Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Primer Semestre de 2017
Catedráticos:
Ing. Bayron López, Ing. Edgar Sabán
Tutores Académicos:
Enio González, Esvin González, Mario Asencio

Basic3D

Proyecto 2 – Un generador de código intermedio interactivo

1. Objetivos

Que el estudiante demuestre todos los conocimientos adquiridos en la clase magistral (gramáticas, definiciones dirigidas por la sintaxis, esquemas de traducción, código intermedio, optimización, entre otros) así como los conocimientos adquiridos en laboratorio (uso de herramientas para la generación de parsers y scanners, buenas prácticas de programación, etc.).

1.1. Objetivo general

Aplicar los conocimientos de Organización de Lenguajes y Compiladores 2 en la creación de una solución de software.

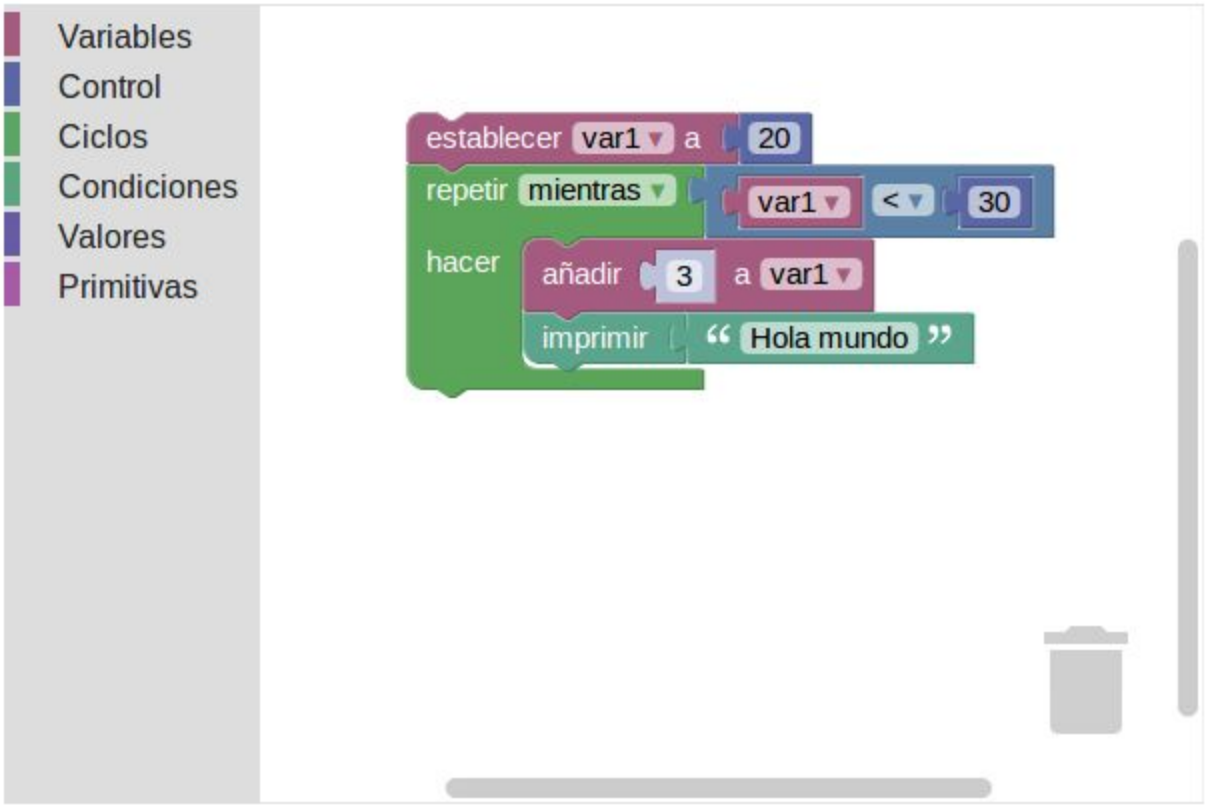
1.2. Objetivos específicos

- Utilizar herramientas específicas (en JavaScript) para la creación de un compilador que cumpla efectivamente con los requerimientos de un lenguaje predeterminado.
- Aplicar los conocimientos de generación de código intermedio en formato de tres direcciones en la implementación de una herramienta que permita generar dicho código a partir de un lenguaje de alto nivel.
- Aplicar los conocimientos del AST para implementar una herramienta que cumpla el papel de un debugger para el código de tres direcciones generado.
- Implementar una herramienta que realice la optimización detallada del código intermedio generado por medio de las técnicas de “mirilla” y “por bloque”.

2. Descripción del proyecto

El proyecto consiste en desarrollar una herramienta que sea capaz de compilar un lenguaje de alto nivel llamado Basic 3D, y que su salida sea la ejecución del código de tres direcciones de una entrada válida.

Dicha herramienta será una aplicación con un entorno altamente gráfico, intuitivo y didáctico para el usuario final. Se requiere que la construcción del código de entrada del usuario pueda realizarse de manera gráfica; esto quiere decir que la herramienta no solo tendrá un cuadro de texto plano como entrada del lenguaje, también, contará con un sistema de bloques de código de alto nivel que podrán arrastrarse hacia el área de entrada por medio de drag-and-drop, para ejemplificar se puede decir que el comportamiento de dichos bloques serán muy similares a estar uniendo un conjunto de legos como se ve en la siguiente imagen.



Muestra de Blockly, una API de Google para crear proyectos que incorporen funciones de drag-and-drop relacionados con programación.

Al momento de la construcción del programa de entrada se podrá observar la generación de código intermedio en tiempo real. Esto quiere decir que conforme el usuario construya su programa, en un área contigua a este se estará desplegando el lenguaje de tres direcciones correspondiente a la entrada que se encuentre en ese momento, y este se actualizará cada vez que se agregue algo nuevo al código de entrada.

El Basic3D es un lenguaje de alto nivel muy completo. Dentro de este se podrán definir estructuras (similares a los structs de C), así mismo el lenguaje también soporta la definición de variables, métodos y funciones entre otras características que se explicarán posteriormente dentro de este documento. Adicional, el usuario tendrá la posibilidad de elegir el modo en que desea editar su código de alto nivel de entrada para la aplicación. Los modos serán tres, los cuales representan niveles de complejidad; estos niveles son básico, intermedio y avanzado. Cada uno de estos tendrán diferentes características y limitaciones.

El código intermedio generado por la aplicación estará en formato de código de tres direcciones que podrá ser ejecutado por la aplicación. Para realizar la ejecución del código intermedio se contará con un módulo de depuración o debugger. Este debugger se utilizará para poder rastrear el comportamiento de la ejecución del código intermedio generado, lo cual facilitará la visualización de la ejecución del código de tres direcciones para el usuario final, con esto se quiere decir que se podrá visualizar en tiempo real el estado de las estructuras del entorno de ejecución (Stack, Heap y String Pool).

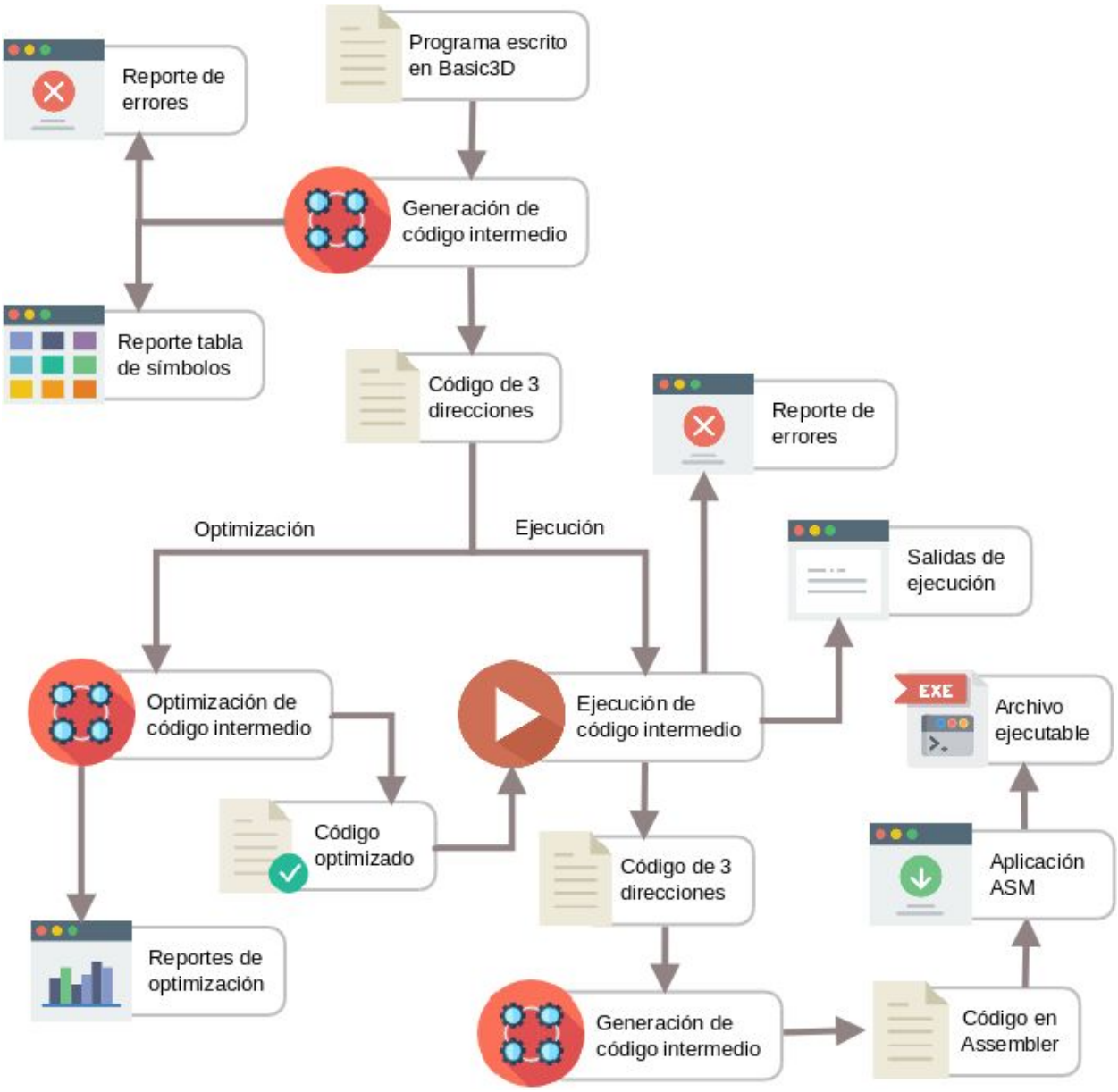
Será posible alterar el código 3D de salida, esto para que el usuario tenga la capacidad de experimentar con la ejecución del mismo, agregando temporales, cambiando valores, colocando breakpoints, entre otras cosas, dentro del código que se está ejecutando. Este debugger facilitará la interacción con el código a ejecutar por medio de acciones que incorpora normalmente cualquier debugger, las cuales son: ejecutar línea por línea, reanudar la ejecución hasta el próximo breakpoint, pausar la ejecución, entre otras.

La aplicación contará con la característica de poder optimizar el código intermedio generado en la salida. Este código optimizado deberá realizar las mismas operaciones que el código original.

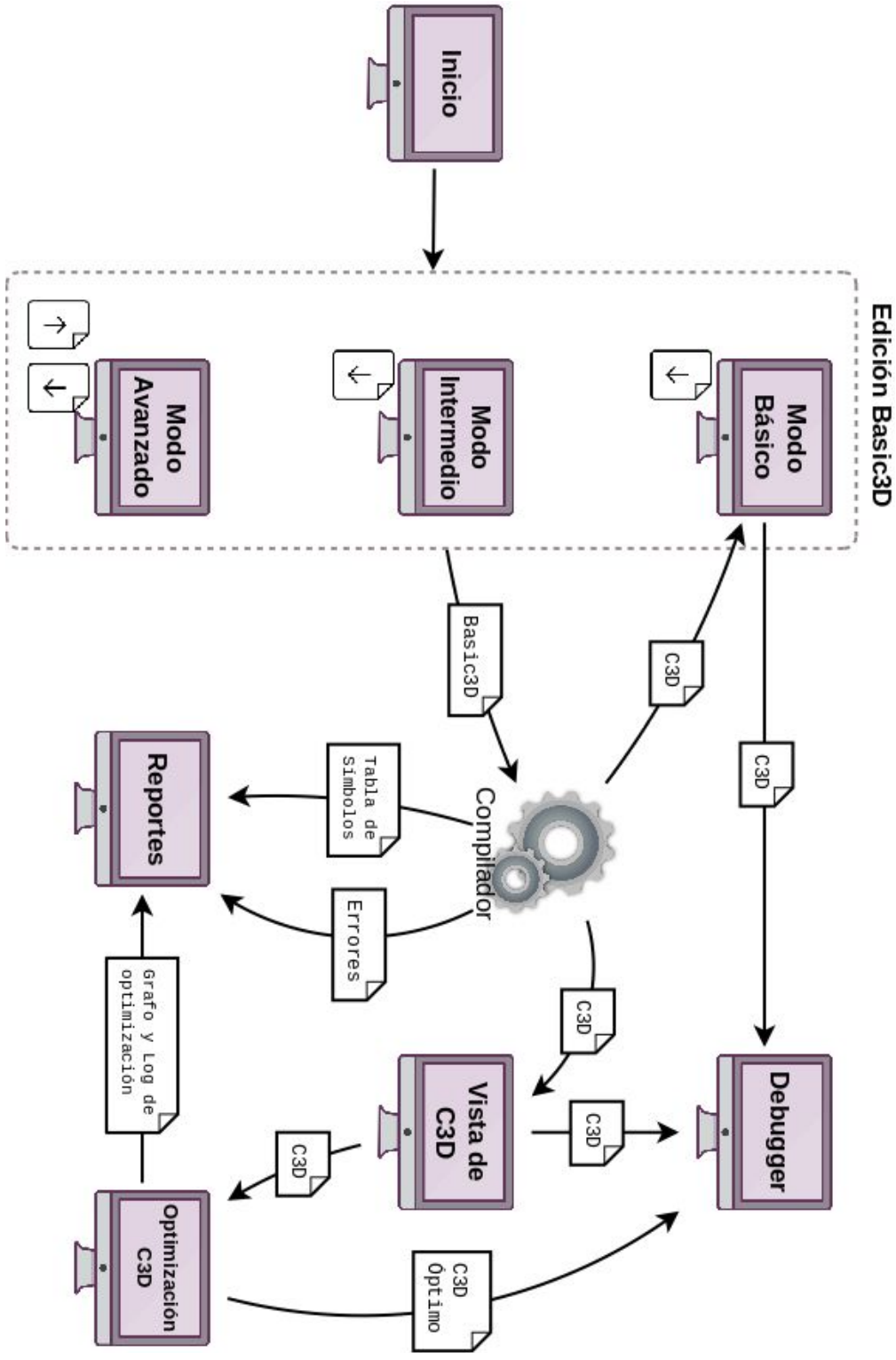
Una característica importante de la herramienta es la generación de reportes. Los reportes serán representativos de la ejecución del código de entrada por parte del usuario. Por lo que es necesario generar reportes durante el proceso de compilación, dentro de estos podemos mencionar el reporte de tabla de símbolos y el reporte de errores, además también será necesario la generación de reportes referentes al proceso de optimización del código de tres direcciones que se genere en la salida de la aplicación.

