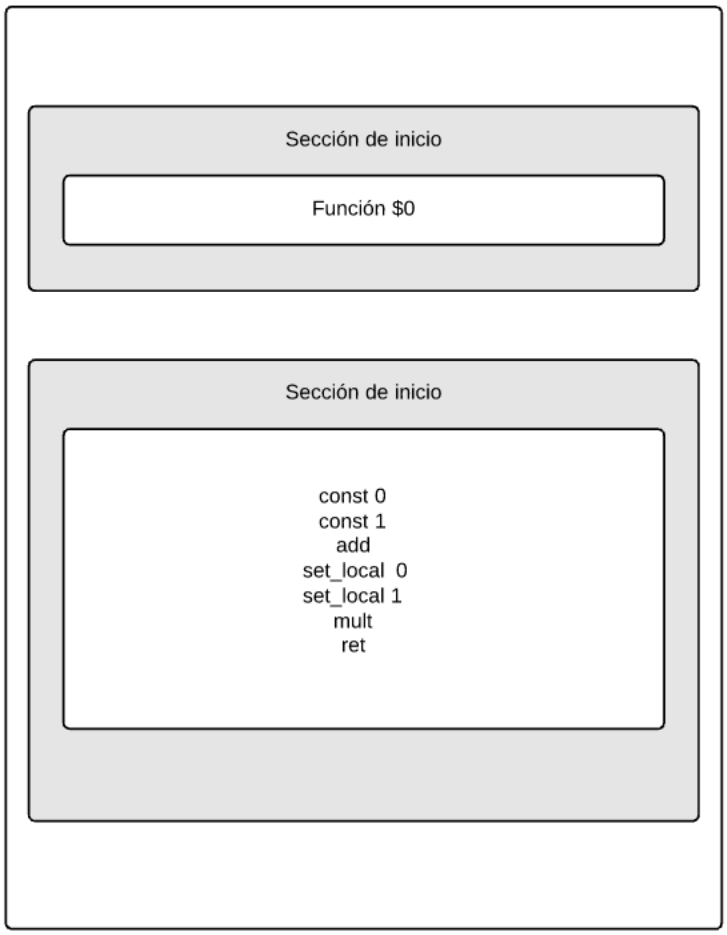


6.2.1 Segmento de código

En esta sección de la pila estarán almacenas todas las instrucciones de código intermedio que se ejecutarán. Esta área funcionará como una máquina de pila por lo que simulará la memoria de la computadora como un stack. EL código creado contará con las siguientes secciones:

- Sección de inicio: en esta sección se tendrá el nombre de la función.
- Sección de código: En esta área estará todo código que se haya generado para la función.

En la siguiente grafica se detalla sus secciones:



6.2.1.1 Set instrucciones DASM

El bytecode DASM contará con un conjunto de instrucciones que se utilizarán para operar los valores que se encuentren en el área código.

Instrucciones aritméticas

- Add: esta instrucción tomará los 2 primeros valores que se encuentren en el área de código los sacará (pop), los sumará y luego introducirá el resultado en la cima del segmento de código.

Sintaxis

```
Valor1
Valor2
Add
```

Ejemplo

```
5
2
Add
```

- Diff: esta instrucción tomará los 2 primeros valores que se encuentren en el área de código los sacará (pop), los restará y luego introducirá el resultado en la cima del segmento de código.

Sintaxis

```
Valor1
Valor2
Diff
```

Ejemplo

```
5
2
Diff
```

- Mult: esta instrucción tomará los 2 primeros valores que se encuentren en el área de código los sacará (pop), los multiplicará y luego introducirá el resultado en la cima del segmento de código.

Sintaxis

```
Valor1
Valor2
Mult
```

Ejemplo

```
5
2
Mult
```

- Div: esta instrucción tomará los 2 primeros valores que se encuentren en el área de código los sacará (pop), los dividirá y luego introducirá el resultado en la cima del segmento de código.

Sintaxis

```
Valor1
Valor2
Div
```

Ejemplo

```
5
2
Div
```

- Mod: esta instrucción tomará los 2 primeros valores que se encuentren en el área de código los sacará (pop), y luego introducirá el resultado en la cima del segmento de código.

Sintaxis

```
Valor1
Valor2
Mod
```

Ejemplo

```
5
2
Mod
```

Instrucciones relacionales

- Lt: esta instrucción tomará los 2 primeros valores que se encuentren en el área de código los sacará (pop), verificará si el valor en la posición 0 es menor que el valor que se encuentra en la posición 1, si el resultado fuera verdadero colocará 1 en la cima de lo contrario colocará 0.

Sintaxis

```
Valor1
Valor2
Lt
```

Ejemplo

```
5
2
Lt
```

- Gt: esta instrucción tomará los 2 primeros valores que se encuentren en el área de código los sacará (pop), verificará si el valor en la posición 0 es mayor que el valor que se encuentra en la posición 1, si el resultado fuera verdadero colocará 1 en la cima de lo contrario colocará 0.

Sintaxis

```
Valor1
Valor2
Gt
```

Ejemplo

```
5
2
Gt
```

- Lte: esta instrucción tomará los 2 primeros valores que se encuentren en el área de código los sacará (pop), verificará si el valor en la posición 0 es menor o igual que el valor que se encuentra en la posición 1, si el resultado fuera verdadero colocará 1 en la cima de lo contrario colocará 0.

Sintaxis

```
Valor1
Valor2
Lte
```

Ejemplo

```
4
5
Lte
```

- Gte: esta instrucción tomará los 2 primeros valores que se encuentren en el área de código los sacará (pop), verificará si el valor en la posición 0 es mayor o igual que el valor que se encuentra en la posición 1, si el resultado fuera verdadero colocará 1 en la cima de lo contrario colocará 0.

