

Universidad de San Carlos de Guatemala  
Facultad de Ingeniería  
Escuela de Ingeniería en Ciencias y Sistemas  
Organización de Lenguajes y Compiladores 2  
Primer semestre 2019



**Catedráticos:** Ing. Bayron López, Ing. Erick Navarro y Ing. Edgar Saban

**Tutores académicos:** Mike Gutiérrez, Javier Navarro, Julio Arango.

# CAAS

*Segundo proyecto de laboratorio*

## Tabla de contenido

1	Objetivos .....	13
1.1	Objetivo general .....	13
1.2	Objetivos específicos .....	13
2	Descripción .....	13
2.1	Antecedentes .....	13
2.2	Compilador como servicio, CAAS. ....	13
2.3	Flujo de la aplicación.....	14
3	Componentes y servicios de la aplicación .....	16
3.1	Editor en línea.....	16
3.1.1	Archivo.....	17
3.1.2	Opciones de compilación.....	17
3.1.3	Ejecución de código 3D .....	18
3.1.4	Reportes .....	18
3.1.5	Área de salida.....	21
3.2	Depurador de código de tres direcciones.....	22
3.2.1	Selección de línea de código para iniciar depuración.....	22
3.2.2	Iniciar depuración .....	22
3.2.3	Pausar depuración .....	23
3.2.4	Parar depuración .....	23

3.3	Interprete de código de tres direcciones. ....	24
4	Notación utilizada para la definición del lenguaje.....	25
4.1	Sintaxis y ejemplos .....	25
4.2	Asignación de colores .....	25
4.3	Expresiones .....	25
4.4	Cerradura positiva.....	25
4.5	Cerradura de Kleene.....	25
4.6	Opcionalidad .....	25
4.7	Salida por consola.....	26
5	Introducción a lenguaje Coline.....	27
5.1	Tipos .....	27
5.1.1	Tipos primitivos.....	27
5.1.2	Tipos de referencia.....	28
5.1.3	Tipo nulo.....	28
5.2	Paradigma de programación.....	28
5.3	Motor de Coline.....	29
5.4	Archivo con extensión .coline.....	29
5.5	Convención de nombres para métodos y funciones .....	29
6	Definición léxica .....	30
6.1	Finalización de línea .....	30
6.2	Espacios en blanco .....	30
6.3	Comentarios.....	30
6.4	Sensibilidad a mayúsculas y minúsculas .....	30
6.5	Identificadores.....	30
6.6	Palabras reservadas .....	31
6.7	Símbolos .....	31
6.8	Literales .....	32
7	Definición de sintaxis .....	33
7.1	Nombres .....	33
7.1.1	Variables.....	33
7.1.2	Tipos de variables .....	33

7.1.3	Variables finales .....	35
7.1.4	Valores predeterminados para las variables.....	35
7.2	Clases .....	36
7.2.1	Introducción a las clases en Coline .....	36
7.2.2	Declaración de una clase .....	37
7.2.3	Miembros de una clase.....	40
7.2.4	Declaración de un campo .....	42
7.2.5	Declaración de un método.....	45
7.2.6	Constructor .....	50
7.2.7	Preparación de una clase .....	51
7.3	Arreglos.....	51
7.3.1	Tipos de arreglos.....	52
7.3.2	Declaración de arreglos.....	52
7.3.3	Asignación de un valor a posiciones de un arreglo .....	53
7.3.4	Inicialización de arreglos .....	54
7.3.5	Array Store Exception.....	54
7.3.6	Miembro length.....	55
7.3.7	Un arreglo de tipo char no es un String .....	55
7.4	Collections de Coline .....	55
7.4.1	LinkedList .....	55
7.5	Excepciones.....	58
7.5.1	Los tipos de las excepciones.....	59
7.5.2	La jerarquía .....	59
7.5.3	Causas de las excepciones .....	60
7.5.4	Manejo en tiempo de ejecución de una excepción .....	60
7.6	Sentencias .....	62
7.6.1	Bloques.....	62
7.6.2	Sentencia import.....	62
7.6.3	Declaración de variables .....	62
7.6.4	Asignación de variables.....	64
7.6.5	Sentencias de transferencia .....	65
7.6.6	Sentencias de selección .....	67