Por último, la herramienta gráfica de Basic3D contará con un módulo externo, el cual será una aplicación de escritorio que será encargada de traducir el código en formato de tres direcciones en su equivalente en código ensamblador y este podrá ejecutarse en un emulador 8086 para poder evaluar que ambas salidas son correctas y equivalentes.

La imagen en esta página muestra el proceso del IDE Basic3D de forma general, en él se observa que todo el proceso inicia con un programa escrito en el lenguaje de alto nivel Basic3D, que puede pasar por distintas fases produciendo diversas salidas. Así mismo se ve el proceso de comunicación con la aplicación externa para generar un ejecutable apartir del código intermedio (traducido a assembler).



En esta imagen se brinda una visión general de las pantallas que tendrá el proyecto, se establecen los flujos de información entre ellas y algunas funcionalidades especiales a cada una de ellas.



3. Funcionamiento del proyecto

Basic 3D, consiste en un lenguaje de programación de alto nivel, el cual se debe montar sobre una aplicación web desarrollada con javascript para poder desarrollar de manera intuitiva y fácil el código que se desee. Al abrir el IDE, se mostrará una pantalla de bienvenida, en donde el usuario podrá seleccionar el modo en el que desea realizar la creación y edición de código de alto nivel. El IDE contará con tres niveles de edición, los cuales son: básico, intermedio y avanzado.

Cada nivel de edición contiene características que permiten llevar a cabo la implementación del programa que se desea desarrollar, éstas incrementan conforme se avanza el nivel. En el momento que se elige un nivel y se termina el desarrollo de código de alto nivel para el mismo (con instrucciones de código según el nivel), se puede proceder a llevar a cabo el proceso de compilación, realizando la fase de análisis y síntesis, para la última mencionada, solamente generación de código intermedio y optimización.

Dentro de Basic3D es posible ver el código intermedio generado y realizar dos acciones las cuales son ejecutar dicho código u optimizarlo para posteriormente ejecutar el código optimizado (en el área de debug se realizan ambas ejecuciones). La ejecución de un programa permite validar, por medio de salidas a consola o por popups, las acciones que se realizaron durante todo el flujo del programa que se desarrolló. Todas las sentencias que pueden ser utilizadas en los distintos niveles son explicadas más a detalle en secciones posteriores.

Los programas desarrollados en Basic3D pueden ser almacenados en el dispositivo que ejecuta la aplicación web (únicamente para el modo avanzado), por medio de una descarga desde el navegador en un formato de texto plano.

Al finalizar los distintos procesos que se pueden realizar en Basic3D, se pueden generar reportes para validar el buen funcionamiento del programa.

Todas las imágenes de esta sección son sugerencias para que el estudiante tome una idea de qué producto de software se espera que elabore, las mismas están disponibles como un sitio web en https://esvux.github.io/Basic3D.

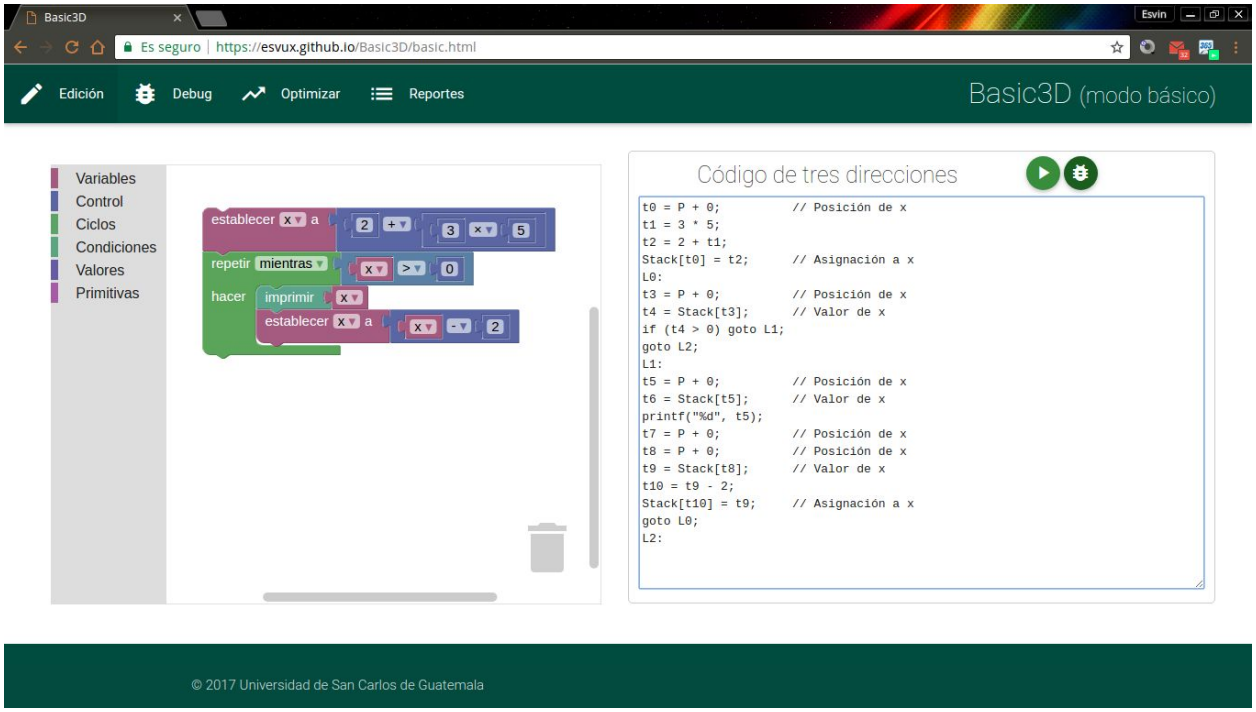
3.1. Pantalla de bienvenida

Al iniciar Basic3D, se debe mostrar una pantalla de bienvenida para crear un ambiente agradable al usuario y que le permita seleccionar de manera sencilla el tipo de edición que desea hacer.

> *En la pantalla de bienvenida se pueden observar las opciones de edición disponibles para el usuario, cada una de ellas con un ícono muy descriptivo, el propósito de esta pantalla es seleccionar el modo de edición para crear un programa opciones la aplicación deberá dirigirse hacia el modo de edición seleccionado.*

3.2. Modo de edición nivel básico

En este modo de edición el usuario tendrá acceso a un conjunto de bloques de acción, estos bloques podrán ser apilados y/o conectados entre sí para formar un único programa a ejecutar, es decir que únicamente se edita el método Principal de un programa de Basic3D.



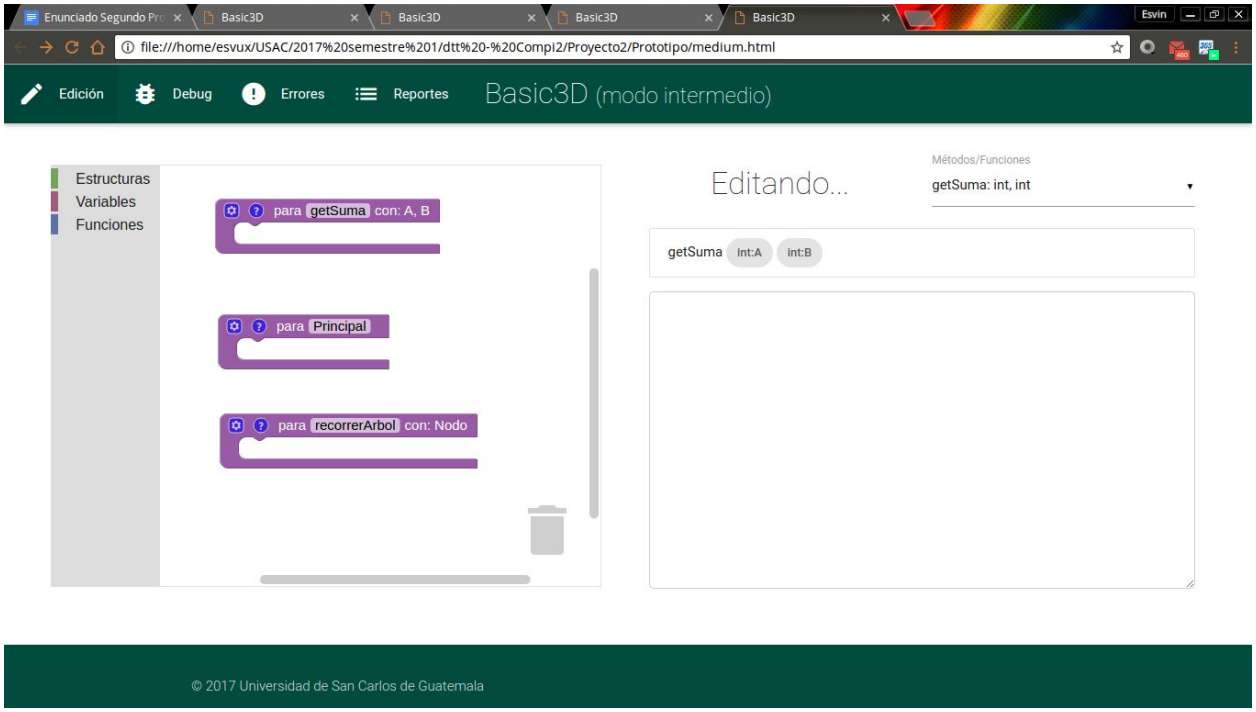
> *En la imagen se muestra que mientras los bloques van siendo conectados el código intermedio será mostrado en tiempo real en el área ubicada al lado del área de edición por bloques.*

En este modo de edición las características del lenguaje Basic3D se limitan a:

Característica	Observaciones
Asignaciones	Solo para variables locales, sin arreglos
Sentencias de control	Todas las descritas en la sección 4.8
Sentencias de ciclos	Todas excepto Timer
Funciones primitivas	Todas pueden ser utilizadas
Excepciones	Todas las que se detallan en la sección 4.12

3.3. Modo de edición nivel intermedio

En este modo de edición se podrá crear la estructura básica de un programa de Basic3D haciendo uso del panel de edición de drag-and-drop. En este modo se permite la creación de estructuras y la declaración de funciones o métodos con o sin parámetros; Haciendo una analogía con otro lenguaje de programación, este modo de edición funciona como un creador de lo que en java son las interfaces. Más información sobre interfaces de java en: https://es.wikipedia.org/wiki/Interfaz_(Java)

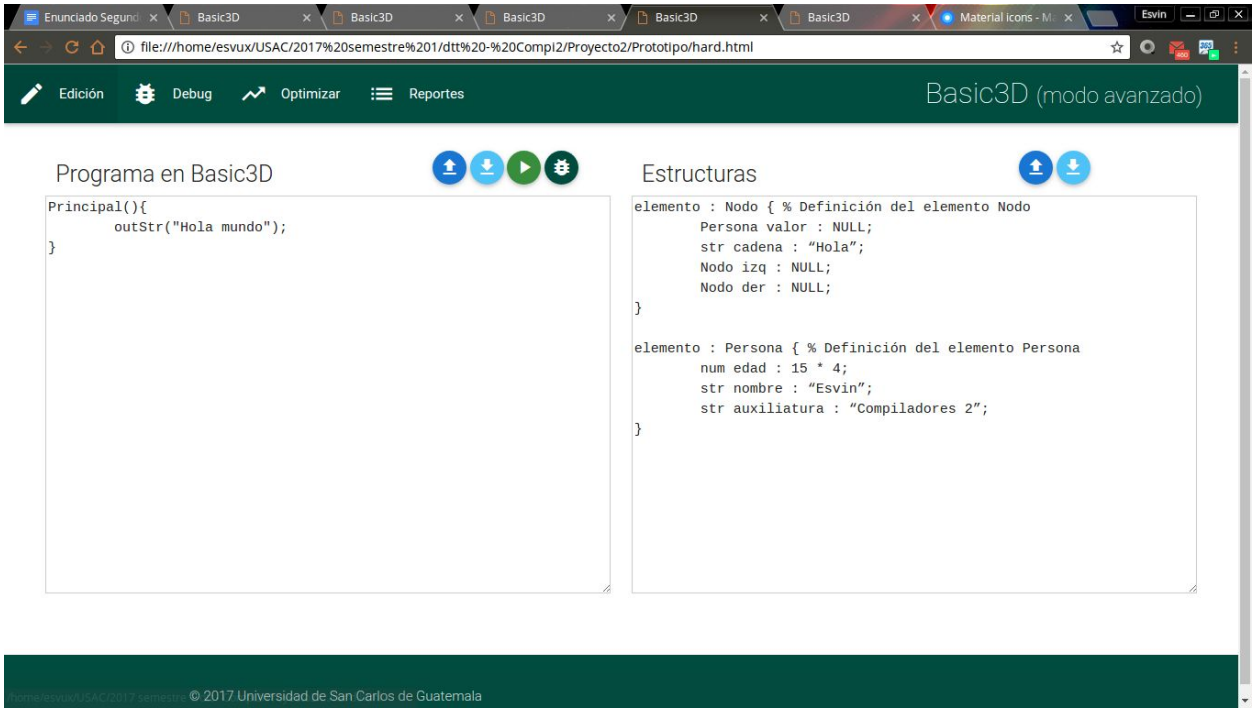


> *En la imagen de arriba puede verse el funcionamiento de este modo de edición, en donde se arrastran las funciones para definir su nombre, tipo y sus parámetros; al ir arrastrando las funciones, se irá agregando el identificador de cada una de ellas en un combobox, al seleccionar una u otra función se podrá editar su conjunto de instrucciones por medio de texto plano. Al determinar de que el programa está listo se puede generar el código intermedio de dicho programa, el cual será trasladado a un área de visualización previa al debugger.*

El modo de edición intermedio cuenta con soporte para todas las instrucciones exceptuando el manejo de arreglos.

3.4. Modo de edición nivel avanzado

Este modo de edición no utiliza los elementos gráficos del drag-and-drop y se focaliza en crear programas a base únicamente de texto, en este modo se permiten todas las características e instrucciones descritas en la sección del lenguaje de alto nivel Basic3D.