Sintaxis

```
Valor1
Valor2
Gte
```

Ejemplo

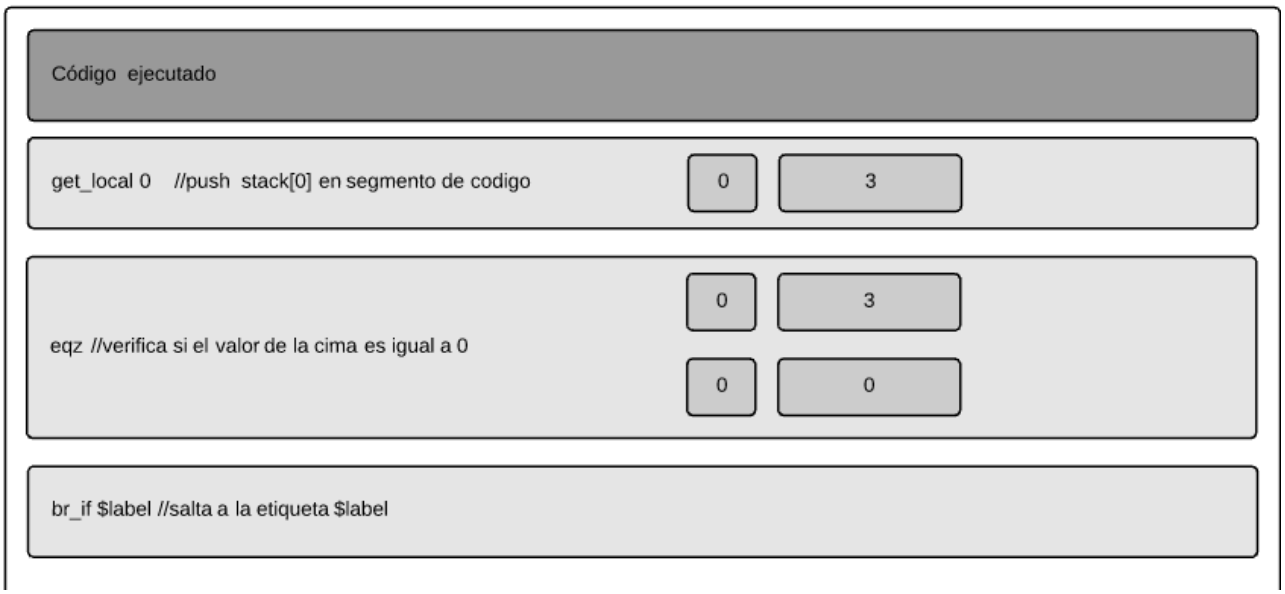
```
5
5
Gte
```

- Eqz: esta instrucción verificará si el contenido de la primera posición en el área de código es igual a 0, sacará el valor de la primera posición del segmento de código y meterá 1 si el valor era igual a 0, de lo contrario colocará un 0.

Instrucciones lógicas

- And: esta instrucción ejecutará una instrucción and bit por bit a los primeros dos valores que se encuentren en la cima, los sacará y hará un push del resultado.
- Or: esta instrucción ejecutará una instrucción or bit por bit a los primeros dos valores que se encuentren en la cima, los sacará y hará un push del resultado.
- Not: esta instrucción ejecutará una instrucción not bit por bit al valor que se encuentre en la cima, lo sacará y hará un push del resultado.
- Br: esta instrucción saltará a la etiqueta que se indique.
- Br_if: saltará a la etiqueta que se indique, si el valor de la cima es igual a 0