7.6.7	Sentencia throw .....	69
7.6.8	Sentencia try catch .....	70
7.6.9	Sentencias cíclicas o bucles .....	70
7.6.10	Sentencias de entrada/salida .....	73
7.7	Expresiones .....	77
7.7.1	this .....	77
7.7.2	Signos de agrupación .....	77
7.7.3	Expresiones de creación de instancias de clase .....	78
7.7.4	Expresiones de creación de arreglos .....	78
7.7.5	Expresiones de acceso a un campo .....	78
7.7.6	Expresión de invocación a método .....	79
7.7.7	Casteos .....	79
7.7.8	Operadores aritméticos .....	83
7.7.9	Operadores prefijos .....	88
7.7.10	Operadores postfijos .....	89
7.7.11	Operadores relacionales .....	90
7.7.12	Operaciones lógicas .....	92
7.7.13	Operador ternario .....	93
8	El formato de código intermedio .....	95
8.1	Definición léxica .....	95
8.1.1	Finalización de línea .....	95
8.1.2	Espacios en blanco .....	95
8.1.3	Comentarios .....	95
8.1.4	Temporales .....	95
8.1.5	Etiquetas .....	96
8.1.6	Identificadores .....	96
8.1.7	Palabras reservadas .....	97
8.1.8	Símbolos .....	97
8.1.9	Literales .....	97
8.2	Definición de sintaxis .....	98
8.2.1	Declaración de variables .....	98
8.2.2	Operadores aritméticos .....	98

8.2.3	Operadores relacionales.....	98
8.2.4	Operadores lógicos .....	99
8.2.5	Asignación .....	99
8.2.6	Destino de un salto .....	99
8.2.7	Salto incondicional.....	99
8.2.8	Salto condicional .....	100
8.2.9	Declaración de métodos .....	100
8.2.10	Invocación a métodos .....	101
8.2.11	Sentencia print .....	101
9	Entorno de ejecución .....	102
9.1	Estructuras del entorno de ejecución .....	102
9.1.1	El Stack y su puntero.....	102
9.1.2	El Heap y su puntero .....	104
9.1.3	Acceso a estructuras del entorno de ejecución .....	104
10	Optimización de código intermedio .....	105
10.1	Eliminación de instrucciones redundantes de carga y almacenamiento. ..	105
10.1.1	Regla 1 .....	105
10.2	Eliminación de código inalcanzable .....	105
10.2.1	Regla 2.....	106
10.2.2	Regla 3.....	106
10.2.3	Regla 4.....	106
10.2.4	Regla 5.....	107
10.3	Optimizaciones de Flujo de control .....	107
10.3.1	Regla 6.....	107
10.3.2	Regla 7.....	107
10.4	Simplificación algebraica y reducción por fuerza .....	108
10.4.1	Regla 8.....	108
10.4.2	Regla 9.....	108
10.4.3	Regla 10.....	108
10.4.4	Regla 11 .....	108
10.4.5	Regla 12.....	109
10.4.6	Regla 13.....	109

10.4.7	Regla 14.....	109
10.4.8	Regla 15.....	109
10.4.9	Regla 16.....	109
10.4.10	Regla 17.....	110
10.4.11	Regla 18.....	110
11	Manejo de errores .....	111
12	Apéndice A: Precedencia y asociatividad de operadores.....	113
13	Apéndice B: Archivo de prueba .....	114
14	Apéndice C: Referencias.....	114
14.1	Jison.....	114
14.2	AWS.....	114
14.3	SDK.....	114
14.4	Tutorial manejo base de datos DynamoDB.....	114
15	Entregables y Restricciones .....	114
15.1	Entregables.....	114
15.2	Restricciones.....	115
15.3	Requisitos mínimos.....	115
15.4	Entrega del proyecto .....	117