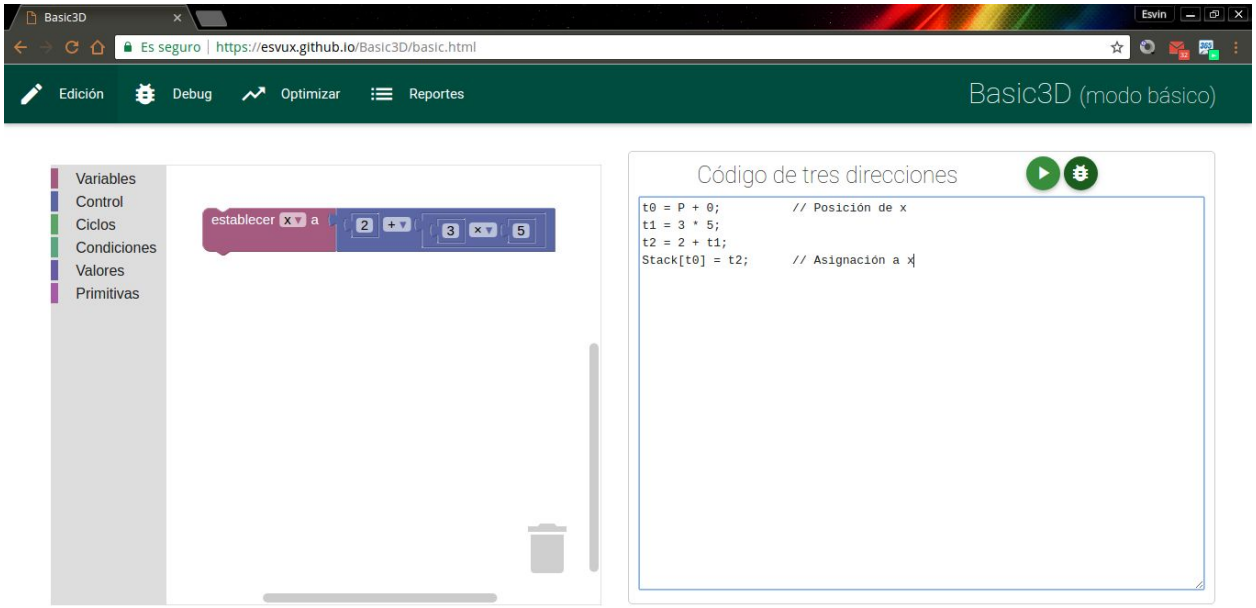


> *En la imagen anterior se muestra un IDE basado completamente en texto, que además de permitir la edición de archivos, hace posible abrir y guardar (cargar y descargar) archivos con extensión ".s3D" para estructuras, o ".b3D" para métodos y funciones. Al seleccionar la opción de ejecutar el código se deberá generar el código intermedio, para luego automáticamente enviarlo al área de previsualización.*

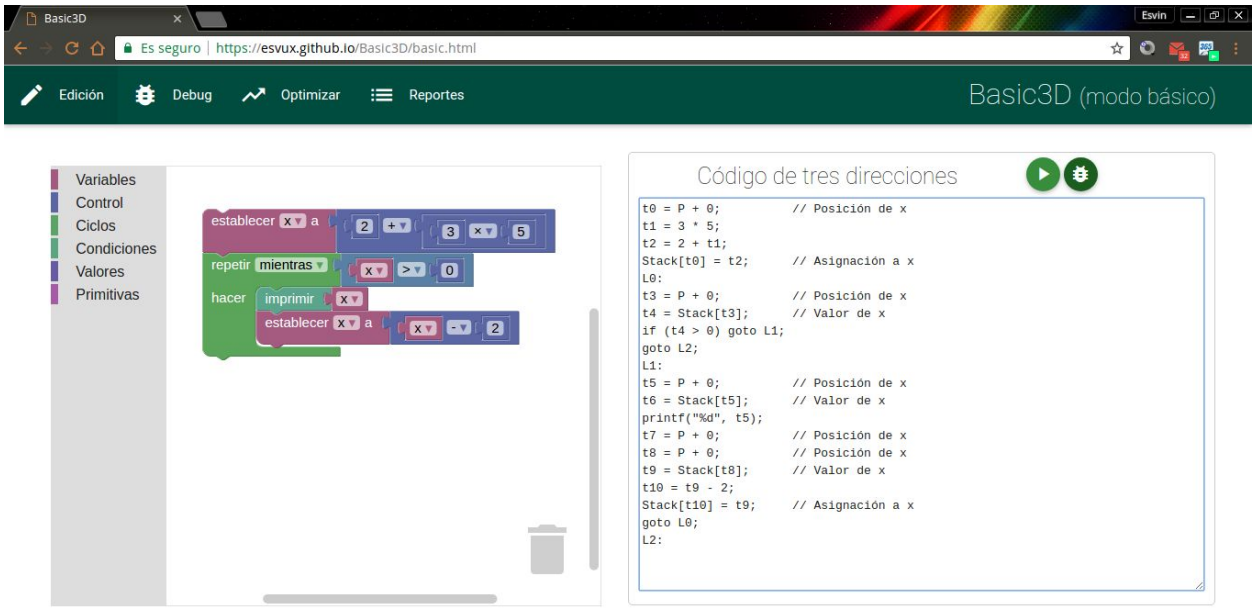
Para este modo de edición no existen limitaciones en cuanto a las instrucciones de Basic3D que se puedan agregar a cualquier programa creado por el usuario.

3.5. Generación de código intermedio

La generación de código intermedio para el modo de edición básico será en tiempo real, esto quiere decir que al momento de estar uniendo los bloques del código de alto nivel, se deberá estar generando la salida a un lado del área de drag and drop de manera que sea visible.



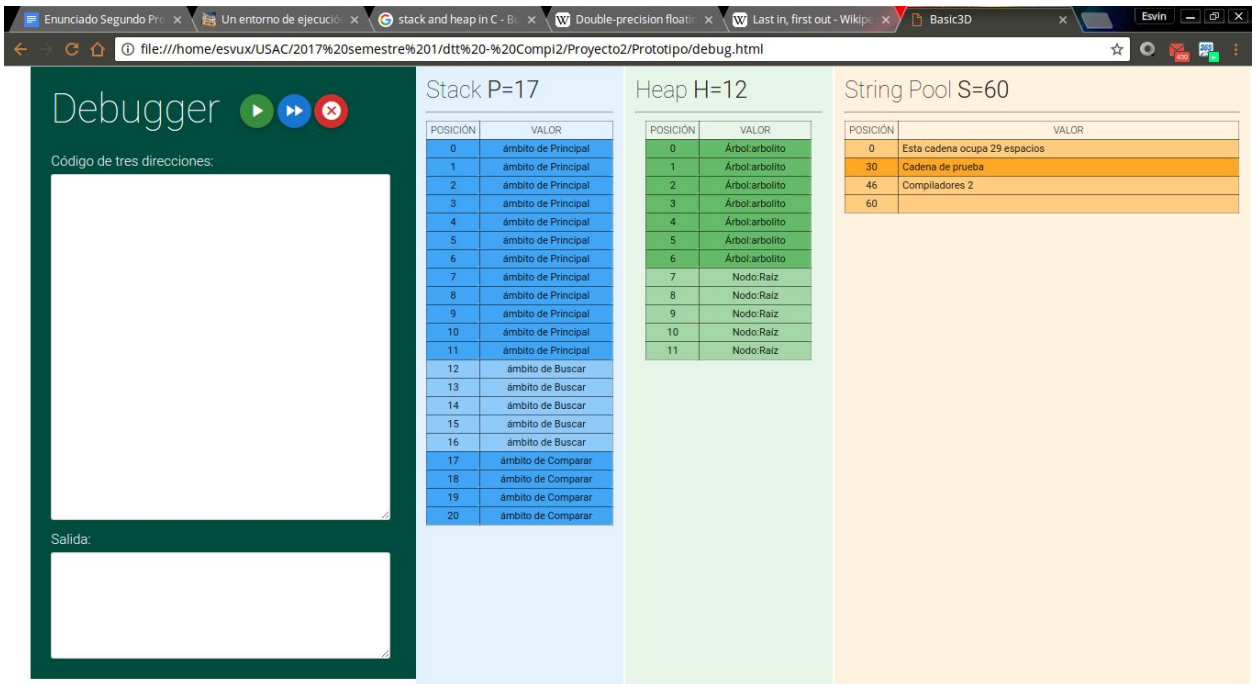
> *En ambas imágenes se puede se puede observar el código intermedio en el área de texto de la derecha, éste se tendrá que ir generando conforme los bloques del lado izquierdo son conectados entre sí. Los comentarios colocados son opcionales*



Mientras que los otros dos modos de edición (intermedio y avanzado) generarán su código intermedio bajo demanda del usuario, es decir pulsando un botón para iniciar el proceso de compilación, finalizando con el código intermedio generado puesto en el área de debug para su posterior ejecución.

3.6. Debugger

Al momento de tener generado el código intermedio, independientemente del modo de edición elegido, se podrá seleccionar la opción de **Debug**, para trasladar el código intermedio generado al área de debug y así poder ejecutarlo, es importante mencionar que en esta sección se debe llevar el reporte de manera gráfica del estado de las estructuras de control del entorno de ejecución, el Stack y su puntero P; el Heap y su puntero H; y el String Pool y su puntero S (éstas estructuras se detallan en la sección 6).



> Durante la ejecución del código colocado en el área de texto, las estructuras del entorno de ejecución tendrán que irse actualizando de forma automática, mostrando los valores según se describe en los párrafos siguientes.

Para cada posición del Stack y el Heap, se deberá colocar el número que le corresponde y el valor que almacena, además de que se debe colorear cada ámbito (Stack) o cada estructura (Heap) de manera distinta, una opción sería intercalar tonos claros y obscuros, como se muestra en el ejemplo, esto con el objeto de distinguir de mejor manera la diferencia entre los espacios reservados para cada ámbito o estructura.

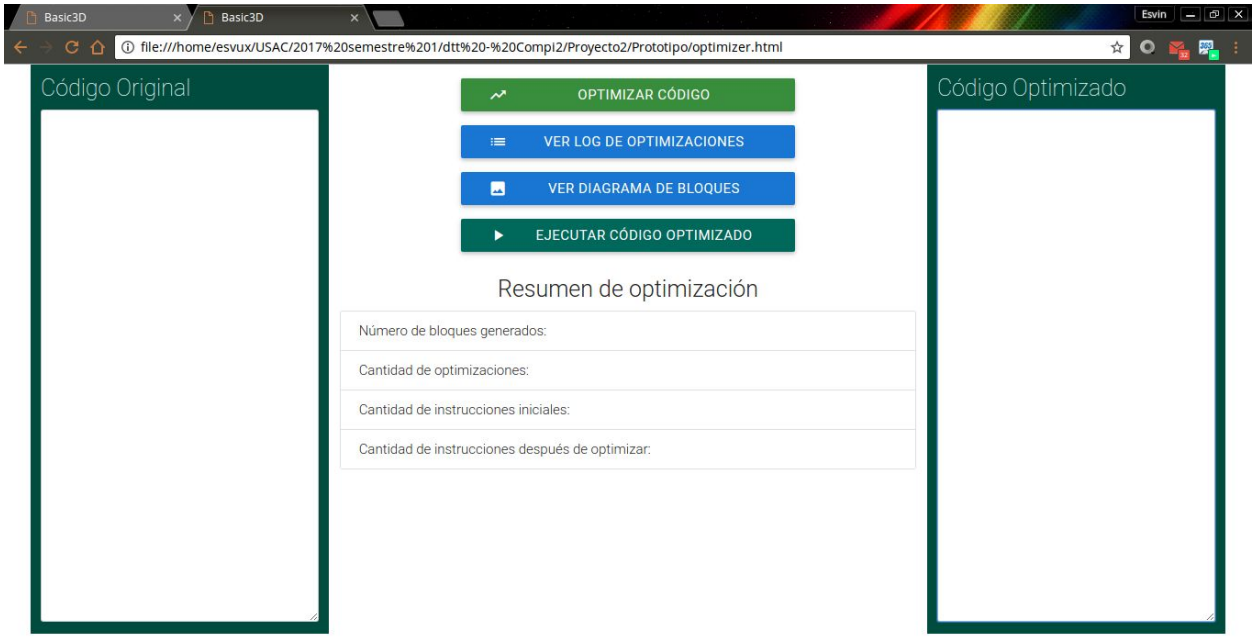
Para el String Pool se deberá mostrar la posición inicial de la cadena y el valor que se almacena en ella. Estas estructuras deben refrescarse con cada paso del debug; Al Stack se le asociará un procedimiento de limpieza para que al abandonar un ámbito éste quede vacío para

las próximas ejecuciones, evitando así confusiones por datos erróneos al reutilizar el espacio que otro ámbito ya ha utilizado.

Al momento de iniciar el proceso de debug (ejecución), **TODAS las estructuras deberán estar vacías**, esto quiere decir que en cada una de sus posiciones deberán de tener asignado NULL, en los ejemplos se les colocó valores irreales solo por fines demostrativos.

3.7. Optimizador de código

El optimizador de código debe ser un área a donde llegará el código intermedio generado (por cualquiera de los modos de edición) para que se le apliquen diferentes técnicas de optimización para así poder generar un código intermedio equivalente pero más reducido. El algoritmo utilizado para optimizar el código intermedio será una mezcla entre dos de las principales técnicas de optimización, esto se explica más detalladamente en la sección 8.



> *En la imagen se muestran dos grandes áreas de texto, en la izquierda irá el código intermedio original, en el centro hay un panel de acciones entre las que están la optimización del código intermedio, ver algunos reportes de la optimización y por último la posibilidad de ejecutar el código optimizado, esta última opción llevará el código optimizado al área de debug para su ejecución. En la derecha debería de encontrarse el código optimizado. Además se puede observar abajo en el centro un resumen de datos breves de la optimización.*

3.8. Generación de lenguaje ensamblador

El proyecto contará con un módulo externo, el cual será una aplicación de escritorio que funcionará como cliente-servidor de manera unidireccional. Esta aplicación será la encargada de traducir el código de tres direcciones en lenguaje ensamblador. Por lo que la herramienta

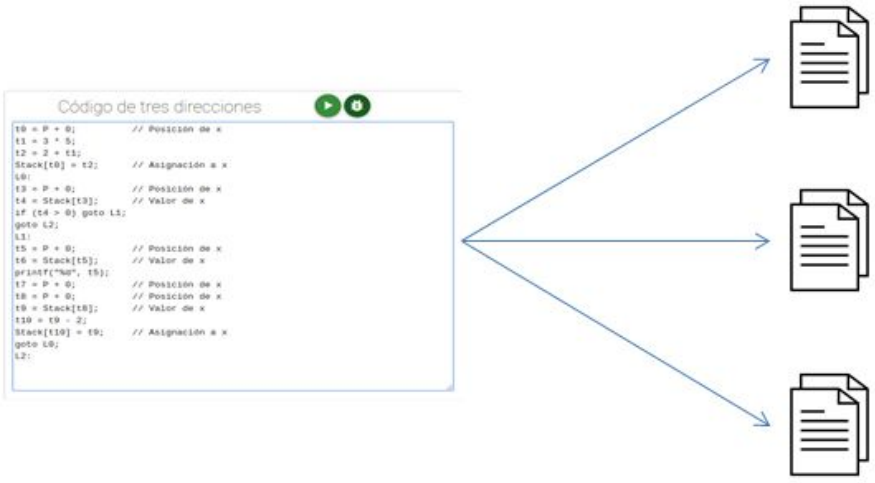
gráfica de Basic3D enviará su código de salida a esta aplicación de escritorio, lo traducirá y generará un código en lenguaje ensamblador, el cual podrá ser ejecutado en un emulador 8086.

Para ver una explicación más a detalle de este módulo externo, ir a la sección 9.

3.9. Sección de reportes

En esta sección se colocarán todos los reportes que generan las diversas acciones que se ejecuten dentro de la aplicación, se solicitan los siguientes reportes:

- Reporte de la tabla de símbolos
- Reporte de los errores encontrados durante el proceso de compilación
- Reporte gráfico de los bloques generados en el proceso de optimización
- Log con las optimizaciones realizadas.



Para ver una explicación más a detalle de los reportes a generar, ir a la sección 10.

4. El lenguaje de alto nivel Basic3D

El lenguaje de alto nivel Basic3D se caracteriza por ser fuertemente tipificado, en otras palabras, este lenguaje define un tipo para todo elemento que es declarado en él. Tiene soporte para operaciones aritméticas, lógicas y relacionales, que a su vez incluyen casteos implícitos entre datos; además de gestionar estructuras complejas compuestas por distintas propiedades; cuenta con estructuras de control y ciclos, variables globales, métodos y funciones con o sin parámetros.

Durante toda esta sección se detallarán las instrucciones que componen el lenguaje de alto nivel, los ejemplos cortos de la sintaxis de cada instrucción y de manera adicional, estan alojados en: https://goo.gl/6Y7Ebz.