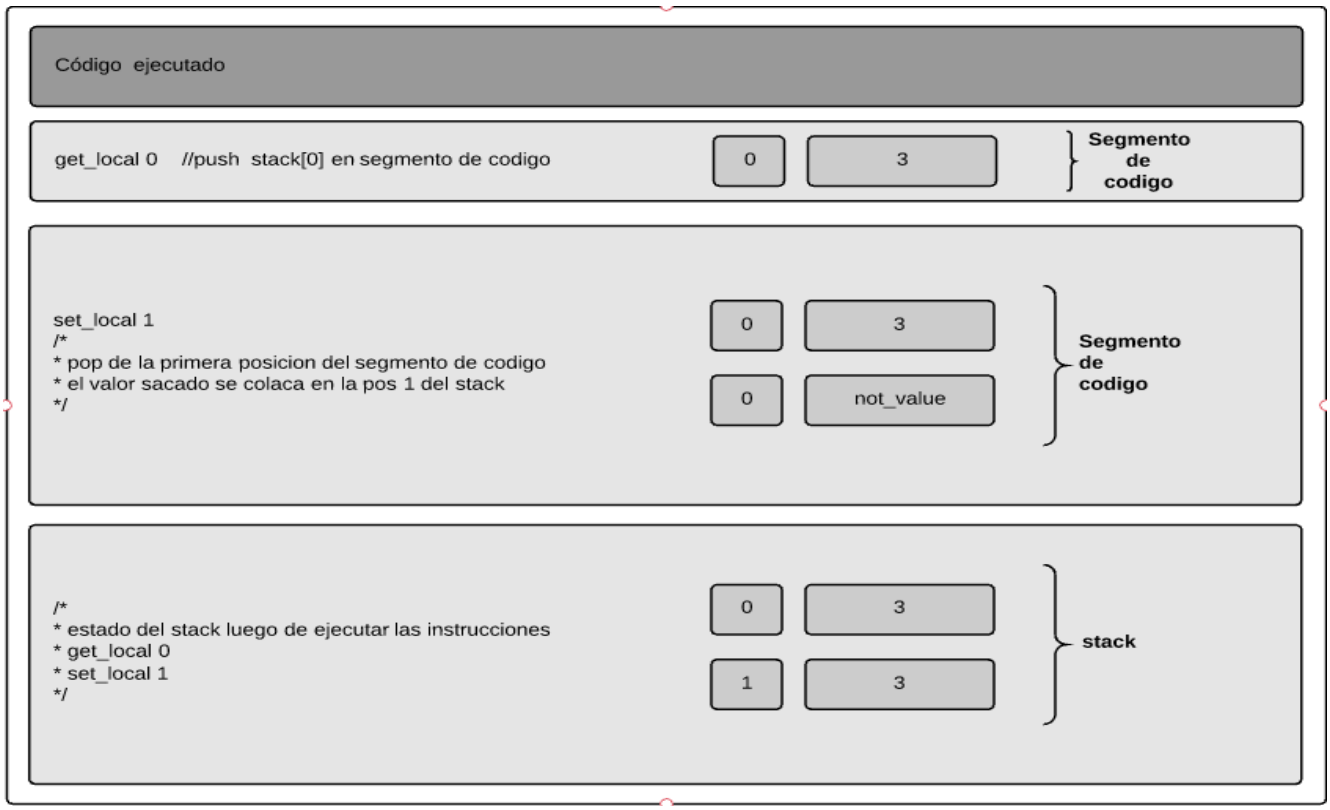
Ejemplo

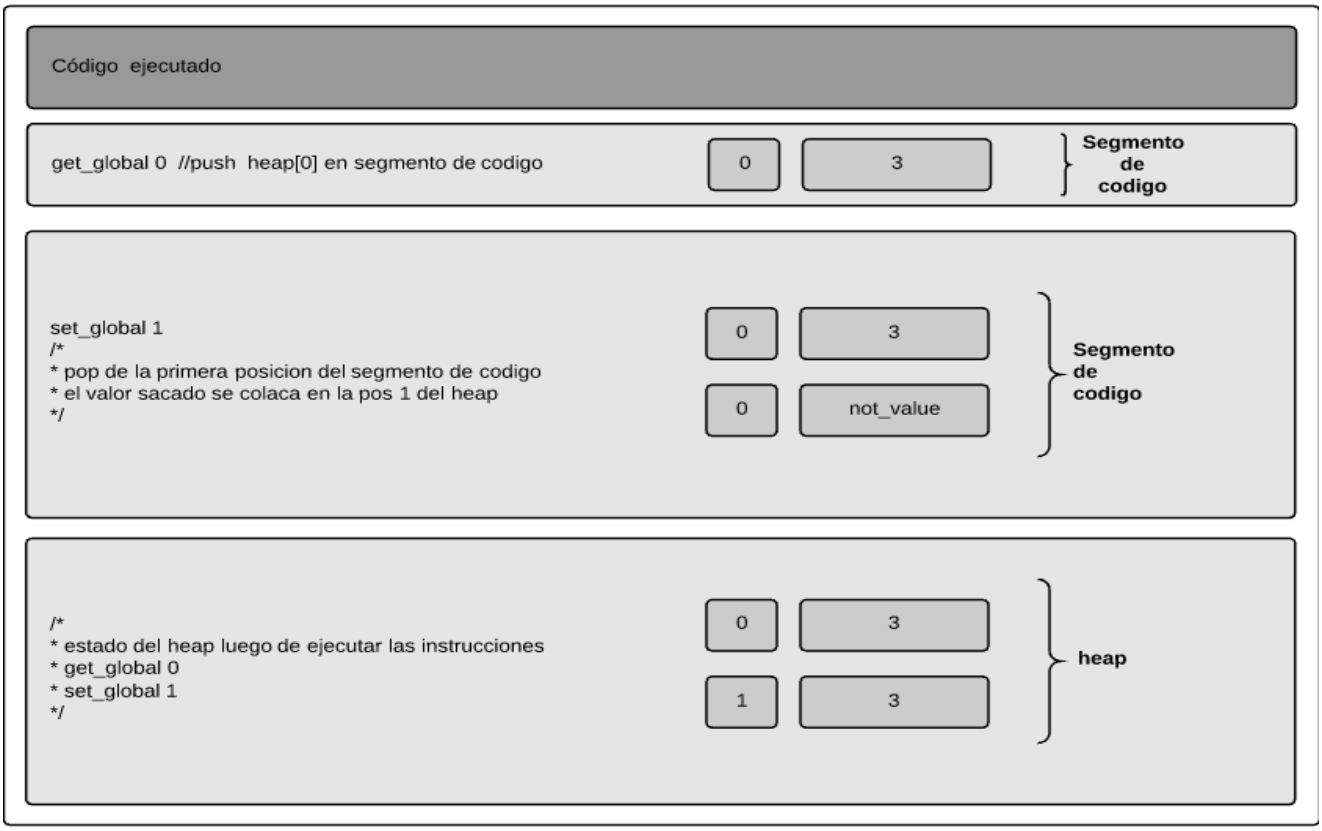


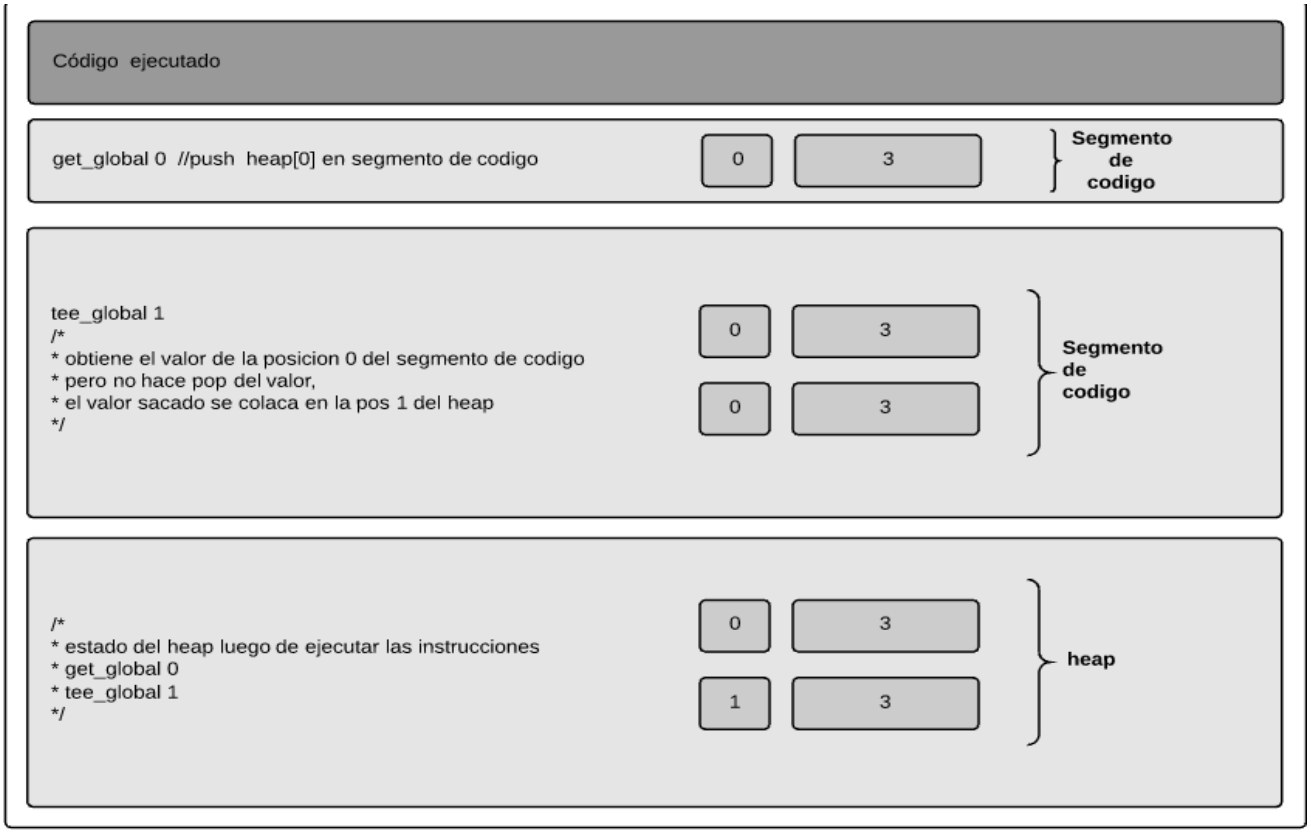
Instrucciones para el manejo de memoria

- Get_local: esta instrucción obtendrá el valor de la dirección de memoria que se indique y luego lo meterá al inicio del segmento de código. La dirección de memoria que se use hará referencia a una posición dentro del stack.
- Get_global: esta instrucción obtendrá el valor de la dirección de memoria que se indique y luego lo meterá al inicio del segmento de código. La dirección de memoria que se use hará referencia a una posición dentro del heap.
- Set_global: esta instrucción obtendrá el valor de la cima del segmento de código y lo almacenará en la dirección de memoria que se indique. La dirección de memoria que se use hará referencia a una posición dentro del heap. La instrucción hará pop del contenido de la cima del segmento código.
- Set_local: esta instrucción obtendrá el valor de la cima del segmento de código y lo almacenará en la dirección de memoria que se indique. La dirección de memoria que se use hará referencia a una posición dentro del stack. La instrucción hará pop del contenido de la cima del segmento código.
- Tee_local: esta instrucción obtendrá el valor de la cima del segmento de código y lo almacenará en la dirección de memoria que se indique. La dirección de memoria que se use hará referencia a una posición dentro del stack. La instrucción no hará pop del contenido de la cima del segmento código.
- Tee_global: esta instrucción obtendrá el valor de la cima del segmento de código y lo almacenará en la dirección de memoria que se indique. La dirección de memoria que se use hará referencia a una posición dentro del heap. La instrucción no hará pop del contenido de la cima del segmento código.