# Índice de figuras

Figura 1: flujo por defecto del proceso de compilación .....	15
Figura 2: flujo alterno del proceso de compilación .....	15
Figura 3: proceso de ejecución .....	16
Figura 4: prototipo para el editor en línea.....	16
Figura 5: menú de archivo en editor en línea .....	17
Figura 6: menú de opciones de compilación en editor en línea .....	17
Figura 7: menú de ejecución de código 3D en editor en línea .....	18
Figura 8: menú de reportes en editor en línea .....	19
Figura 9: prototipo de reporte de errores de compilación.....	19
Figura 10: prototipo de reporte de tabla de símbolos .....	20
Figura 11: prototipo de reporte de AST .....	20
Figura 12: prototipo de reporte de optimización de código 3D .....	21
Figura 13: prototipo de área de salida del editor en línea .....	21
Figura 14: prototipo de intérprete de código 3D .....	24
Figura 15: proceso interno al declarar una variable .....	63
Figura 16: proceso interno al asignar una variable .....	64
Figura 17: diagrama de flujo para la sentencia de selección if .....	68
Figura 18: casteos permitidos .....	80
Figura 19: diagrama de flujo para un operador ternario .....	94
Figura 20: Representación gráfica del Stack.....	103
Figura 21: Representación grafica del Heap .....	104

# Índice de tablas

Tabla 1: código de colores .....	25
Tabla 2: rangos y requisitos de almacenamiento de cada tipo de dato.....	27
Tabla 3: lista de palabras reservadas.....	31
Tabla 4: lista de símbolos.....	31
Tabla 5: sistema de tipos para la suma.....	84
Tabla 6: sistema de tipos para la resta.....	84
Tabla 7: sistema de tipos para la multiplicación .....	85
Tabla 8: sistema de tipos para la multiplicación .....	86
Tabla 9: sistema de tipos para el modulo.....	86
Tabla 10: sistema de tipos para la división.....	87
Tabla 11: operadores relacionales .....	91
Tabla 12: tabla de verdad para operadores booleanos .....	92
Tabla 13: lista de palabras reservadas para el código 3D.....	97
Tabla 14: lista de símbolos del código 3D.....	97
Tabla 15: operaciones aritméticas permitidas en el código 3D .....	98
Tabla 16: operaciones relacionales permitidas en el código 3D .....	98
Tabla 17: parámetros permitidos para sentencia print C3D .....	101
Tabla 18: regla 1 de optimización .....	105
Tabla 19: regla 2 de optimización .....	106
Tabla 20: regla 3 de optimización .....	106
Tabla 21: regla 4 de optimización .....	106
Tabla 22: regla 5 de optimización .....	107
Tabla 23: regla 6 de optimización .....	107
Tabla 24: regla 7 de optimización .....	107
Tabla 25: regla 8 de optimización .....	108
Tabla 26: regla 9 de optimización .....	108
Tabla 27: regla 10 de optimización .....	108
Tabla 28: regla 11 de optimización .....	108
Tabla 29: regla 12 de optimización .....	109
Tabla 30: regla 13 de optimización .....	109
Tabla 31: regla 14 de optimización .....	109
Tabla 32: regla 15 de optimización .....	109
Tabla 33: regla 16 de optimización .....	109
Tabla 34: regla 17 de optimización .....	110
Tabla 35: regla 18 de optimización .....	110
Tabla 36: precedencia y asociatividad de operadores lógicos .....	113