4.1. Comentarios

Basic3D permite los dos tipos clásicos de comentarios. De una línea, que empezarán con **%** y terminarán con un salto de línea y también comentarios de múltiples líneas que estarán delimitados por los símbolos **¿** y **?**.

```
% Este es un comentario de una sola línea
¿ Y este es un comentario
que ocupa varias líneas ?
```

4.2. Tipos de datos

En esencia, los tipos de datos primitivos que soportará Basic3D son: booleanos, números (enteros o con punto decimal) y cadenas de caracteres (texto); estos tipos de datos son asociados a una palabra reservada y también a una definición formal de su dominio que puede observarse en la siguiente tabla.

Tipo de dato	Palabra reservada	Posibles valores
Booleano	`bool`	true o false
Numérico	`num`	Cualquier número positivo o negativo
Texto (o cadena)	`str`	Secuencias de caracteres encerradas entre comillas dobles “así” o entre comillas simples ‘así’

Algunos ejemplos de valores para los tipos de datos que se detallaron en la anterior tabla pueden ser:

Tipo	Valores de ejemplo
`num`	90 -190212 120 -90.06 -5123.3905
`str`	“Cadena de prueba” ‘Cadena de “prueba”, pero con comillas simples’

Dentro de las cadenas de texto (para ambos estilos de comillas) se permiten algunas de las más útiles secuencias de escape que a continuación se enlistan junto a su definición formal:

Caracter	Secuencia de escape	Caracter	Secuencia de escape
`\n`	Salto de línea	`\\`	Para colocar una barra invertida
`\t`	Tabulación horizontal	`\′ \″`	Según sea necesario en la cadena

4.3. Operadores

Para manipular datos y valores es necesario definir, tanto las operaciones que se realizarán, como su precedencia, sus limitantes o validaciones y los valores que se obtendrán producto de la combinación de diferentes tipos de datos según la operación que se haya realizado. En esta sección se tratan las expresiones que forman Basic3D.

4.3.1. Operadores aritméticos

Los operadores aritméticos soportadas por Basic3D son los que se detallan en la siguiente tabla:

Símbolo	Operación
+	suma o concatenación (dependiendo de los argumentos)
-	resta (incluido también el menos unario)
*	multiplicación
/	división
%	módulo
^	potencia

4.3.2. Operadores relacionales

Las condiciones deben ser evaluadas según el tipo de dato al que pertenezcan los operandos, Por lo que para **TODAS** las operaciones relacionales, el tipo de ambos operadores debe ser el mismo y siempre que una operación sea evaluada correctamente producirá como resultado un valor booleano.

Comparación de cadenas

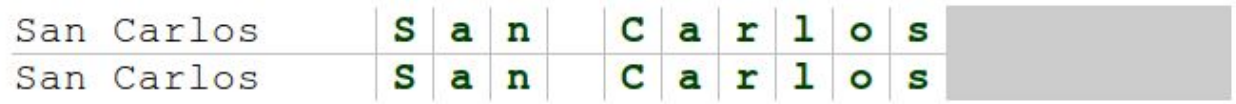
Para los valores de texto (cadenas) se realizará una comparación lexicográfica, con base en el código ascii de cada uno de los caracteres que componen las dos cadenas, que determinará el orden de una cadena respecto a otra.

El operador ==, si la cadena de la izquierda es idéntica tanto en orden de caracter por caracter como en la longitud de caracteres a la cadena de la derecha. El operador !=, realiza una comparación contraria a la anterior, es decir que se devolverá falso si las cadenas son idénticas.

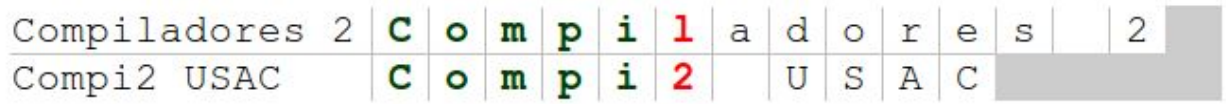
El operador <, determina si la cadena izquierda antecede a la cadena derecha, mientras que el operador > determina si la cadena izquierda precede a la cadena de la derecha, según el código ascii de cada uno de los caracteres que componen las dos cadenas en cuestión.

Ejemplos:

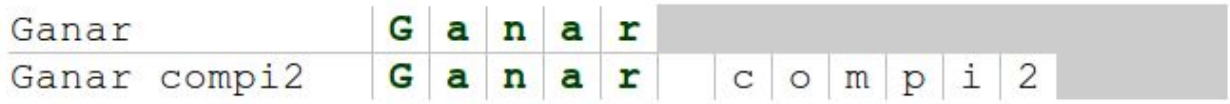
En este ejemplo se comparan dos cadenas, y se puede observar como, caracter a caracter, las dos cadenas coinciden y al llegar al final de las cadenas (área gris) se llega a la conclusión de que las dos cadenas son iguales.



En este otro ejemplo tenemos dos cadenas diferentes, el análisis inicia caracter a caracter y al llegar a la posición 6 los caracteres dejan de coincidir, por lo que según el código ascii de los caracteres se debe tomar la decisión de qué orden les corresponde a ambas cadenas; en este caso el código ascii de la letra **`l`** es **`108`**, mientras que el código de el número **`2`** es **`50`** la cadena **`'Compi2 USAC'`** es "menor que" la cadena **`'Compiladores 2'`**.



Por último este ejemplo muestra el comportamiento que debería de tener la comparación de textos al toparse con el caso de que la cadena termine y no haya una discrepancia entre los caracteres. Para realizar la comparación en este punto se debe tomar el caracter que determina el final de la cadena **`'\0'`** y comparar su código ascii (**`0`**) contra el código ascii del caracter que corresponde a la otra cadena, el espacio en blanco (**`32`**). Por lo que en este ejemplo la cadena **`'Ganar compi2'`** es "mayor que" la cadena **`'Ganar'`**.



4.3.3. Operadores lógicos

Las operaciones lógicas siempre esperan como operando un dato de tipo lógico, pero si no se cumple ese tipo de dato es necesario recurrir a los casteos implícitos que se detallan en la sección de casteos implícitos; los símbolos y su conectivo lógico se especifican en la siguiente tabla, además de incorporar la tabla de verdad para cada operador.

Operador: **&&** (and)	Operador: **||** (or)	Operador: **|&** (xor)

`true  && true  = true`
`true  && false = false`
`false && true  = false`
`false && false = false`
	`true  || true  = true`
	`true  || false = true`
	`false || true  = true`
	`false || false = false`
		`true  |& true  = false`
		`true  |& false = true`
		`false |& true  = true`
		`false |& false = false`

Operador: &? (nand)	**Operador: |?** (nor)	**Operador: !** (not)

`true  &? true  = false`
`true  &? false = true`
`false &? true  = true`
`false &? false = true`
	`true  |? true  = false`
	`true  |? false = false`
	`false |? true  = false`
	`false |? false = true`
		`! true  = false`
		`! false = true`

Nota: *Estas operaciones no pueden aplicarse directamente sobre valores de texto.*

4.3.4. Precedencia de operadores en las expresiones

Con el objetivo de determinar el orden correcto en que se deben de realizar las distintas operaciones que conforman una expresión; en la siguiente tabla se asocia un valor de precedencia para cada operador.

Operador						Precedencia
`+`			`-`			6
`*`			`/`			7
`^`						8
`- (unario)`						9
`==`	`!=`	`>`	`<`	`>=`	`<=`	5
`||`			`|?`			1
`&&`			`&?`			2
`|&`						3
`!`						4

4.3.5. Sistema de tipos

Basic3D permite operaciones entre diferentes tipos de datos, así como asignaciones de valores a variables de otros tipos. En la siguiente tabla se detalla el comportamiento de cada operador aritmético según el tipo de los operandos, las celdas de color negro con los operadores representan la operación que se realiza entre las celdas que contienen el tipo de los operandos, siendo las intersecciones (celdas de color blanco) la acción que se debe realizar. Si la operación no es permitida en la tabla aparecerá la palabra error.

Operador	Operación	Resultado
Suma (+)	bool + bool bool + num num + bool num + num str + bool str + num bool + str num + str str + str	suma suma suma suma concatena concatena concatena concatena concatena
Resta (-)	bool - bool bool - num num - bool num - num str - bool str - num bool - str num - str str - str	error resta resta resta error error error error error
Multiplicación (*)	bool * bool bool * num num * bool num * num str * bool str * num bool * str num * str str * str	multiplica multiplica multiplica multiplica error error error error error
División (/)	bool / bool bool / num num / bool num / num str / bool str / num bool / str	error divide divide divide error error error

	num / str str / str	error error
Módulo (%)	bool % bool bool % num num % bool num % num str % bool str % num bool % str num % str str % str	error módulo módulo módulo error error error error error
Potencia (^)	bool ^ bool bool ^ num num ^ bool num ^ num str ^ bool str ^ num bool ^ str num ^ str str ^ str	error potencia potencia potencia error error error error error

Ejemplos:

Operación	Resultado
5 + 20 * 10	num : 205
-3/40 - (5+true)*2.40	num : -14.475
“El resultado de 2+3*5 es: ” + (2 + 3 * 5)	str : “El resultado de 2+3*5 es: 17”
“El resultado de 2+3*5 es: ” + 2 + 3 * 5	str : “El resultado de 2+3*5 es: 215”

En la siguiente tabla se muestra qué operadores relacionales son válidos según el tipo de dato de sus operandos, las celdas con la palabra Error, representan un error semántico de incompatibilidad de tipos para la operación en cuestión.

Operador	Operación	Resultado
Igualación (==)	bool == bool num == num str == strr	válido válido válido
Diferencia (!=)	bool != bool num != num	válido válido

	str != str	válido
Mayor que (>)	bool > bool num > num str > str	error válido válido
Menor que (<)	bool < bool num < num str < str	error válido válido
Mayor o igual que (>=)	bool > bool num > num str > str	error válido error
Menor o igual que (<=)	bool <= bool num <= num str <= str	error válido error

Nota: para las operaciones lógicas, ambos operadores deben ser del mismo tipo de ambos lados de la operación.

Toda variable u estructura puede compararse contra el valor **NULL** por medio de los operadores == o !=, la palabra reservada NULL forma parte del lenguaje de alto nivel Basic3D y representa la nada (un valor que nunca ha sido asignado).

Para convertir valores de un tipo de dato a otro se debe realizar una operación de conversión según determinadas reglas, a continuación se detallan las reglas de casteo implícito que serán aplicadas antes de realizar alguna operación según sea conveniente para el compilador.

Tipo origen	Tipo destino	Regla para la conversión
Numérico	Booleano	El valor 0 equivale a false, cualquier otro valor será verdadero
Numérico	Texto	El valor del texto serán los dígitos que componen su valor
Booleano	Numérico	Falso equivale a 0, mientras que verdadero equivale a 1
Booleano	Texto	Este valor se convertirá a “true” si el valor es verdadero y a “false” si el valor fuese falso
Texto	Booleano	Ver sección de casteos explícitos
Texto	Numérico	

Los casteos implícitos expuestos en la tabla anterior aplica **para operaciones** en donde un tipo de dato sea obligatoriamente requerido **como un tipo específico**.

4.4. Variables

Las variables en Basic3D pueden ser tanto globales como locales, se pueden declarar en cualquier porción de código, y la sintaxis de su declaración es la siguiente:

```
<tipo> <id> : <valor_inicial>;
<tipo> <id> , <id>, <id> : <valor_inicial>;
```

Es importante mencionar que el valor inicial es opcional y de no colocarse la o las variables deberán ser consideradas siempre como nulas, es decir que su valor podría ser cualquier cosa sobre la que el programador no tiene garantía alguna, para las variables del tipo de algún **`element`** declarado previamente se debe asignar NULL por defecto.

```
Nodo raiz; % esta variable tendría como valor NULL
num v1, v2, v3 : 0; % v1, v2 y v3 tendrían valor de 0
str cadena_prueba : "Carmen, se me perdió la cadenita";
bool temp; % esta variable tendría un valor nulo o indefinido
```

4.5. Arreglos

Basic3D permite la declaración de arreglos de tipos de datos primitivos (bool, num, str) de múltiples dimensiones, estos arreglos son declarados de la siguiente forma:

```
array : <id> [<num>..<num>] of <tipo>;
array : <id> [<num>..<num>][<num>] of <tipo>;
```

El mapeo de estos arreglos para modificar o consultar su información deberá hacerse por medio de un acceso de filas, columnas, profundidad. Cada dimensión del arreglo será definida como una pareja de índices o como un índice independiente, para el caso en donde se define una pareja de índices los elementos podrán ser accedidos desde la posición <índice inferior> hasta <índice superior> - 1; mientra que para el índice independiente (un único número) el acceso puede hacerse desde la posición 0 hasta la posición <índice> - 1.

Es necesario verificar en tiempo de ejecución que éstos índices no sean sobrepasados. Además es necesario verificar en el momento de compilación que la pareja de índices tenga un formato correcto, es decir que el inferior sea menor que el superior. NO es posible redefinir arreglos en cuanto a su cantidad de dimensiones o longitud de las mismas y éstas pueden ser determinadas únicamente por números enteros puntuales (no expresiones).

```
array : vector [5..20] of str;
array : estados [5][3] of bool;
array : casillas [1..5][4][1..3] of num;
```

4.6. Estructuras

Similar a los **structs** en C, Basic3D soporta lo que a partir de esta sección llamaremos **elementos**; los elementos son agrupaciones de propiedades, cada definición de un elemento cuenta con un nombre único para identificarse. Una propiedad es una variable asociada a un elemento que cuenta con un nombre, un tipo y un valor inicial. Para declarar un elemento se utilizará la palabra reservada **element** y la siguiente sintaxis:

```
element : <id> {
      <tipo> <id> : <valor_inicial>;
      % mínimo una propiedad
      <tipo> <id> : <valor_inicial>;
      <tipo> <id> : <valor_inicial>;
      % pero pueden venir muchas más
}
```

Todo elemento tiene que contar con al menos una propiedad y el valor inicial para las propiedades puede ser cualquier expresión, sin incluir llamadas a métodos. Las propiedades que tienen como tipo un elemento siempre se asignará como valor inicial el valor **NULL** o una llamada al método especial **create**.

```
element : Nodo { % Definición del elemento Nodo
      num valor : 15 * 4;
      str cadena : "Hola";
      Nodo izq : NULL;
      Nodo der : NULL;
}
```

Estando dentro de alguna porción de código ejecutable, por ejemplo el cuerpo de un método se puede "instanciar" un elemento declarado como una variable. Esta "instanciación" se realizará por medio de la palabra reservada **create**, que representa una instrucción propia del lenguaje. En el siguiente recuadro se detalla la sintaxis general y un ejemplo de la aplicación de esta instrucción.

```
<id_element> <id> : create(<id_element>);
Nodo raiz : create(Nodo);
```

Esta instrucción, si hiciéramos una analogía con los lenguajes orientados a objetos, haría las veces de constructor para el elemento Nodo realizando las asignaciones creadas en la declaración del elemento en cuestión y otorgándole una posición en memoria para almacenar los valores de sus propiedades. En la implementación, el valor **NULL** puede ser manejado como undefined o como un valor negativo muy grande, como -92038746.100212 por ejemplo.

Es importante destacar que es posible definir elementos dentro de elementos, por lo que la definición de elementos no se basa únicamente en tipos de datos primitivos, sino en estructuras más completas que representan una agrupación ordenada de datos.

```
element : Arbol{ % Definición del elemento Arbol
     element : Nodo{ % Definición del elemento Nodo
          num valor : 0;
          Nodo izq : NULL;
          Nodo der : NULL;
     }
     Nodo izq : NULL;
     str : titulo : "Sin título";
}
```

4.7. Asignaciones

Es posible realizar asignaciones a las variables o arreglos creados por medio de distintos operadores; una asignación cuenta con ciertas limitantes las limitantes aplicarán tanto para el destino como en el valor a asignar.