Ejemplos


Código ejecutado
get_local 0 //push stack[0] en segmento de codigo
0
3
Segmento de codigo
set_local 1
/*
* pop de la primera posicion del segmento de codigo
* el valor sacado se colaca en la pos 1 del stack
*/
0
3
0
not_value
Segmento de codigo
/*
* estado del stack luego de ejecutar las instrucciones
* get_local 0
* set_local 1
*/
0
3
1
3
stack


Código ejecutado
get_global 0 //push heap[0] en segmento de codigo
0
3
Segmento de codigo
set_global 1
/*
* pop de la primera posicion del segmento de codigo
* el valor sacado se colaca en la pos 1 del heap
*/
0
3
0
not_value
Segmento de codigo
/*
* estado del heap luego de ejecutar las instrucciones
* get_global 0
* set_global 1
*/
0
3
1
3
heap



Procedimientos

Los métodos podrán ser declarados de la siguiente forma.

Sintaxis

```
Function <<nombre del metodo >>
        ..
        ..
        ..
        ..
End
```

Ejemplo

```
Function $ejemplo
        5
        10
        add
End
```

Print

Esta sentencia recibirá un parámetro de tipo cadena y un identificador que tendrá el valor de lo que se mostrará en la consola de salida. Los valores que tomarán $calc y $ret deberán de ser calculados por el estudiante, la forma que decida hacerlo quedará a su criterio.e

Sintaxis

```
F//sera el formato, entero
V//sera el valor, entero
print
```

En la siguiente tabla se muestran los parámetros permitidos para esta sentencia.

Parámetro	Acción
“%c”	Imprime el valor carácter del identificador.
“%d”	Imprime el valor entero del identificador.
“%f”	Imprime el valor con punto flotante del identificador.