# Índice de definiciones de sintaxis

Sintaxis 1: declaración de una clase .....	37
Sintaxis 2: modificadores de clase .....	37
Sintaxis 3: modificadores de clase .....	40
Sintaxis 4: declaración de campo(s) .....	42
Sintaxis 5: modificadores de campo.....	42
Sintaxis 6: declaración de un método.....	45
Sintaxis 7: parámetros formales.....	46
Sintaxis 8: modificadores de un método.....	46
Sintaxis 9: tipo de retorno de un método.....	48
Sintaxis 10: declaración de un constructor.....	50
Sintaxis 11: modificadores de un constructor.....	50
Sintaxis 12: invocaciones explícitas de constructores .....	51
Sintaxis 13: declaración de arreglos .....	53
Sintaxis 14: inicialización de arreglos.....	54
Sintaxis 15: declaración e instanciación de una LinkedList.....	55
Sintaxis 16: bloque de sentencias.....	62
Sintaxis 17: sentencia import .....	62
Sintaxis 18: declaración de una variable local.....	62
Sintaxis 19: asignación de variables .....	64
Sintaxis 20: sentencia break .....	65
Sintaxis 21: sentencia continue.....	65
Sintaxis 22: sentencia return .....	66
Sintaxis 23: sentencia if.....	67
Sintaxis 24: sentencia switch .....	69
Sintaxis 25: sentencia throw.....	70
Sintaxis 26: sentencia try .....	70
Sintaxis 25: sentencia while .....	71
Sintaxis 28: sentencia do .....	71
Sintaxis 29: sentencia for .....	72
Sintaxis 30: sentencia for each .....	73
Sintaxis 31: sentencia println.....	73
Sintaxis 32: sentencia read_file.....	74
Sintaxis 33: sentencia write_file .....	74
Sintaxis 34: sentencia read .....	75
Sintaxis 35: sentencia graph .....	76
Sintaxis 36: signos de agrupación.....	77
Sintaxis 37: expresión de instancia de clase.....	78
Sintaxis 38: expresión de creación de arreglos.....	78
Sintaxis 39: expresión de acceso a un campo .....	79
Sintaxis 40: expresión de invocación de método .....	79

Sintaxis 41: casteo explícito. ....	80
Sintaxis 42: casteo explícito con cadenas .....	81
Sintaxis 43: operación aritmética de potencia .....	86
Sintaxis 44: operador prefijo menos unario .....	88
Sintaxis 45: operador prefijo de incremento .....	88
Sintaxis 46: operador prefijo de decremento .....	89
Sintaxis 47: operador postfijo de incremento .....	89
Sintaxis 48: operador postfijo de decremento .....	90
Sintaxis 49: operador instanceof .....	92
Sintaxis 50: operador ternario .....	93
Sintaxis 51: declaración de variables C3D .....	98
Sintaxis 52: asignación C3D .....	99
Sintaxis 53: destino de un salto .....	99
Sintaxis 54: salto incondicional .....	99
Sintaxis 55: salto condicional if .....	100
Sintaxis 56: salto condicional ifFalse .....	100
Sintaxis 57: declaración de métodos .....	100
Sintaxis 58: invocación de métodos .....	101
Sintaxis 59: sentencia print .....	101
Sintaxis 60: sentencia clean scope .....	103
Sintaxis 61: acceso a estructuras de ejecución (set) .....	104
Sintaxis 62: acceso a estructuras de ejecución (get) .....	105

# Índice de ejemplos

Ejemplo 1: ejemplo de comentarios .....	30
Ejemplo 2: identificadores validos .....	31
Ejemplo 3: identificadores no válidos .....	31
Ejemplo 4: implementación de una clase abstracta .....	38
Ejemplo 5: relación entre superclases y subclases .....	39
Ejemplo 6: campos estáticos.....	43
Ejemplo 7: inicialización de campos.....	44
Ejemplo 8: inicialización de variables de instancia .....	44
Ejemplo 9: declaración de un constructor .....	50
Ejemplo 10: declaración de arreglos .....	53
Ejemplo 11: asignación de un valor a posiciones de un arreglo.....	53
Ejemplo 12: inicialización de arreglo .....	54
Ejemplo 13: excepción Array Store Exception .....	54
Ejemplo 14: obtención de la longitud de un arreglo .....	55
Ejemplo 15: declaración e instanciación de una LinkedList .....	56
Ejemplo 16: uso de una LinkedList .....	58
Ejemplo 17: sentencia import.....	62
Ejemplo 18: declaración de variables.....	63
Ejemplo 19: asignación de variables .....	64
Ejemplo 20: sentencia break .....	65
Ejemplo 21: sentencia continue .....	66
Ejemplo 22: sentencia return.....	67
Ejemplo 23: sentencia if .....	68
Ejemplo 24: sentencia switch .....	69
Ejemplo 25: sentencia throw .....	70
Ejemplo 26: sentencia try .....	70
Ejemplo 27: sentencia while.....	71
Ejemplo 28: sentencia do .....	72
Ejemplo 29: sentencia for.....	72
Ejemplo 30: sentencia for each .....	73
Ejemplo 31: sentencia println .....	73
Ejemplo 32: sentencia read_file .....	74
Ejemplo 33: sentencia write_file.....	75
Ejemplo 34: sentencia read.....	75
Ejemplo 35: sentencia graph.....	76
Ejemplo 36: this.....	77
Ejemplo 37: signos de agrupación. ....	77
Ejemplo 38: expresión de instancia de clase. ....	78
Ejemplo 39: expresión de creación de arreglos. ....	78