Para la asignación se utilizará el operador =, el lado izquierdo se restringirá a cualquier variable, sin importar si el identificador hace referencia a: un element instanciado o alguna propiedad del mismo, accesible por un punto (.) o un arreglo (que puede ser el arreglo completo o una posición específica del mismo) o una variable local de tipo primitivo.

Mientras que el lado derecho tiene las siguientes limitantes, si el destino es de tipo primitivo, puede venir cualquier expresión aritmética, lógica o relacional válida (incluyendo acceso a otras variables, estructuras, llamadas a métodos, etc.), sin embargo si el destino es una estructura (element), únicamente puede venir expresiones cuyo resultado sea una estructura del mismo tipo que el destino.

```
% Declaración de variables
num var1;
array : vector [5][20] of str;
Nodo nodo;
Persona persona;

% Asignaciones válidas
vector = getMatriz(); ¿ Asignación del resultado de una función
a una variable ?
```

```
var1 = getValor() + 2.059; ¿ Asignaciones del resultado del
retorno de una variable sumado a un número decimal a una
variable ?

vector[2][5] = "var1 vale: " + var1; ¿ Asignación de una cadena
a una posición de un arreglo de cadenas ?

nodo = create(Nodo);  ¿ Asignación de la creación de un nuevo
element?

nodo.alguien = create(Persona); ¿ Asignación de un nuevo
element a un atributo de un element existente ?

persona = nodo.alguien; ¿ Asignación de un element a un
atributo de un element existente ?
```

4.8. Control

Las estructuras de control se describen a detalle en esta sección, tanto su sintaxis, como su comportamiento.

4.8.1. Instrucción IF-THEN

Esta instrucción debe contar con un cuerpo para el `if` principal, al inicio se colocará la palabra reservada if, seguida de su condición, y la palabra reservada then para indicar que lo que se encuentra a continuación es el cuerpo de la sentencia que se ejecutará si se cumple la condición.

```
if ( <expresión> ) then {
      ¿ Instrucciones que se ejecutarán si se cumple la
      condición ?
}
```

4.8.2. Instrucción IF-THEN-ELSE

Esta instrucción debe contar con un cuerpo para el `if` principal, al inicio se colocará la palabra reservada if, seguida de su condición, y la palabra reservada then para indicar que lo que se encuentra a continuación es el cuerpo de la sentencia que se ejecutará si se cumple la condición.

Por lo contrario, se continuará la ejecución del resto del código que no se encuentre dentro del if principal y que indique que es el código que se ejecutará si la condición no se cumple. Esta sentencia es iniciada con la palabra reservada else seguida del cuerpo a ejecutar si la condición no se cumple.

```
if ( <expresión> ) then {
      ¿ Instrucciones que se ejecutarán si se cumple la
      condición ?
} else {
      ¿ Instrucciones que se ejecutarán si no se cumple ninguna
      condición ?
}
```

4.8.3. Instrucción SWITCH

Esta instrucción cuenta con una expresión, un modo de operación (valor booleano), una lista de casos para ejecutar y opcional existirá un bloque de instrucciones por `default`.

El comportamiento de esta instrucción varía según sea el modo de ejecución. Si el modo es verdadero el `switch` hará que al cumplirse uno de los casos se ejecuten las instrucciones que le corresponden y luego volverá a evaluar toda la instrucción (funcionando casi como un ciclo); mientras que si el modo es falso y se cumple alguno de los valores para entrar a un caso en

particular la ejecución del programa continua de caso en caso, sin preguntar o consultar la condición, hasta encontrar una instrucción de salida `break` o el final de la instrucción. Sin importar el modo de ejecución, toda vez se haya cumplido alguno de los casos jamás se ejecutará el bloque de instrucciones por `default`, sin embargo estas instrucciones sí se ejecutarán si no se cumple alguno de los casos.

```
switch ( <expresión> , <modo> ) {
      case valor1 :
            % Instrucciones a ejecutar si expresión == valor1
      case valor2 :
            % Instrucciones a ejecutar si expresión == valor2
      default :
            % Instrucciones a ejecutar si no cumple ningún caso
            % Si se ejecuta este código, sin importar el modo,
            % la instrucción habrá finalizado
}
```

Es importante aclarar que los valores para los `case` podrán ser únicamente valores primitivos, números o cadenas, sin mezclar ambos tipos de datos en un solo `switch`. El **<modo>** puede ser sólo `true` o `false`.

Adicional, el switch tendrá la capacidad de identificar rangos dentro de sus casos. Esto quiere decir que es posible que dentro de un case pueda venir un rango de valores, estos pueden ser numéricos o de tipo caracter.

```
% Ejemplo para valores numéricos
```

```
      switch ( <expresión> , <modo> ) {
            case 1 - 10 :
                  % Instrucciones a ejecutar si la expresión está
                  dentro de los valores 1 a 10
            case 11 - 20 :
                  % Instrucciones a ejecutar si la expresión está
                  dentro de los valores 1 a 10
            default :
                  % Instrucciones a ejecutar si no cumple ningún caso
      }

% Ejemplo para valores de tipo caracter

      switch ( <expresión> , <modo> ) {
            case 'a' - 'g':
                  ¿ Instrucciones a ejecutar si la expresión está
                  dentro de los caracteres 'a' al 'g' ?
            case 'h' - 'm' :
                  ¿ Instrucciones a ejecutar si la expresión está
                  dentro de los caracteres 'h' al 'm' ?
            case 'n' - 'z' :
                  ¿ Instrucciones a ejecutar si la expresión está
                  dentro de los caracteres 'n' a la 'z' ?
            default :
                  % Instrucciones a ejecutar si no cumple ningún caso
      }
```

4.8.4. Instrucciones Branching

Las sentencias de branching o ramificación son sentencias condicionales que se encuentran dentro de métodos, funciones, sentencias de control y de flujo, según aplique.

4.8.4.1. Instrucción BREAK

Esta instrucción tiene la funcionalidad de romper o interrumpir la ejecución de otra instrucción, así pues si se encuentra dentro de algún ciclo hace que la ejecución del mismo finalice y da el foco de la ejecución a la siguiente instrucción fuera del ciclo.

Esta sentencia será utilizada únicamente por sentencias de flujo que se verán en la sección 4.9, las cuales serán `switch-case`, `while`, `for`, `repeat-until`, `do-while` y `loop`.

```
      break;
```

4.8.4.2. Instrucción CONTINUE

Aplicable únicamente a ciclos, su función es interrumpir la iteración actual para saltarse el resto y empezar la siguiente iteración.

Esta sentencia será utilizada únicamente por sentencias de flujo que se verán en la sección 4.9, las cuales serán las sentencia `switch-case`, `while`, `for`, `repeat-until`, `do-while` y `loop`.

```
continue;
```

4.8.3.3. Instrucción RETURN

Esta sentencia puede venir a cualquier nivel de profundidad dentro del código e interrumpirá toda la ejecución del método o función que esté corriendo en ese momento. El retorno vacío es exclusivo para los métodos, mientras que el retorno con una expresión es para controlar las funciones.

Como se mencionaba anteriormente, el return será exclusivo para métodos y funciones, pero este se encontrará en cualquier parte del código y podrá encontrarse más de uno en un solo método o función; por ejemplo, puede existir una sentencia de return por cada opción del switch-case, por lo que el método retornará un valor diferente según el valor del case..

```
return;
return <expresión>;
```

4.9. Ciclos

Los ciclos son sentencias que repiten una porción de código bajo cierto comportamiento, es decir que la manera en que controlan sus repeticiones depende de la instrucción que se trate.

4.9.1. Ciclo WHILE

Sentencia de ciclo que se ejecuta mientras el valor de su condición sea verdadero, en otras palabras sus iteraciones terminarán cuando la condición sea falsa.

```
while ( <expresión> ) {
      % Instrucciones a ejecutar si se cumple la condición
}
```

4.9.2. Ciclo DO-WHILE

Sentencia de ciclo que se debe ejecutar al menos una vez, luego debe comprobar la condición y ejecutar las instrucciones las veces que sea necesario mientras la condición se cumpla.

```
do {
      % Instrucciones a ejecutar si se cumple la condición
} while ( <expresión> )
```

4.9.3. Ciclo REPEAT-UNTIL

Sentencia de ciclo que debe ejecutarse al menos una vez y continuará la ejecución de sus instrucciones mientras la condición **NO** se cumpla.

```
repeat {
      % Instrucciones a ejecutar si se cumple la condición
} until ( <expresión> )
```

4.9.4. Ciclo FOR

Sentencia de ciclo, la cual permite inicializar o establecer una variable para control. Este ciclo contiene una condición que se verifica en cada una de las iteraciones que realice, y luego se debe definir una asignación simplificada que afecta directamente a la variable de control.

Cada vez que se ejecuta un ciclo para luego verificar si se cumple con la condición para poder continuar. El uso de esta sentencia de ciclo es ideal para en los casos donde se conozca el número de iteraciones a realizar o se necesite hacer un recorrido de arreglos.

```
for( <variable de control> ; <condicion> ; <asignacion> ){
      % Instrucciones a ejecutar si se cumple la condición
}
```

4.9.5. Ciclo LOOP

Sentencia de ciclo que lleva asociado un identificador, este ciclo ejecuta sus instrucciones infinitamente hasta que encuentre una sentencia `break` que le indique cuándo finalizar. Este break puede ser utilizado de dos maneras; la primera como se utiliza normalmente, y la segunda seguido del identificador, esta última se utiliza para poder detener la iteración del ciclo loop principal a pesar de que se encuentren otros ciclos anidados.

```
loop <id>{
      % Instrucciones a ejecutar si se cumple la condición
      break <id>;
}
```

Este ciclo se considera como la versión simple para sustituir la instrucción `while(true)`.

4.9.6. Ciclo COUNT

Sentencia de ciclo que ejecuta instrucciones las veces que se le indiquen según un valor número. Este ciclo es muy similar al ciclo `while`, con la diferencia que la expresión esperada no es una condicional, sino una expresión que tenga un valor numérico entero.

```
count ( <expresión> ) {
      % Instrucciones a ejecutar según el valor numérico de la
      expresión
}
```

4.9.7. Ciclo DO-WHILEX

Es un ciclo que se debe ejecutar al menos una vez si se cumple una de las dos condiciones, luego debe seguir ejecutando sí y solo sí las dos condiciones son verdaderas

```
do {
      % Instrucciones a ejecutar si se cumple la condición
} whilex ( <expresion_1>, <expresion_2> )
```

4.10. Métodos y funciones

Basic3D permite la declaración, y también la llamada, de métodos y funciones; a partir de este punto se asumirá que **un método es una función** de tipo void (vacío). Una función es un conjunto de instrucciones agrupadas bajo una estructura lógica que comparten un ámbito común. Cada función tiene asociados un tipo de dato (primitivo o de estructura), un nombre o mejor dicho un identificador, un conjunto de parámetros y un cuerpo, que son las instrucciones que componen la función. Las funciones de Basic3D soportan sobrecarga por parámetros. La sintaxis para la declaración de una función se define en este recuadro:

```
<tipo> : <id> (<parametros>) {
      % Instrucciones que componen el cuerpo de la función
}
```

Los tipos permitidos para la declaración de funciones son todos los datos primitivos, es decir **`void`**, **`num`**, **`str`** y **`bool`**, así como cualquier **`array`** de **`num`**, **`str`** o **`bool`**, o cualquier **`elemento`** que haya sido declarado con anterioridad. Los parámetros permiten también cualquiera de los tipos enumerados anteriormente, a excepción del tipo **`void`**, la manera de declarar los parámetros es por medio de una lista en donde cada item tiene la siguiente estructura y están separados entre sí por una coma.

```
<bool num str o id> <id>
<bool num str> <id> [<num>][<num>..<num>]
```

El envío de los parámetros varía según sea el tipo; entre parámetro **por referencia**, que indica que cualquier modificación sobre dicho parámetro, dentro del cuerpo del método que fue llamado, es como que se estuviera modificando la misma estructura o variable que se ha enviado: **Los cambios realizados en el parámetro se verán reflejados en la estructura (element) original**; o bien se pueden enviar parámetros por valor, lo que significa que los cambios realizados sobre el parámetro **NO** tienen efecto sobre el valor original. A continuación se listan las distintas formas de enviar parámetros.

- Los tipos numéricos o booleanos siempre serán enviados por valor
- El tipo cadena será enviado, por defecto, por valor; ha no ser de que se indique explícitamente que se desea enviar por referencia, esto se hará por medio del operador *, colocándolo entre el tipo y el nombre del parámetro: str * param1
- Al enviar una cadena como parámetro por referencia (utilizando el operador *) en una llamada a métodos se debe comprobar que dicha llamada lleve como parámetro un valor **referenciable**, es decir, un valor que tenga una posición de memoria fija, no se permite enviar cadenas “quemadas” o cadenas producto de una expresión.
- Las estructuras (element) siempre serán enviadas por referencia

Ejemplos válidos de cómo declarar una función se encontrarán en el siguiente bloque:

```
% Métodos válidos :)
num[][] : getTablero (num x, bool b, Objeto obj) {
      ¿ Nótese el formato que tiene el tipo de una
        función cuando devuelve un arreglo, además Objeto es el
      nombre del tipo de la estructura?
}

Nodo : getNodo(num valor, str * cadena) {
      % Instrucciones de getNodo
}

num : buscarElemento(num val, num datos [5..20]) {
      % Instrucciones de buscarElemento
}

void : metodo1() {
      % Instrucciones de metodo1
}
```

4.10.1. Método principal

Es como cualquier otro método, con la salvedad de que este método será el inicio de todo, no puede contar con parámetros y dentro de él vendrán todo tipo de instrucciones (incluyendo el `return`). Cualquier excepción encontrada dentro de este método producirá un error fatal por medio de una ventana emergente mostrando el detalle de la excepción. A continuación se detalla cómo será la declaración del método principal.

```
Principal() {
      % Instrucciones del método principal
}
```

4.11. Funciones primitivas

En lenguajes de programación como C existen funciones básicas agrupadas en librerías como stdio.h, mientras que Java cuenta con las funciones declaradas en java.lang.* importadas por defecto a cualquier programa desarrollado en este lenguaje; esto con el objetivo de ahorrar trabajo común a los desarrolladores, es en este punto donde Basic3D también define su conjunto de funciones nativas, útiles entre otras cosas para realizar conversiones explícitas, salidas en consola, excepciones, etc.

4.11.1. Conversiones

Para realizar el casteo explícito de un tipo de dato en otro, ya sea para su operación o su asignación directa se cuentan con las siguientes funciones propias del lenguaje.

`bool : getBool(str valor)`
Esta función toma su argumento y lo compara con la palabra “true”, si dicha comparación resulta cierta la función devolverá un valor true, en cualquier otro caso se devolverá false.

`num : getNum(str valor, str base, num default)`
Esta función trata de extraer de una cadena el valor numérico que representa según la base indicada en el segundo argumento, si en el proceso de conversión existe algún error la función retornará el valor por defecto que recibe como tercer parámetro. Las bases numéricas y el respectivo formato esperado se detalla en la siguiente tabla:

Base	Expresión	Ejemplos	Valor devuelto
“bin”	b[01]+(.[01]+)?	“b100.11“ “b10100”	4.75 20
“hex”	0x[A-Fa-f0-9]+	“0xFFF” “0x90BcdF”	4095 9485535
“dec”	(-)?[0-9]+(.[0.9]+)?	“1090” “-15.34”	1090 -15.34

4.11.2. Entradas y salidas

Estas funciones serán utilizadas para interactuar con el usuario, tanto para mostrar datos en consola o por medio de ventanas emergentes, pero también serán útiles para obtener información del usuario en tiempo de ejecución.