Ejemplo

```
“%c”
65      //Salida A
Print
```

Función Calcular

Calcular es una dirección de memoria que debe ser calculada por el estudiante y queda a su criterio para poder realizarlo. Esta utilizará la palabra reservada “**calc**”, y deberá ser llamada utilizada anteponiendo el símbolo “$”.

Sintaxis

```
$calc
```

Ejemplo

```
get_local $calc
```

Función Retorno

Retorno es la dirección de memoria en donde se decida almacenar el retorno de una función esta queda a discreción del estudiante para realizarlo. Es utilizará la palabra reservada "**ret**" y deberá ser llamada utilizada anteponiendo el símbolo "$".

Sintaxis

```
$calc
```

Ejemplo

```
get_local $calc
```

Funciones nativas

Para cada función nativa de los lenguajes Dracospcrit y D++ existe una versión en DASM. Para llamar una función nativa en DASM basta con escribir su nombre y su lista de parámetros. Los valores que tomarán $calc y $ret deberán de ser calculados por el estudiante, la forma que decida hacerlo quedará a su criterio.

Pintar punto

Esta función se representará de la siguiente manera en bajo nivel:

Sintaxis

```
X //entero
Set_local $calc
Y//entero
Set_local $calc
color
Set_local $calc
diametro
Set_local $calc
Call $Point
```

- El valor de X deberá ser de tipo entero.

- El valor de Y deberá ser de tipo entero.
- Color será un entero en hexadecimal, a diferencia del código en algo nivel que será una cadena.
- El diámetro deberá ser de tipo entero.

Ejemplo

```
100
Set_local $calc
100
Set_local $calc
16007168
Set_local $calc
 50
Set_local $calc
Call $point
```

Pintar Cuadrado

Esta función nativa se representará de la siguiente manera:

Sintaxis

```
X //entero
Set_local $calc
Y//entero
Set_local $calc
color
Set_local $calc
ancho
Set_local $calc
alto
Set_local $calc
Call $Quadrate
```

- El valor de X deberá ser de tipo entero.
- El valor de Y deberá ser de tipo entero.
- Color será un entero en hexadecimal, a diferencia del código en algo nivel que será una cadena.
- El valor del Ancho será un entero.
- El valor del Alto será un entero.

Ejemplo

```
100
Set_local $calc
100
Set_local $calc
16007168
Set_local $calc
 50
Set_local $calc
 500
Set_local $calc
Call $Quadrate
```

Este valor “16007168” es equivalente a “#F44000”;

Pintar Ovalo

Esta función nativa se representará de la siguiente manera:

Sintaxis

```
X //entero
Set_local $calc
Y//entero
Set_local $calc
color
Set_local $calc
ancho
```

```
Set_local $calc
alto
Set_local $calc
Call $Quadrate
```

- El valor de X deberá ser de tipo entero.
- El valor de Y deberá ser de tipo entero.
- Color será un entero en hexadecimal, a diferencia del código en algo nivel que será una cadena.
- El valor del Ancho será un entero.
- El valor del Alto será un entero.

Ejemplo

```
100
Set_local $calc
100
Set_local $calc
16007168
Set_local $calc
 50
Set_local $calc
 500
Set_local $calc
Call $oval
```

Pintar cadena

Esta función nativa se representará de la siguiente manera:

Sintaxis

```
X //entero
Set_local $calc
Y//entero
Set_local $calc
```

```
color
Set_local $calc
PtrInicioCadena //inicio de la cadena, entero
Set_local $calc
Call $string
```

- El valor de X deberá ser de tipo entero.
- El valor de Y deberá ser de tipo entero.
- Color será un entero en hexadecimal, a diferencia del código en algo nivel que será una cadena.
- El cuarto parámetro deberá ser de tipo cadena y será el texto a desplegar en el área del micro navegador.

Ejemplo

```
100
Set_local $calc
100
Set_local $calc
16007168
Set_local $calc
 50
Set_local $calc
Call $point
```

Pintar línea

Esta función nativa se representará de la siguiente manera:

Sintaxis

```
Xi //entero
Set_local $calc
Yi//entero
Set_local $calc
Xf //entero
```

```
Set_local $calc

Yf//entero

Set_local $calc

Color //entero

Set_local $calc

Grosor //entero

Set_local $calc

Call $line
```

- El valor de Xi deberá ser de tipo entero.
- El valor de Yi deberá ser de tipo entero.
- El valor de Xf deberá ser de tipo entero.
- El valor de Yf deberá ser de tipo entero.
- Color será un entero en hexadecimal, a diferencia del código en algo nivel que será una cadena.
- El valor de grosor deberá ser de tipo entero.

Ejemplo

```
Line  (100, 500, 150, 200,16007168, 15);
100 //entero
Set_local $calc

500//entero

Set_local $calc

150//entero
Set_local $calc

200//entero

Set_local $calc

16007168//entero

Set_local $calc

15//entero

Set_local $calc

Call $line
```

Llamadas a procedimientos

Las llamadas a procedimientos se harán con la instrucción Call seguido del nombre del método.

```
call $name //ejemplo de llamada a una función
```

6.2.2 Stack

El stack se usará para almacenar parámetros de funciones y variables locales.