Ejemplo 40: expresión de acceso a un campo.....	79
Ejemplo 41: expresión de creación de arreglos. ....	79
Ejemplo 42: casteos implícitos .....	80
Ejemplo 43: casteo explícito .....	81
Ejemplo 44: casteo explícito con cadenas.....	83
Ejemplo 45: operador prefijo menos unario.....	88
Ejemplo 46: operador prefijo de incremento.....	88
Ejemplo 47: operador postfijo de decremento .....	89
Ejemplo 48: operador postfijo de incremento .....	89
Ejemplo 49: operador postfijo de decremento .....	90
Ejemplo 50: declaración de arreglos .....	92
Ejemplo 51: operador ternario.....	94
Ejemplo 52: ejemplo de comentarios .....	95
Ejemplo 53: expresión regular para los temporales .....	96
Ejemplo 54: temporales válidos .....	96
Ejemplo 55: expresión regular para los etiquetas .....	96
Ejemplo 56: etiquetas válidos.....	96
Ejemplo 57: identificadores validos .....	96
Ejemplo 58: identificadores no válidos .....	96
Ejemplo 59: declaración de variables C3D.....	98
Ejemplo 60: asignación C3D .....	99
Ejemplo 61: destino de un salto .....	99
Ejemplo 62: salto incondicional .....	99
Ejemplo 63: salto condicional if .....	100
Ejemplo 64: salto condicional ifFalse .....	100
Ejemplo 65: declaración de métodos .....	101
Ejemplo 66: invocación de métodos.....	101
Ejemplo 67: sentencia print .....	101
Ejemplo 68: sentencia clean scope .....	103

# 1 Objetivos

## 1.1 Objetivo general

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

## 1.2 Objetivos específicos

- Utilizar herramientas para la generación de analizadores léxicos y sintácticos.
- Realizar análisis semántico a los diferentes lenguajes a implementar.
- Que el estudiante aplique los conocimientos adquiridos en la carrera para el desarrollo de un intérprete de máquina de pila.
- Que el estudiante sea capaz de implementar los conocimientos adquiridos en el curso con las nuevas tecnologías.

# 2 Descripción

## 2.1 Antecedentes

El almacenamiento en la nube es un modelo de almacenamiento de datos basado en redes de computadoras donde los datos están alojados en espacios de almacenamiento virtualizados que últimamente está tomando mucho auge en esta era tecnológica ya que permite a todos los usuarios tener su información en cualquier parte o a los desarrolladores poder contar con diversas tecnologías pagando únicamente por sus servicios.

AWS es uno de los principales proveedores de servicios de Cloud Computing, dando una gran diversidad de servicios, entre ellos, bases de datos, herramientas de desarrollo, robótica, IOT, entre muchas más. Algunos de estos servicios se encuentran como free Tier, capas gratuitas, entre los más conocidos de estos esta DynamoDB, que es una base de datos no relacional que provee a los usuarios un uso gratuito de hasta 25 GB.

Teniendo esto en cuenta se desea crear un compilador que genere reportes los cuales no se quieren perder conforme el tiempo, para ello se propone hacer uso de DynamoDB como base de datos, en la cual se almacenarán los distintos reportes generados.

## 2.2 Compilador como servicio, CAAS.

Compilador como servicio o CAAS por sus siglas en inglés (Compiler as a service), es un software de compilación en línea, cuya implantación será hecha de forma local por los estudiantes. CAAS llevará un control de todos los procesos del compilador en una base de datos no relacional haciendo uso de los componentes de la nube, para este caso Amazon web services.