`void : outStr(str valor)`
Con esta función se imprimirá en consola el texto que está recibiendo como parámetro.

`void : outNum(num valor, bool comoEntero)`
Esta función imprimirá en consola el texto que representa los dígitos que componen el número que recibe como primer parámetro y al recibir segundo parámetro determinará si se desea mostrar como un número con punto decimal, si el segundo parámetro es falso; caso contrario lo imprimirá como un número entero.

`void : inStr(<id>, str msg)`
Esta función interrumpirá el flujo normal del programa para pedir una cadena al usuario, cuya longitud máxima será de 100 caracteres, a través de una ventana emergente (popup) mostrando como mensaje en dicha ventana el texto que recibe como segundo parámetro. El valor ingresado será almacenado en la variable identificada por el primer parámetro.

`num : inNum(str msg, num default)`
Se comporta igual que la función anterior, con la única diferencia que el dato que solicita al usuario es un número, si el usuario no indica un número válido en formato decimal la función retornará el valor que recibe como segundo parámetro.

`void : show(str msg)`
Esta función muestra al usuario una ventana emergente mostrando un mensaje con el texto que recibe como parámetro.

4.11.3. Otras funciones

`num : getRandom()`
Devuelve un número con punto decimal random entre 0 y 1 (incluyendo ambos extremos).

`num : getLength(<id>, num dimension)`
Devuelve la longitud del arreglo <id> en la dimensión indicada por el segundo parámetro.

`num : getLength(str cadena)`
Devuelve la longitud de la cadena que recibe como parámetro.

4.12. Excepciones

Las excepciones en los lenguajes de programación sirven para controlar situaciones fuera del flujo normal de un programa, o situaciones que el programador necesita reportar como un error mayúsculo. Para Basic3D se incorpora el manejo de las siguientes excepciones.

Nombre	Código	Descripción
NullPointerException	102	Cuando se trata de utilizar alguna estructura (element) que no ha sido instanciada o en otras palabras, es NULL.
MissingReturnStatement	243	Se presenta cuando en una función en donde se esperaba un retorno de cualquier tipo que no sea void NO se da ningún retorno.
ArithmeticException	396	Se presenta cuando se trata de hacer operaciones matemáticas indeterminadas, como una división entre 0.
StackOverFlowException	624	Serán lanzadas en tiempo de ejecución por medio del intérprete de código intermedio, se presentarán al momento de exceder el límite de alguna de las estructuras del entorno de ejecución.
HeapOverFlowException	789	
PoolOverFlowException	801	

Las excepciones pueden darse en tiempo de ejecución, ya que son situaciones fuera del control de quien escribió el programa y lanzadas por el intérprete de código intermedio. Sin embargo también pueden ser lanzadas por el programador de manera intencional llamando al método **`throws`** y enviándole como parámetro el nombre de la excepción a lanzar. Estas llamadas a excepciones colocadas por el usuario serán tratadas en código intermedio como llamadas a la función **`exit(código)`** en donde el código que reciban como parámetro será el código que se detalló en la tabla anterior. A continuación se encuentra un ejemplo de la instrucción para lanzar una excepción controlada por el programador.

```
% Sintaxis
throws(<nombre_excepcion>);
% Por ejemplo
throws(NullPointerException);
```

Al producirse una excepción acabará la ejecución del método en cuestión y devolverá un error al ámbito que lo ha llamado. Si la excepción ocurre en el método principal el programa acabará y deberá mostrar un mensaje de error grave en la pantalla. Todas las excepciones ocurridas durante la ejecución deberán ir al log de errores del programa. En el recuadro de la siguiente página se puede observar por medio de ejemplos la utilidad de las excepciones.

```
num : suma(num a, num b){
      num x = a + b;
      % Cualquier llamada a este método lanzará una excepción
      % de tipo MissingReturnException, puesto que no existe
      % ningún retorno, cuando se esperaba un valor num
}

num : division(num a, num b){
      if(b == 0){
            throws(ArithmeticException);
      }
      return a / b;
      % En este caso la ocurrencia de la excepción garantiza
      % que no se realiza la operación indeterminada de 'a'
      % dividido entre 0
}
```

5. El formato de código intermedio

El formato de código intermedio que se tendrá que generar es el código de tres direcciones, se le llama “de tres direcciones” porque todas sus sentencias deben tener a lo sumo tres posiciones en memoria o valores, es decir que cualquier operación compuesta debe ser “separada” en operaciones que cumplan con dicha premisa, en esta sección se describirán los detalles del código intermedio a generar:

5.1. Comentarios

Para mejorar la legibilidad, y poder realizar comprobaciones sobre el código de mejor manera se define la posibilidad de agregar comentarios al código intermedio, estos comentarios pueden ser de una sola línea, empezarán con `//` y terminarán con un salto de línea; o bien pueden ser comentarios multilínea, que inicien con `/*` y terminen con `*/`.

Ejemplos:

```
// Este es un comentario de una línea

/* Este en cambio, es
      un comentario
      múltiples
      líneas
*/
```

5.2. Temporales

Un temporal o variable auxiliar es una variable creada en la generación de código de tres direcciones para almacenar el resultado de una operación. El nombre propio de un temporal está compuesto por una letra t seguida de cualquier número entero. Es necesario tomar en cuenta que el índice que da nombre a los temporales debe aumentar independientemente de los métodos en los que los temporales sean utilizados.

Ejemplo:

```
void metodo_1(){
      t1 …
      t2 …
      t3 …
      t4 …
}

void metodo_2(){
      t5 …
      t6 …
      t7 …
}
```

5.3. Etiquetas

Una etiqueta es la identificación de un lugar específico dentro del código de tres direcciones, la cual puede trasladar la ejecución del código por medio de algún salto de manera explícita. Una etiqueta está compuesta por la letra L seguida de cualquier número entero. Es necesario tomar en cuenta que el contador debe aumentar independientemente de los métodos.

Ejemplo:

```
void metodo_1(){
      L1:
            //instrucciones
      L2:
      L3:
            //instrucciones
}
void metodo_2(){
      L4:
            //más instrucciones
}
```

5.4. Proposiciones de asignación

Las proposiciones de asignación serán utilizadas para poder asignar un valor a una variable (temporal o puntero). Las proposiciones de asignación tendrán la siguiente sintaxis:

Asignación con un operador aritmético

```
<id> = <id_o_valor> <operador> <id_o_valor>;
```

Asignación simple

```
<id> = <id_o_valor>;
```

Donde:

- **<id>** es el destino del valor que se encuentra al lado derecho de la asignación
- **<id_o_valor1>** y **<id_o_valor2>** pueden ser temporales, punteros, o valores fijos
- **<operador>** debe ser exclusivamente un operador ARITMÉTICO

Ejemplo:

```
t1 = 1990.0620;
t2 = 10;
t3 = t1 + t2;
t4 = 5 * t4;
```

NOTA: En las asignaciones (simples o con operaciones) también se puede consultar o modificar los valores almacenados en las estructuras propias del entorno de ejecución y también sus respectivos punteros.

5.5. Operaciones aritméticas

Las operaciones aritméticas que se realizarán dentro del código de tres direcciones serán las operaciones más básicas, las cuales se describen en la siguiente tabla:

Operación	Símbolo
Suma	+
Resta	-
Multiplicación	*
División	/
Módulo	%

5.6. Operaciones relacionales

Las operaciones relacionales que se realizarán dentro del código de tres direcciones serán las operaciones más básicas, las cuales se describen en la siguiente tabla:

Operaciones	Símbolo
Comparación	==
Diferencia	!=
Mayor	>
Mayor o igual	>=
Menor	<
Menor o igual	<=
Verdadero	1==1
Falso	1==0

5.7. Operaciones lógicas

El código intermedio de las operaciones lógicas será escrito en formato de corto circuito. Por lo tanto NO es posible utilizar los operadores que normalmente se utilizan para dichas operaciones.

5.8. Salto incondicional

Este salto **NO** depende del cumplimiento de una condición para poder realizarse. El salto incondicional sigue la siguiente sintaxis:

```
goto <etiqueta>;
```

Ejemplo:

```
void metodo_1(){
    L1:
    t1 = 2;
    t2 = 5.89;
    goto L2; // Este es un salto incondicional
    t3 = S + 0;
    t4 = 10;
    t5 = 3.14 * t4;
    L2:
    //Instrucciones para la etiqueta L2
}
```

5.8. Salto condicional

La ejecución de este salto, depende del cumplimiento de la condición señalada por el `if`, si dicha condición se cumple, el salto SI debe realizarse, de lo contrario, la ejecución continuará en la siguiente línea de código. El salto condicional sigue la siguiente sintaxis.

Ejemplo:

```
if (<id_o_valor1> <op_rel> <id_o_valor2>) goto <etiqueta>;
```

Donde:

- **<op_rel>** representa cualquiera de los operadores relacionales
- **<id_o_valor1>** y **<id_o_valor2>** pueden ser temporales, punteros, o valores fijos
- **<etiqueta>** representa el lugar dentro del código al cual se trasladará la ejecución en caso de que sí se cumpla la condición

5.9. Declaración de métodos

Dentro del código de tres direcciones un método será declarado por medio de la siguiente sintaxis, cabe resaltar que TODOS los métodos son de tipo **void** y se permiten llamadas recursivas.

```
void identificador(){
    //Instrucciones en código de 3 direcciones
}
```

Ejemplo:

```
void metodo_1(){
      //Instrucciones en código de 3 direcciones
}
```

5.10. Llamadas a métodos

Mientras que para las llamadas a métodos se utilizará la siguiente sintaxis, en este nivel, los métodos no admiten parámetros, ya que los mismos se trasladan por medio del Stack.

```
<id_metodo>();
```

Ejemplo:

```
void metodo_1(){
      // Esta es una llamada al metodo_2 dentro de metodo_1
      metodo_2();
}

void metodo_2(){
      // Instrucciones en código de 3 direcciones
}
```

5.11. Funciones nativas del código intermedio

Como en todo lenguaje de programación, para soportar las funciones primitivas del lenguaje de alto nivel es necesario definir un core de funciones nativas (al estilo de stdio.h de C).

5.11.1. Core

Basic3D incorpora un repertorio de funciones nativas de código de tres direcciones para apoyar al lenguaje de alto nivel, el desarrollo del cuerpo de las instrucciones de este core queda a discreción del estudiante, pero deben tener los siguientes nombres:

Función de Basic3D	Función nativa	Parámetros en la pila
bool : getBool(str)	$$_getBool()	[0 | retorno (0 = F, 1 = V) [1 | referencia a la cadena a convertir]
num : getNum(str, str, num)	$$_getNum()	[0 | retorno, valor numérico] [1 | referencia a la base] [2 | referencia a la cadena a convertir] [3 | valor numérico por defecto]
void : outStr(str)	$$_outStr()	[0 | retorno, no se usa] [1 | referencia a la cadena a mostrar]

void : outNum(num, bool)	$$_outNum()	[0 | retorno, no se usa] [1 | valor numérico] [2 | valor booleano (0 = F, 1 = V)]
void : inStr(id, str)	$$_inStr()	[0 | retorno, no se usa] [1 | referencia donde guardar el valor] [2 | referencia a la cadena a mostrar]
num : inNum(str, num)	$$_inNum()	[0 | retorno, valor numérico] [1 | referencia a la cadena a mostrar] [2 | valor numérico por defecto]
void : show(str)	$$_show()	[0 | retorno, no se usa] [1 | referencia a la cadena a mostrar]
num : getRandom()	$$_getRandom	[0 | retorno, valor numérico]
num : getLength(id, num)*	$$_getArrLength	[0 | retorno, valor numérico] [1 | referencia al arreglo] [2 | valor numérico, dimensión]
num : getLength(str)	$$_getStrLength	[0 | retorno, valor numérico] [1 | referencia de la cadena a analizar]

Nota: * Como no se pueden redefinir los arreglos es posible que las llamadas a esta función sean reemplazadas en tiempo de compilación por el valor consultado directamente desde la tabla de símbolos, asumiendo que el segundo parámetro únicamente puede ser un valor numérico (entero) primitivo, es decir, no expresiones o variables.

Para llamar a cualquiera de estas funciones es necesario cargar previamente los parámetros en la pila para seguidamente proceder a realizar la llamada a la función y luego sustraer el valor retornado según sea el caso.

5.11.2. Printf

El código de 3 direcciones contará con una función para poder imprimir salidas en consola, para ello se utilizará la palabra clave “printf” con la sintaxis que se muestra a continuación:

```
printf(<param> , <id_o_valor>);
```

Donde:

- **<param>** indicará el formato en que se mostrará el valor
- **<id_o_valor>** representa el valor que se mostrará

Parámetro	Formato según el valor de param
%c	Indica que el valor a imprimir será un carácter.
%d	Indica que el valor a imprimir será un entero.
%f	Indica que el valor a imprimir será un número con punto flotante.

5.11.3. Exit

Este método es la forma en que se traducen las excepciones al código intermedio. Las llamadas a este método tienen un código que indica cuál es el error que se ha suscitado, al finalizar la ejecución de todo el programa debe llamarse a este método con un 0 como valor de su parámetro, esto indicará que la ejecución ha terminado correctamente.

6. Entorno de ejecución

Como ya se expuso en otras secciones de este documento, en el código intermedio **NO** existen cadenas, operaciones complejas, llamadas a métodos con parámetros y muchas otras cosas que sí existen en los lenguajes de alto nivel. Esto debido a que el código intermedio busca ser una representación próxima al código máquina, por lo que todas las sentencias de las que se compone se deben basar en las estructuras que componen el entorno de ejecución. Típicamente se puede decir que los lenguajes de programación cuentan con dos estructuras para realizar la ejecución de sus programas en bajo nivel, la pila (Stack) y el montículo (Heap), en la siguiente sección se describen estas estructuras .

6.1. Estructuras del entorno de ejecución

Las estructuras del entorno de ejecución no son más que arreglos de bytes que emplean ingeniosos mecanismos para emular las sentencias de alto nivel. En este proyecto las estructuras de control serán representadas por arreglos de números con punto flotante, esto con el objetivo de que se pueda tener un acceso más rápido a los datos sin necesidad de leer por grupos de bytes y convertir esos bytes al dato correspondiente, como normalmente se hace en otros lenguajes, por ejemplo en Java, un int ocupa 4 bytes, un boolean ocupa 1 solo byte, un double ocupa 8 bytes, un char 2 bytes, etc.

6.1.1. El Stack y su puntero

El stack es la estructura que se utiliza en código intermedio para controlar las variables locales, y la comunicación entre métodos (paso de parámetros y obtención de retornos en llamadas a métodos). Se compone de ámbitos, que son secciones de memoria reservados exclusivamente para cierto grupo de sentencias.

Cada llamada a método o función que se realiza en el código de alto nivel cuenta con un espacio propio en memoria para comunicarse con otros métodos y administrar sus variables locales. Esta estructura recibe su nombre debido al comportamiento que presenta para la administración de los ámbitos, puesto que utiliza la política LIFO(Last In First Out) para apilar los ámbitos uno sobre otro y conforme la ejecución de un método termina se va liberando el espacio reservado para éste, eliminando el ámbito que se encuentra en la cima de la pila. Más información de LIFO en https://es.wikipedia.org/wiki/Last_in,_first_out.