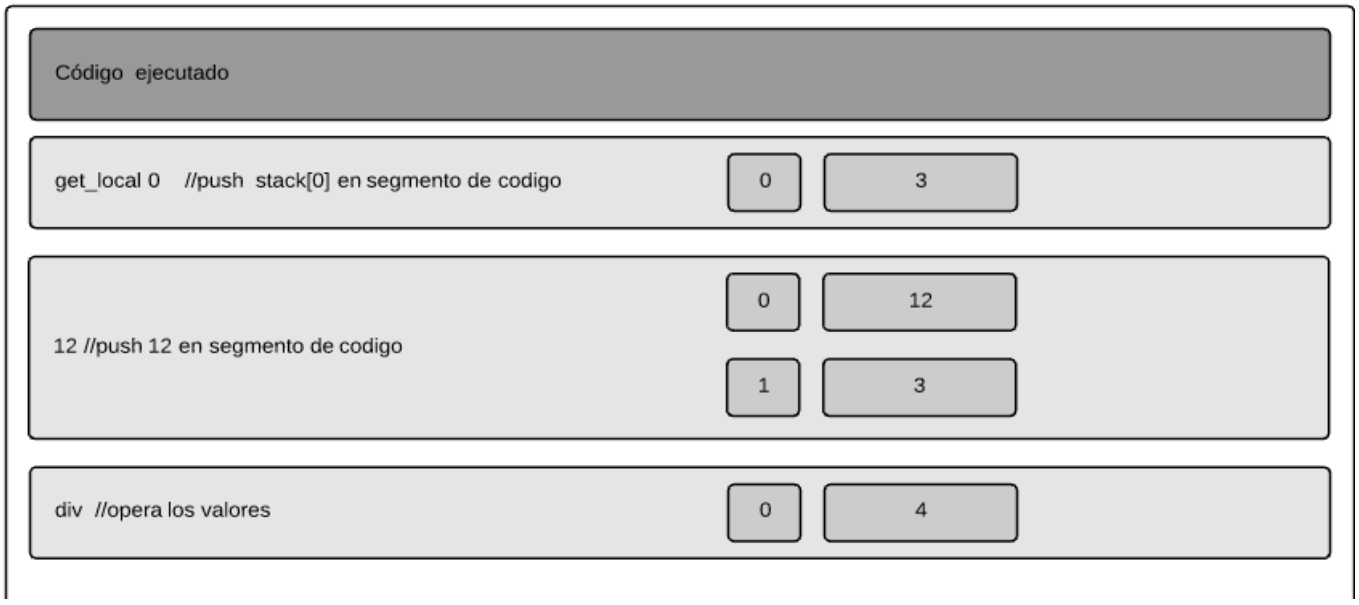
6.2.3 Heap

El heap representará un segmento de memoria que se usará para almacenar estructuras de datos y cadenas. Este espacio de memoria crecerá para adaptarse al programa que se esté ejecutando.

6.2.4 Ejemplo de ejecución

```
function $name
    get_local 0  //obtiene el valor de la pos 0 del stack y lo mete al inicio del segmento de código
    12  //mete el valor 12 al inicio del segmento de código
    div //saca los dos primeros valores y los divide, mete el resultado en el segmento de código
end //fin de la función
```

En la siguiente imagen se observará el comportamiento del segmento de código luego de ejecutar la función **$name**



7 Ejemplo de Compilación

Para mostrar la funcionalidad deseada de **Draco Ensamblado Web** se desarrolló una aplicación completa utilizando los lenguajes **DracoScript** y **D++** con el respectivo código **DASM** generado.

El código podrá encontrarse accediendo al siguiente enlace:

Ejemplo de compilación D++ a DASM

La funcionalidad desarrollada dibuja una flor y además contiene un ejemplo de una función recursiva, la elaboración de esta flor se realizó haciendo uso de las sentencias básicas que el lenguaje **DracoScript** provee para dibujar una parte de la flor, para completar dicho dibujo se hizo uso del lenguaje **D++** y su código equivalente en **DASM**.

El código **DracoScript** está contenido en el archivo que lleva por nombre:

Dibujar_Flor – DracoScript.djs

Al interpretar el archivo con código de **DracoScript** realizará el dibujo de 4 líneas, el marco de la flor, dicha funcionalidad tiene un grado de complejidad bajo.

Luego se realiza la integración con las funcionalidades contenidas en el archivo **DASM**, siendo este el código generado por el compilador, este realizará los pétalos y las hojas de la flor dicha funcionalidad tiene un grado de complejidad alto.

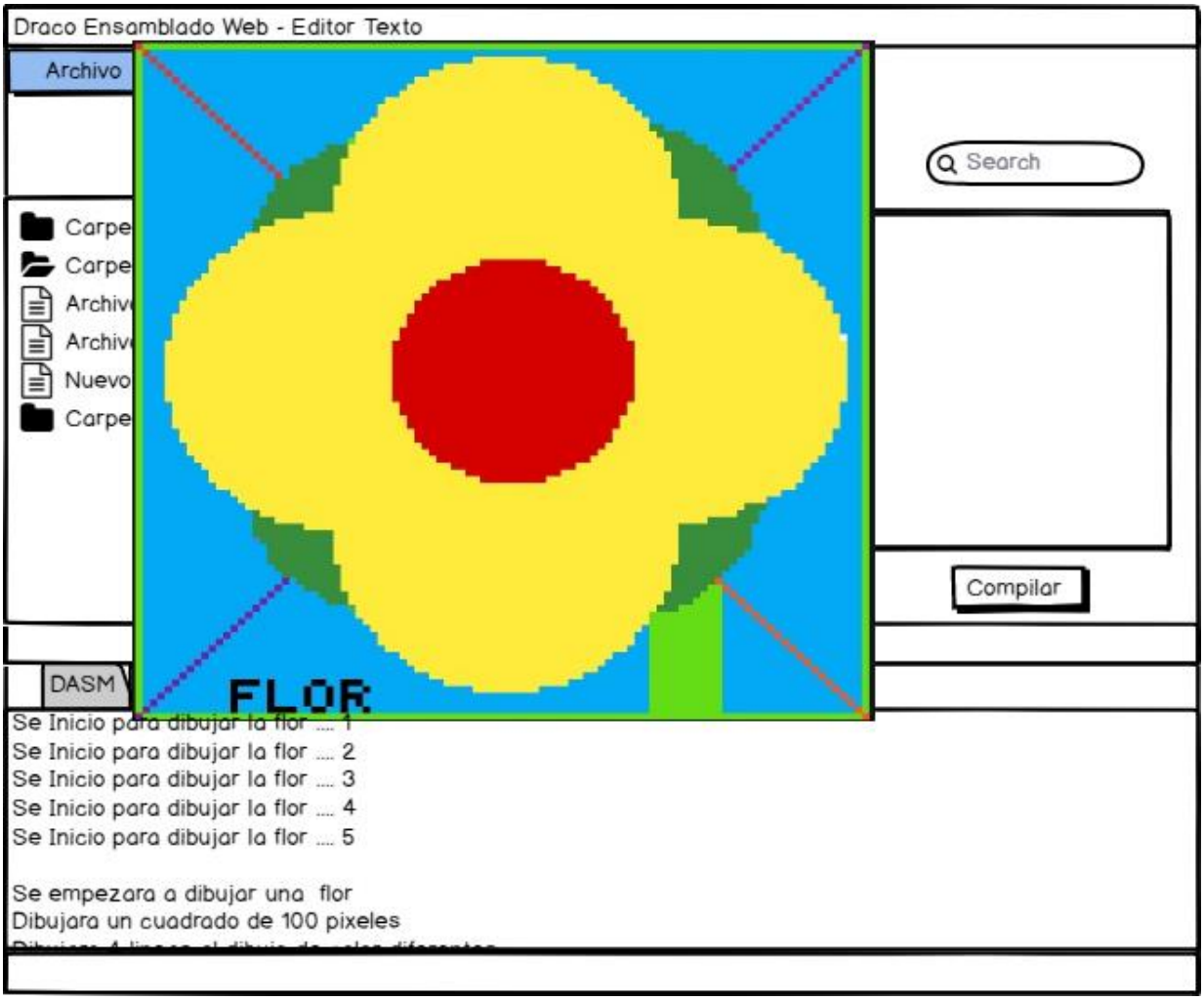
El código de alto nivel se encuentra en el archivo siguiente de la carpeta compartida:

Dibujar_Flor – Alto Nivel.dpp

Al compilar el archivo **D++** se deberá generar el código contenido en el archivo que lleva por nombre:

Dibujar_Flor.dasm

Se adjunta la salida esperada de la aplicación desarrollada en **Draco Ensamblado web**:



8 Tabla de símbolos

La tabla de símbolos es un reporte que se mostrará en el micro navegador y contiene una estructura de datos donde mostrará cada símbolo perteneciente al lenguaje con información relevante de este. La opción se encontrará en el editor, y esta mostrará la tabla de símbolos de la última ejecución realizada por la herramienta.

La tabla de símbolos se mostrará como un reporte luego de haber compilado el lenguaje D++ en la aplicación. Este reporte deberá contener el día y hora de la ejecución, seguido de la tabla que contendrá el detalle de los símbolos relevantes del programa de entrada.

La tabla de símbolos deberá contener como mínimo la siguiente información:

- Nombre o Identificador del símbolo

- Tipo de dato del símbolo: mostrará el tipo de dato del atributo, si es una función mostrará el tipo de dato que retorna.
- Rol: definirá el rol que maneja, es decir, si es un atributo, función, parámetro, etc.
- Ámbito: Mostrará el ámbito donde se encuentra, si es ámbito local, global.
- Posición: mostrará la posición en donde se encuentre en el ámbito.
- Tamaño.

Ejemplo

Ámbito	Nombre	Tipo	Posición	Tamaño	Rol
Global	Principal	void	------	12	Principal
Local	Retorno_Principal	void	0	1	Retorno

9 Manejo de Errores

La herramienta deberá ser capaz de identificar todo tipo de errores (léxicos, sintácticos y semánticos) al momento de interpretar lenguaje de entrada (D++, DracoScript). Estos errores se almacenarán y se mostrarán en la aplicación. Este reporte deberá contener el día y hora de ejecución, seguido de una tabla que contendrá la descripción detallada de cada uno de los errores.

Lo tipos de errores que se deberán de manejar son los siguientes:

- Errores léxicos:
- Errores sintácticos.
- Errores semánticos.

Un **error léxico** es aquel en el que se encuentra un carácter que no está permitido como parte de un lexema.

Ejemplo 1:

```
ente$ro numero = 15;
```

En la cadena anterior se introdujo arbitrariamente el componente léxico "$", esto deberá de provocar un error léxico. La forma en que se recuperará de un error léxico es con la estrategia del modo de pánico, por lo que se ignorarán los siguientes caracteres hasta encontrar un token bien formado. En el ejemplo anterior se ignorarán los siguientes caracteres "ente$ro" hasta terminar de formar el token "numero".

Los **errores sintácticos** son aquellos que darán como resultado de ingresar una cadena de entrada que no corresponde con la sintaxis definida para el lenguaje.

Ejemplo 2:

```
numero entero = 10;
```

En la entrada del ejemplo anterior se invirtieron el orden de los tokens que se usan para formar una declaración de una variable (ver ejemplo 3). Al encontrarse con un error sintáctico la estrategia que se tomará será la de terminar el proceso de análisis y reportar el error en ese mismo instante.

Ejemplo 3:

```
entero numero = 10;
```

En el ejemplo anterior se muestra el formato correcto en el que se declarará una variable.

Un **error semántico** se dará por ejemplo de intentar usar una variable que no ha sido declarada.

Ejemplo 4:

```
…
entero numero1 = numero2 + 10;
...
```

Como se ve en ejemplo 4 la variable "numero2" no ha sido declarada por lo que se mostrará un error semántico, la forma en que se manejaran los errores semánticos consistirá en parar el análisis y mostrar un mensaje de error.

Consideraciones

- Si se existiesen varios errores léxicos se podrá continuar con el análisis de la cadena de entrada y los errores detectados se mostrarán al final del análisis.
- Si se encontrara algún un error sintáctico en la cadena de entrada se deberá de reportar en ese instante y pararse el proceso de análisis.
- Si se hallará un error de tipo semántico se procederá a parar el proceso de análisis y se reportará el error de inmediato.