Esto se logra modificando el puntero del Stack, que en el proyecto se identifica con la letra P, para ir moviendo el puntero de un ámbito a otro, cuidando de no corromper ámbitos ajenos al que se está ejecutando. A continuación se ilustra su funcionamiento con un ejemplo.

Stack P=17

POSICIÓN	VALOR
0	ámbito de Principal
1	ámbito de Principal
2	ámbito de Principal
3	ámbito de Principal
4	ámbito de Principal
5	ámbito de Principal
6	ámbito de Principal
7	ámbito de Principal
8	ámbito de Principal
9	ámbito de Principal
10	ámbito de Principal
11	ámbito de Principal
12	ámbito de Buscar
13	ámbito de Buscar
14	ámbito de Buscar
15	ámbito de Buscar
16	ámbito de Buscar
17	ámbito de Comparar
18	ámbito de Comparar
19	ámbito de Comparar
20	ámbito de Comparar

```
% Código de alto nivel

Principal() {
      % Se llena un element llamado árbol
      % Se llama al método buscar
}

Nodo : BuscarHoja(num llave, Nodo raiz) {
      % Busca el elemento, si el nodo
      % es un nodo hoja, la búsqueda
      % realiza una llamada al
      % método Comparar
}

bool : Comparar(num valor, num llave) {
      % Se realiza la comparación y se
      % retorna el resultado de la misma
}
```

Explicación:
En la imagen de la izquierda se puede observar una representación gráfica de cuál sería el estado del Stack al momento de estar ejecutando el método comparar si el árbol tuviera como único nodo la raíz misma.
El ámbito del método **`Principal`** está en el "fondo" de la pila, mientras que el ámbito del método **`Comparar`** (en ejecución) se encuentra en la "cima" de la pila, próximo a ser eliminado. El valor del puntero indica el ámbito en el que se encuentra la ejecución.

Cabe resaltar que por su misma naturaleza (LIFO) el Stack se reutiliza por lo que una buena práctica es limpiar el área que ya no se utilizará, colocando valores nulos en todo el espacio que ha dejado libre la finalización de un método, para ello se cuenta con la instrucción nativa.

`$$_SGB(num inicio, num longitud)`
Se debe llamar al terminar una llamada a un método y se le deben enviar como parámetros el valor de inicio de un ámbito y el tamaño del mismo para realizar la limpieza del espacio ocupado por el método. Esta instrucción no será programada en el core de funciones nativas, más sí tendrá que ser ejecutada por el intérprete de código de tres direcciones. La colocación de esta instrucción debe estar en la propuesta del código intermedio que realice el estudiante para las llamadas a métodos, como sugerencia se podría tener la siguiente plantilla para las llamadas a métodos o funciones.

```
tx = P + K;          % Simulación de cambio de ámbito, K = tamaño
. . .                % Carga de parámetros (si aplica)
. . .                % Sigue la carga de parámetros
. . .                % Termina la carga de parámetros
P = P + K;           % Cambio real del ámbito
```

```
metodo_funcion();    % Llamada al método
. . .                % Obtención del retorno (si aplica)
P = P - K;           % Regreso del ámbito
$$_SGC(tx, K);       % Llamada a SGC para limpiar el ámbito terminado
```

6.1.2. El Heap y su puntero

El Heap (o en español, montículo) es la estructura de control del entorno de ejecución encargada de guardar las estructuras o referencias a variables globales. Esta es una estructura que también puede almacenar basura digital (datos inalcanzables) pero la implementación de un garbage collector para este tipo de estructuras requiere un esfuerzo mucho mayor que no realizaremos en este proyecto.

El puntero H, a diferencia de P (que aumenta y disminuye según lo dicta el código intermedio), solo aumenta y aumenta, su función es brindar siempre la próxima posición libre de memoria. A continuación se muestra un ejemplo de cómo se vería el Heap después de almacenar dos estructuras (element), una de tipo **`Arbol`** y otra de tipo **`Nodo`**, además de poder ver que la próxima posición que nos brindaría el Heap para almacenar las estructuras sería la 12, el valor de su puntero H.

Heap H=12

POSICIÓN	VALOR
0	Árbol:arbolito
1	Árbol:arbolito
2	Árbol:arbolito
3	Árbol:arbolito
4	Árbol:arbolito
5	Árbol:arbolito
6	Árbol:arbolito
7	Nodo:Raíz
8	Nodo:Raíz
9	Nodo:Raíz
10	Nodo:Raíz
11	Nodo:Raíz

```
¿ Los elements que podrían estar
representados  en ese Heap podrían ser?
element : Arbol {
      num nodos = 0;
      num balance = 0;
      str titulo = "Arbolito";
      str creador = "El auxiliar";
      Nodo raiz = NULL;
      str cadena = "Cadena de prueba";
      bool esAVL = false;
}

element : Nodo {
      num valor = 10;
      str cadena = "Nodo N";
      str padre = "Otro nodo";
      Nodo izq = NULL;
      Nodo der = NULL;
}
```

Llegados a este punto, puede que exista el cuestionamiento de ¿Dónde están almacenadas las cadenas de texto? para eso en este proyecto se implementará una estructura adicional en el entorno de ejecución, una estructura llamada String Pool, cuyo único objetivo es el de almacenar cadenas de texto, esta estructura almacenará exclusivamente los códigos ascii de cada uno de los caracteres que componen una cadena. En la siguiente subsección se describe el uso y otros aspectos de esta estructura.

6.1.3. El String Pool y su puntero

Esta estructura está dedicada a almacenar únicamente cadenas de caracteres (texto) cada cadena almacenada debe terminar con el caracter de final de cadena '\0', al que le corresponde el código ascii 0, así pues se tendrá un forma fácil de observar las cadenas almacenadas desde la vista de debug.

El String Pool será accesible por medio del identificador Pool y su puntero estará identificado por la letra S y éste, al igual que H, solo podrá ir en aumento, otorgando siempre la próxima posición disponible en memoria. Para almacenar la cadena "USAC" se debería generar algo como lo siguiente:

```
% Almacenando USAC
tx = H;              % Copia de H
H = H + 1;           % Aumento de H
Heap[tx] = S;        % Almacenando referencia inicial a String Pool
Pool[S] = 85;        % Copia 'U', ascii 85
S = S + 1;           % Aumenta S
Pool[S] = 83;        % Copia 'S', ascii 83
S = S + 1;           % Aumenta S
Pool[S] = 65;        % Copia 'A', ascii 65
S = S + 1;           % Aumenta S
Pool[S] = 67;        % Copia 'C', ascii 67
S = S + 1;           % Aumenta S
Pool[S] = 0;         % Copia del caracter de fin '\0', ascii 0
S = S + 1;           % Aumenta S
```

En el manejo de las cadenas siempre es importante realizar una **"doble referencia"**, esto quiere decir que cualquier cadena siempre tendrá primero una posición en el Heap, en la que pueden almacenarse dos posibles cosas, se almacenará la posición de inicio de la cadena real (el primer caracter) dentro del String Pool o se almacenará NULL, con lo primero se garantiza de que toda cadena tenga una referencia válida, y con lo segundo se garantiza que dicha referencia pueda apuntar a NULL que sería el caso en el que no se ha inicializado la cadena o bien se hizo una asignación de **`cadena=NULL`**.

Después de realizar varias declaraciones de cadenas se desea que el String Pool sea visible en el debug de una manera similar a como se muestra en la siguiente imagen.

String Pool S=60

POSICIÓN	VALOR
0	Esta cadena ocupa 29 espacios
30	Cadena de prueba
46	Compiladores 2
60	

6.2. Acceso a estructuras del entorno de ejecución

Para asignar un valor a una estructura del sistema es necesario colocar el identificador del arreglo (Stack, Heap o Pool), la posición donde se desea colocar el valor seguido del valor a asignar con la siguiente sintaxis:

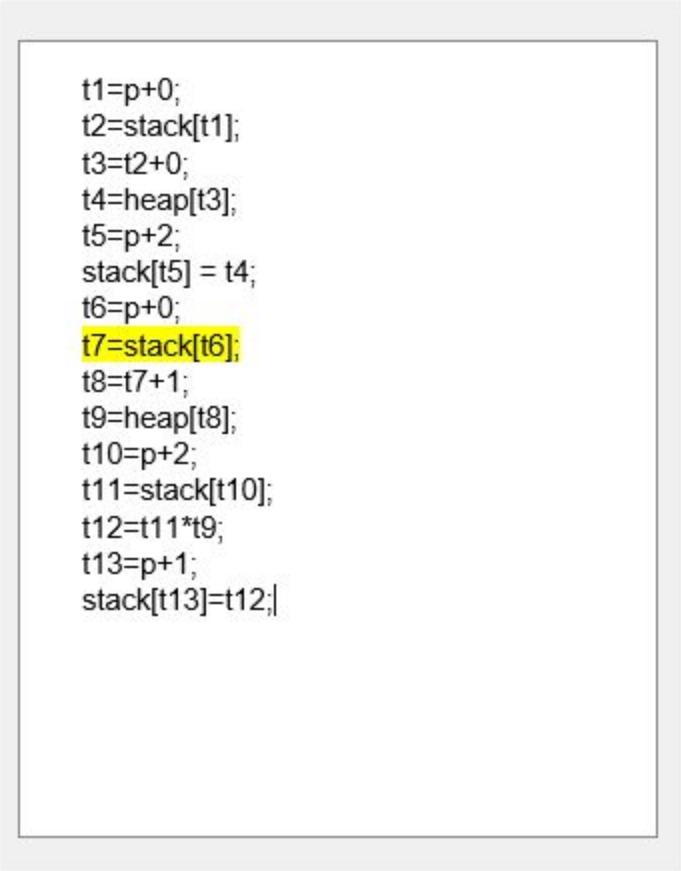
```
Stack[id_o_valor] = id_o_valor;
// o bien
Heap[id_o_valor] = id_o_valor;
// o bien
Pool[id_o_valor] = id_o_valor;
```

Para la obtención de un valor de cualquiera de las estructuras (Stack, Heap o Pool) se realizará con la siguiente sintaxis:

```
id = Stack[id_o_valor];
// o bien
id = Heap[id_o_valor];
// o bien
id = Pool[id_o_valor];
```

7. Debugger

El IDE de Basic3D contará con un módulo de depuración o debugger. Este módulo permitirá ejecutar de manera controlada las instrucciones en 3 direcciones que genere Basic3D en su salida. Para poder conocer la instrucción exacta que se está ejecutando, el debugger deberá resaltar la línea que se está ejecutando.



El debugger deberá mostrar durante la ejecución el estado de las estructuras de control (Stack, Heap y String Pool) de manera gráfica. Mostrando la posición y el contenido de los arreglos.

Para poder visualizar el correcto funcionamiento del debugger, este contará con 5 opciones o características importantes, las cuales permitirán la ejecución, pausa y aceleración e esta característica de la herramienta. Estas opciones se describen a continuación

7.1. Inicio/Play

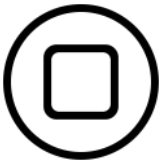
Característica del debugger que permite iniciar o reanudar la ejecución (animación) del debugger.

7.2. Pausa

Característica que permite pausar la ejecución (animación) del debugger en su estado actual.

7.3. Detener/Stop

Permite detener la ejecución del debugger de manera definitiva.



7.4. Velocidad

Permite determinar a qué velocidad se ejecutará el debugger. Esta opción permitirá aumentar o disminuir la velocidad de ejecución del debugger.

7.4.1. Aumento de velocidad

El aumento de velocidad será representado por la siguiente imagen:

7.4.2. Disminución de velocidad

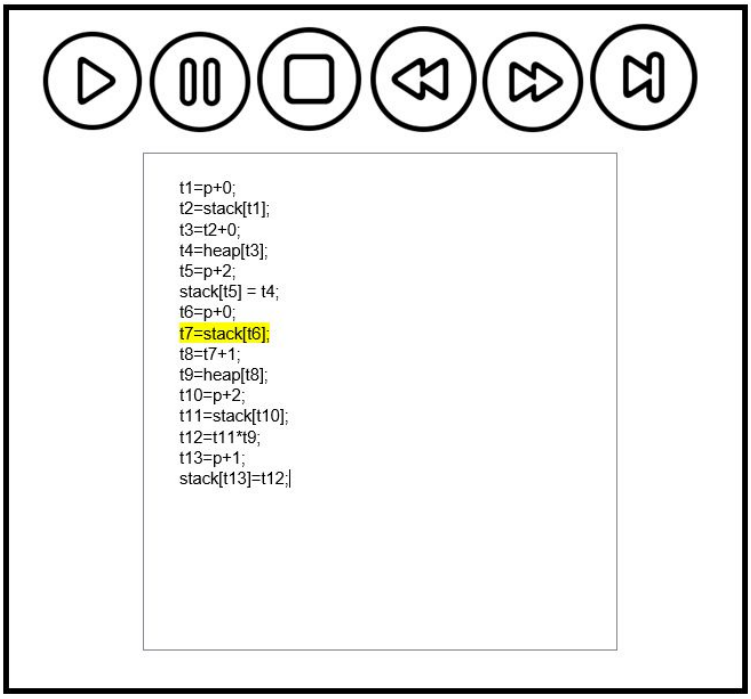
La disminución de velocidad será representada por la siguiente imagen:

7.5. Siguiente instrucción

Característica que permite saltar a la siguiente instrucción cuando el debugger se encuentre en pausa.

Nota: Las imágenes de los iconos son ejemplos ilustrativos, por lo que pueden utilizarse imágenes similares o diferentes pero que representen lo mismo

A continuación se ilustra la interfaz recomendada para el debugger dentro de la herramienta gráfica de Basic3D.



.

8. Optimización de código intermedio

La optimización de código consiste en mejorar el rendimiento y la eficiencia del programa compilado. La optimización de código debe realizarse mediante la unión de la optimización por bloques y la optimización por mirilla.

Primero se deberá aplicar las reglas de optimización de flujo de control sobre todo el código de entrada, luego se realizará la unión de las optimizaciones (bloques y mirilla) aplicando el concepto de construcción de bloques, luego la optimización se hará utilizando unas de las reglas de manera local de la optimización por bloques y para cada bloque se aplicarán las reglas de la optimización por mirilla

Para la construcción de los bloques, se debe considerar los líderes de los bloques que no son más que la instrucción con la que se comienza dentro del bloque, las reglas para buscar líderes son:

1. La primera instrucción de tres direcciones en el código intermedio
2. Cualquier instrucción que sea el destino de un salto condicional o incondicional
3. Cualquier instrucción que siga justo después de un salto condicional o incondicional

Con las reglas establecidas se puede establecer que para cada instrucción líder, su bloque básico consiste en sí misma y en todas las instrucciones hasta, pero sin incluir a la siguiente instrucción líder o el final del programa intermedio (Ver sección 8.4 del libro de clase).

Como principio de la optimización por bloques se deberán establecer relaciones entre los bloques, las cuales consisten en apuntadores de un bloque a otro y que existen para ver el flujo que contiene la ejecución del código intermedio.

Después de tener establecido los bloques, se debe aplicar las transformaciones de **manera local** sobre los bloques y obtener como salida el código optimizado. Las transformaciones a considerar son 3, dentro de las cuales se definirán 16 reglas para optimizar el código.

- Eliminación de subexpresiones comunes y propagación de copias
- Simplificación algebraica
- Reducción por fuerza

8.1. Eliminación de subexpresiones comunes

Regla No. 1

Para eliminar una subexpresión, esta debió ser calculada previamente y los valores que integran la subexpresión no deben cambiar desde el momento que es calculada hasta que se utilice. Ejemplo:

Código generado	Código optimizado

`a = b + c`
`b = a - d`
`c = b + c`
`d = a - d`
	`a = b + c`
	`b = a - d`
	`c = b + c`
	`d = b`

8.2. Propagación de copias

Regla No. 2

La regla para propagación de copias permitirá eliminar código inactivo o redundante :

Código generado	Código optimizado

`t1 = i + 4`
`i = i + 4`
`t2 = j - 1`
`j = j -1`
	`t1 = i + 4`
	`i = t1`
	`t2 = j - 1`
	`j = t2`

8.3. Simplificación algebraica

Un optimizador puede utilizar identidades algebraicas para eliminar las instrucciones ineficientes. Para esta transformación definimos las siguientes reglas de optimización:

Regla No. 3

Se eliminará la instrucción que cumpla con el siguiente formato:

```
x = x + 0; % Donde x es cualquier identificador
% ó bien
x = 0 + x; % Donde x es cualquier identificador
```

Dado que no cumple ninguna funcionalidad dentro del código.

Regla No. 4

Se eliminará la instrucción que cumpla con el siguiente formato:

```
x = x - 0; % Donde x es cualquier identificador
```

Dado que no cumple ninguna funcionalidad dentro del código.

Regla No. 5

Se eliminará la instrucción que cumpla con el siguiente formato:

```
x = x * 1; % Donde x es cualquier identificador
% ó bien
x = 1 * x; % Donde x es cualquier identificador
```

Dado que no cumple ninguna funcionalidad dentro del código.

Regla No. 6

Se eliminará la instrucción que cumpla con el siguiente formato:

```
x = x / 1; % Donde x es cualquier identificador
```

Dado que no cumple ninguna funcionalidad dentro del código.

Puede que existan instrucciones como las anteriores pero aplicadas con diferentes variables en el destino y en la expresión del lado derecho de la asignación, en este caso la operación se descarta y se convierte en una asignación. Para estos casos se definen las siguientes reglas de optimización:

Regla No. 7

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o identificador
x = y + 0;
% ó bien
x = 0 + y;
% Se sustituirá por:
x = y;
```

Aplicando la propiedad del elemento neutro de la suma.

Regla No. 8

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o identificador
x = y - 0;
% Se sustituirá por:
x = y;
```

Aplicando la propiedad del elemento neutro de la resta.

Regla No. 9

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o identificador
x = y * 1;
% ó bien
x = 1 * y;
% Se sustituirá por:
x = y;
```

Aplicando la propiedad del elemento neutro de la multiplicación.

Regla No. 10

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o identificador
x = y * 0;
% ó bien
x = 0 * y;
% Se sustituirá por:
x = y;
```

Aplicando la propiedad del producto cero.

Regla No. 11

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o identificador
x = y / 1;
% Se sustituirá por:
x = y;
```

Aplicando la propiedad del elemento neutro de la división.

Regla No. 12

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o identificador
x = 0 / y;
% Se sustituirá por:
x = 0;
```

Aplicando la propiedad del cociente cero.

Regla No. 13

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o identificador
x = y ^ 0;
% Se sustituirá por:
```

```
x = 1;
```

Aplicando la propiedad del exponente 0.

Regla No. 14

Si existe una instrucción que cumpla con la forma siguiente:

```
% Donde y es cualquier valor o identificador
x = y ^ 1;
% Se sustituirá por:
x = y;
```

Aplicando la propiedad del exponente 0.

8.4. Reducción por fuerza

De manera similar puede aplicarse una transformación de reducción por fuerza para sustituir operaciones costosas por expresiones equivalentes relativamente más económicas. Para esta clasificación, se definen las siguientes reglas de optimización:

Regla No. 15

Si existe una instrucción de la forma:

```
% Donde y es cualquier valor o identificador
x = y ^ 2;
% Se sustituirá por:
x = y * y;
```

Regla No. 16

Si existe una instrucción de la forma:

```
% Donde y es cualquier valor o identificador
x = y * 2;
% Se sustituirá por:
x = y + y;
```

8.5. Optimizaciones de flujo de control

Como se mencionó, éstas deberán aplicarse antes de la construcción de bloques y la aplicación de las otras reglas. Ésta optimización surge con la existencia de saltos innecesarios que pueden eliminarse, por lo que se definen las siguientes reglas:

Regla No. 17

Si existe un salto incondicional de la forma *goto Lx* y existe la etiqueta *Lx:* y la primera instrucción después de la etiqueta es otro salto, de la forma *goto Ly* podemos modificar el primer salto para que haga referencia a la etiqueta *Ly:* y de esta forma ahorrar la ejecución de un salto

Código generado	Código optimizado
goto L1; **<instrucciones>** **L1: goto L2;**	**goto L2;** **<instrucciones>** **L1: goto L2;**

Regla No. 18

Si existe un salto condicional de la forma *if<cond> goto Lx;* y la primera instrucción después de la etiqueta es otro salto, de la forma goto *Ly* podemos modificar el primer salto para que haga referencia a la etiqueta *Ly:* y de esta forma ahorrar la ejecución de un salto.

Código generado	Código optimizado
if t1<t2 goto L1; **<instrucciones>** **L1: goto L2;**	**if t1<t2 goto L2;** **<instrucciones>** **L1: goto L2;**

9. Generación de código ensamblador

La herramienta gráfica de Basic3D contendrá un módulo externo encargado de traducir el lenguaje en formato de tres direcciones que genere en sus salida. Este módulo externo será una aplicación que tendrá como tarea la traducción el código de tres direcciones (visto en la sección 5) en código ensamblador.

La salida de la herramienta gráfica de Basic3D deberá enviarse a la aplicación como código de entrada. Como el lenguaje ensamblador es un lenguaje de bajo nivel, se podrá evaluar el código intermedio por medio de un emulador y garantizar que la salida en tres direcciones es correcta.

9.1. Entrada de ASM

Como se mencionó anteriormente el código de entrada para la aplicación de escritorio será el código en tres direcciones generado por la herramienta gráfica Basic3D. Esta entrada deberá enviarse de forma similar a la arquitectura cliente servidor pero con comunicación en una sola vía o unidireccional, y no como una simple entrada de texto plano.
Nota: Queda a criterio del estudiante la forma en que se envía el código de una aplicación a otra.

9.2. Salida de ASM

La aplicación de escritorio traducirá el código de entrada formato de tres direcciones generada por Basic3D en su equivalente para código Assembler 8086. Este código generado deberá ser ejecutado en un emulador (se recomienda Dosbox para realizar pruebas).

9.3. Formato de código ensamblador

9.3.1. Instrucciones aritméticas

El formato para las funciones aritméticas se representa de la siguiente manera:

```
MOV registro1, valor1
MOV registro2, valor2
<instruccion> registro1, registro2
```

Donde `MOV` es la instrucción de assembler para realizar la copia de un registro dentro de otro. Y dentro del apartado <instruccion> se hace referencia a instrucciones aritméticas propias de assembler como por ejemplo `ADD` para sumar o `SUB` para restar.

9.3.2. Etiquetas

De la misma manera que el código de tres direcciones mencionado en la sección 5.3, el lenguaje ensamblador posee etiquetas que identifican el inicio de instrucciones de código a ejecutar si se encuentra posicionado en ese lugar.

```
etiqueta:
```

9.3.3. Saltos incondicionales

Este salto NO depende del cumplimiento de una condición para poder realizarse. El salto incondicional sigue la siguiente sintaxis:

```
JMP <destino>
```

9.3.4. Saltos condicionales

Este salto depende del cumplimiento de una condición para poder realizarse. El salto condicional sigue la siguiente sintaxis:

```
MOV registro1, valor1
MOV registro2, valor2
CMP registro1, registro2
<instruccion de salto condicional> destino
```

Donde CMP realiza una comparación entre ambos registros para este ejemplo. Es importante destacar que existen muchas formas de comparar registros. A continuación se describe una tabla con algunos de los saltos condicionales existentes en el lenguaje Assembler.

Instrucción	Descripción
JE/JZ	Jump Equal or Jump Zero
JNE/JNZ	Jump not Equal or Jump Not Zero

JG/JNLE	Jump Greater or Jump Not Less/Equal
JGE/JNL	Jump Greater/Equal or Jump Not Less
JL/JNGE	Jump Less or Jump Not Greater/Equal
JLE/JNG	Jump Less/Equal or Jump Not Greater

9.3.5. Llamadas a métodos y funciones

Para poder realizar una llamada a un método o función dentro del lenguaje ensamblador es necesario utilizar la palabra `CALL` seguido del nombre del método o función.

```
CALL <nombre metodo o funcion>
```

9.3.6. Declaración de métodos y funciones

Dentro del lenguaje ensamblador la declaración de métodos y funciones se realiza mediante la palabra reservada `PROC`, seguida del identificador y las instrucciones a ejecutar.

```
PROC <nombre metodo o funcion>
;instrucciones
:para detener el flujo de una subrutina utilizar la
;instrucción RET
ENDP
```

9.3.7. Print

Para poder mostrar en pantalla un valor, se utilizará la interrupción 21h.

9.3.8. Comentarios

El lenguaje ensamblador reconoce comentarios, estos inician con punto y coma `(;)` seguidos de cualquier valor.

```
;esto es un comentario
```

Nota: Es importante destacar que esta sección únicamente tiene algunas instrucciones básicas del lenguaje ensamblador, por lo que es necesario que el estudiante investigue las instrucciones que necesite aplicar en su solución.

10. Reportes

Producto de la realización de los diferentes procesos del IDE Basic3D se generan distintas salidas que sirven para verificar el funcionamiento correcto de todo el proceso de compilación (o detectar los posibles errores que hayan ocurrido).

10.1. Reporte de tabla de símbolos

Se debe generar el reporte de la tabla de símbolos al finalizar el proceso de transformar el código de alto nivel a código de tres direcciones. La tabla se deberá generar en un archivo html, el cual se tiene que abrir en una nueva pestaña. Se recomienda que el contenido de la tabla sea como mínimo:

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

Nota: En el caso de los arreglos debe añadirse una columna que guarde una lista de las dimensiones que contiene. Además queda criterio del estudiante agregar la información adicional que considere necesaria para que la tabla de símbolos le dé el soporte necesario durante la ejecución del código.

10.2. Reporte de optimización

Después de realizar una optimización, se deberá poder consultar el reporte de optimización con el fin de evaluar las reglas de transformación aplicadas. El reporte deberá estar seccionado según los bloques que lo compongan y deberá mostrar el código original de cada bloque y luego el código optimizado del mismo, también deberá contener el conjunto de reglas aplicadas (si una regla se usa más de una vez, debe mostrarse en el reporte imprimiendola las veces que se aplique). Este reporte deberá visualizarse de la siguiente manera:

Código original	Código optimizado	Reglas aplicadas	No. Bloque

`L1:`
`t1 = t2 + t3;`
`t2 = t1 - t4;`
`t3 = t2 + t3;`
`t4 = t1 - t4;`
`goto L2;`
	`L1:`
	`t1 = t2 + t3;`
	`t2 = t1 - t4;`
	`t3 = t2 + 13;`
	`t4 = t2;`
	`goto L2;`
		Regla No.1	Bloque 1

`L4:`
`t1 = t20;`
`t6 = t6+0;`
`t20 = t1;`
`t8 = t8+0;`
	`L4:`
	`t1=t20;`
		Regla No. 3
		Regla No. 2
		Regla No. 3
			Bloque 15

10.3. Reporte de errores

El reporte de errores deberá incluir como mínimo una descripción clara del error encontrado, la fila, la columna y el tipo de error. Este reporte deberá mostrar:

- Errores léxicos

- Errores sintácticos
- Errores semánticos (que a su vez incluye de comprobación de tipos)
- Errores en tiempo de ejecución (excepciones no programadas por el usuario)
- Errores provocados por el lanzamiento de excepciones (colocadas por el usuario)
- Errores generales (cualquier error que no se esté en las categorías anteriores)

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
Excepción	En la llamada a num:suma(num, num) no se obtuvo ningún retorno	132	1
Excepción	NullPointerException, controlada por el usuario	140	20

Los errores que se deben reportar como excepción, son los errores que ocurran durante la ejecución del programa, como el desbordamiento de la pila (StackOverFlowException), una operación matemática indeterminada no controlada por el usuario (ArithmeticException), un acceso a alguna propiedad de un element que no ha sido instanciado (NullPointerException), y también se incluyen como excepciones las llamadas intencionales al método throws creadas por el usuario.

No son excepciones, por lo tanto no tienen que ser reportadas como tal, los errores léxicos, sintácticos o semánticos encontrados durante la fase de análisis y en la generación de código intermedio.

11. Consideraciones finales

Para el desarrollo de este proyecto se deben tener en cuenta las siguientes consideraciones o restricciones.

11.1. Restricciones

- El proyecto será desarrollado de forma individual
- No se permiten copias
- El proyecto será desarrollado con el lenguaje de programación JavaScript
- El generador de analizadores a utilizar será Jison
    - Sitio oficial https://zaa.ch/jison/
    - Implementación de Bison para JavaScript
    - Utilizado por frameworks como CoffeScript
- La API para implementar el drag-and-drop será Blockly
    - Sitio oficial https://developers.google.com/blockly/
    - Desarrollada por Google
    - Utilizada por proyectos como Scratch
- El proyecto puede realizarse en JavaScript puro, pero el estudiante es libre de utilizar cualquier framework (o librería) de JavaScript para facilitar el desarrollo de la aplicación; opciones como Angular.js, node.js, React.js, JQuery u otra son totalmente permitidas
- El IDE queda a discreción del estudiante
- El Sistema Operativo también queda a discreción del estudiante
- Es deseable que en el modo de edición avanzada se incorporen características para facilitar la escritura de programas, tales como resaltado de sintaxis o autocompletado de instrucciones y/o palabras reservadas.
- La aplicación de escritorio ASM se deberá realizar con Java, utilizando JFlex y CUP para analizar la entrada del código de tres direcciones.

11.2. Entregables

Los entregables para este proyecto son:

- Aplicación funcional, sin dependencias rotas
- Todos los archivos de código fuente, como para correr el proyecto sin necesidad de una conexión a internet (librerías, archivos js, css, html, imágenes, tipografías, etc)
- Gramática desarrollada

11.3. Requerimientos mínimos

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

Lenguaje Basic3D:

- Operadores aritméticos
- Operadores relacionales
- Operadores lógicos
- Estructuras
- Variables
- Sentencias de control
    - If
    - Switch
    - Break
    - Continue
    - Return
- Ciclos
    - While
    - Do-While
    - Repeat-Until
    - For
    - Loop
- Métodos y funciones
- Método principal
- Funciones primitivas
- Generación de código intermedio tres direcciones en tiempo real
- Recuperación de errores

Generación de código intermedio (código de tres direcciones):

- Asignación de temporales
- Generación de temporales
- Uso de estructuras especiales (Stack, Heap y String Pool)
- Saltos
- Llamadas a métodos
- Printf

Funcionamiento Debugger:

- Inicio/Play
- Pausa
- Detener/Stop
- Velocidad de la ejecución
- Resaltar línea actual de ejecución
- Visualización del comportamiento de las estructuras durante la ejecución

Reportes:

- Reporte de tabla de símbolos
- Reporte de optimización
- Reporte de errores
- Gráfica de bloques simple

Fecha de entrega: Viernes 19 de mayo antes de las 6:59:59 pm.