La tabla de errores debe contener como mínimo la siguiente información:

- Línea: Número de línea donde se encuentra el error.
- Columna: Número de columna donde se encuentra el error.
- Tipo de error: Identifica el tipo de error encontrado. Este puede ser léxico, sintáctico o semántico.
- Descripción del error: Dar una explicación concisa de por qué se generó el error.

Ejemplo

Tipo	Descripción	Fila	Columna
Léxico	El caracter '@' no pertenece al alfabeto del lenguaje	3	1
Sintáctico	Se esperaba <id> y se encontró "+"	85	15
Semántico	No es posible asignar <str> a la variable 'x' de tipo num	120	2

10 Requisitos Mínimos

Para que la entrega y calificación del primer proyecto sea efectiva, la solución entregada deberá contener los siguientes requisitos mínimos:

- Draco Compiler
  - IDE
  - Editor de Texto
  - Consola
  - Tabla de símbolos
  - Errores
- Micro-Navegador
- Lenguaje interpretado DracoScript
  - Características del lenguaje
  - Variables
  - Declaración de variable
  - Asignación de variables
  - Tipos de datos
  - Operando
  - Condiciones
  - Expresiones relacionales
  - Expresiones lógicas
  - Sentencias de control
  - Bucles
    - For
  - Imprimir
  - Funciones nativas de integración
  - Funciones nativas de dibujo
    - Puntos
    - Cuadrados
    - Cadenas
- Lenguaje de alto nivel D++
  - Case sensitive
  - Recursividad
  - Variables
  - Tipos de datos

    - Operaciones aritméticas
    - Operaciones relacionales
    - Operaciones lógicas
    - Declaración de variables
    - Asignación de variables
    - Arreglos de una dimensión
        - Asignación
        - Declaración e inicialización
    - Sentencias de selección
        - Si
    - Sentencias cíclicas
        - Mientras
        - Para
    - Sentencias de transferencia
        - Detener
        - Retornar
    - Estructuras
    - Acceso a los atributos de estructuras
    - Métodos y Funciones
        - Llamadas de métodos y funciones
    - Método principal
    - Funciones nativas de dibujo
        - Punto
        - Cuadrado
        - Cadena
- Generación de código intermedio DASM (Todo)

11 Entregables y Restricciones

11.1 Entregables

- Código fuente de las dos aplicaciones.
- La aplicación deberá ser funcional.
- Ejecutables de las dos aplicaciones.
- Gramáticas de los 3 lenguajes
- Manual Técnico
- Manual de Usuario.

11.2 Criterios de Entrega

- La forma de entrega será virtual y en los días previos a la fecha de entrega se publicará un enlace web en el cual los estudiantes deberán subir su proyecto, el contenido a subir deberá ser todos los entregables mencionados en el enunciado, comprimidos en formato zip.

- El desarrollo del proyecto y la entrega del mismo es individual.
- El proyecto deberá ser desarrollado en los lenguajes y con las herramientas para la generación de analizadores descritos a continuación:
    - **La aplicación deberá ser desarrollada en el lenguaje java, los interpretes de DracoScript, DASM y el compilador para el lenguaje D++ deberán ser desarrollados con los analizadores generados JFlex y Cup.**
- No se recibirán proyectos después de la fecha y hora de entrega.
- Deberán entregarse todos los archivos necesarios para la ejecución de las aplicaciones, así como el código fuente de ambas aplicaciones, las gramáticas y la documentación.
- El estudiante es completa y únicamente responsable de verificar el contenido de los entregables.

11.3 Criterios de Calificación

- La calificación del proyecto se realizará presencialmente.
- La calificación del proyecto se realizará con los archivos entregados en la fecha establecida y desde los ejecutables.
- No se podrá agregar o quitar algún símbolo en los archivos de entrada. El proyecto deberá funcionar con los archivos que sean proveídos por los auxiliares para la calificación, sin modificación alguna.
- No será permitido compartir los archivos de entrada durante ni después de la calificación.
- La calificación del proyecto será personal y existirá un tiempo límite.
- Se debe tomar en cuenta que no pueden estar personas ajenas a la calificación, de lo contrario no se calificará el proyecto.
- Anomalías o copias detectadas en el proyecto tendrán de manera automática una nota de 0 puntos y los involucrados serán reportados a la Escuela de Ingeniería en Ciencias y Sistemas, para que se apliquen las sanciones correspondientes.
- Existirá horarios para la calificación de cada proyecto, por el cual el estudiante deberá de elegir el horario que mejor le convenga.
- Anomalías detectadas en los archivos entregables tendrá de manera automática una nota de 0 puntos, por ejemplo: no se envió código el código correcto, se envió parte del código y no el código completo, archivos ajenos a los entregables del proyecto, no se hizo uso de las herramientas descritas en el enunciado de cada proyecto, entre otras.
- Los archivos de entrada contendrán errores semánticos, sintácticos y léxicos para la verificación de recuperación de errores de la aplicación.

11.4 Restricciones

- **La aplicación deberá ser desarrollada en el lenguaje java, los interpretes de DracoScript, DASM y el compilador para el lenguaje D++ deberán ser desarrollados con los analizadores generados JFlex y Cup.**
- Copias de proyectos tendrán automáticamente nota de 0 puntos y se reportará a los involucrados a la Escuela de Ingeniería en Ciencias y Sistemas
- No se recibirán proyectos después de la fecha y hora de entrega.
- La calificación se realizará sobre archivos ejecutables.
- La entrega deberá de cumplir con los requisitos mínimos.

Fecha de entrega: viernes 9 de noviembre del 2018 hasta las 08:00 p.m.

El día de la entrega se publicarán los links donde deberán subir su proyecto.