CAAS será un software donde el compilador estará alojado en un entorno local desarrollado por los estudiantes, este compilador será el encargado de pasar de código fuente a código intermedio y ejecutar este código intermedio.

El lenguaje fuente recibido por el compilador será **Coline** (§5) que es un lenguaje de programación orientado a objetos, capaz de manejar todas las características de este paradigma, encapsulamiento, abstracción, herencia y polimorfismo, basado en uno de los lenguajes más importantes en la actualidad, Java.

El código intermedio generado por el compilador será **código de tres direcciones**, se le llama de esa forma porque solo permite referenciar a 3 direcciones de memoria al mismo tiempo, para lograr esto se descompone cada instrucción a una más sencilla de ejecutar que cumpla con el paradigma de tres direcciones.

Además, la aplicación CAAS tendrá la capacidad de generar una serie de reportes que serán usados para diagnosticar errores o ver detalles del proceso de compilación.

- **Reporte de errores:** dará un reporte de todos los errores generados durante la compilación o ejecución y los guardará en DynamoDB.
- **Reporte de tabla de símbolos:** dará un reporte de la tabla de símbolos utilizada durante la compilación y los guardará en DynamoDB.
- **Generación de AST:** dará un reporte con el AST del lenguaje fuente y lo guardará en DynamoDB.
- **Reporte de optimización:** dará un reporte de la optimización del código de tres direcciones generado.

## 2.3 Flujo de la aplicación.

El flujo de la aplicación estará constituido por dos subprocesos, el primero será el proceso de compilación del código de alto nivel **Coline**, y el segundo será la ejecución del código de tres direcciones generado a partir del proceso de compilación.

A partir de todo este proceso de compilación y ejecución se podrá generar diversos reportes, los cuales se desea guardar en la nube haciendo uso de Amazon DynamoDB, esta es una base de datos no relacional proporcionada por Amazon Web Services, lo que se busca es que todos los reportes que desea almacenar el usuario, los pueda acceder desde cualquier parte y en cualquier momento.

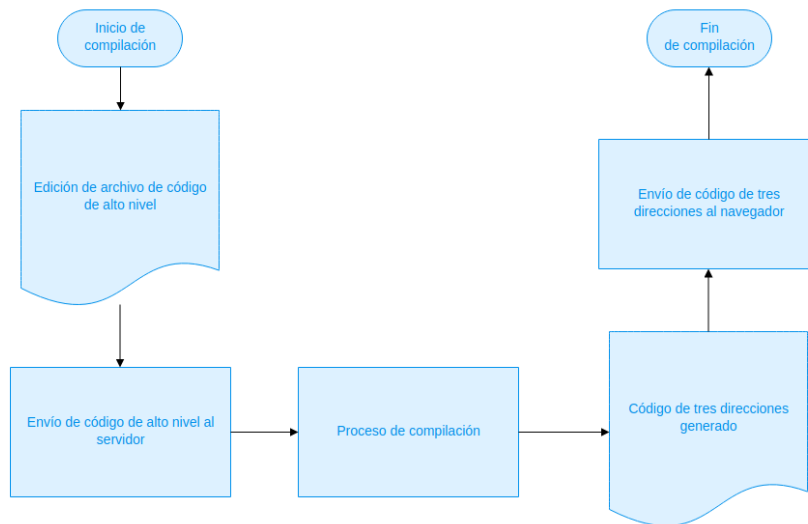
### 2.3.1.1 Proceso de compilación

El proceso de compilación tendrá como entrada código de alto nivel y como salida la representación de éste en código de tres direcciones. El proceso de compilación seguirá los siguientes pasos.

#### Flujo por defecto:

1. Edición de código de alto nivel **Coline** en el editor descrito en la sección (§3.1).
2. Envío del código de alto nivel al servidor.
3. Procesamiento del código de alto nivel en el servidor, este procesamiento llevará a cabo los procesos de análisis léxico, sintáctico y semántico del código de alto nivel para creación del código de tres direcciones.
4. Envío del código de tres direcciones al navegador.

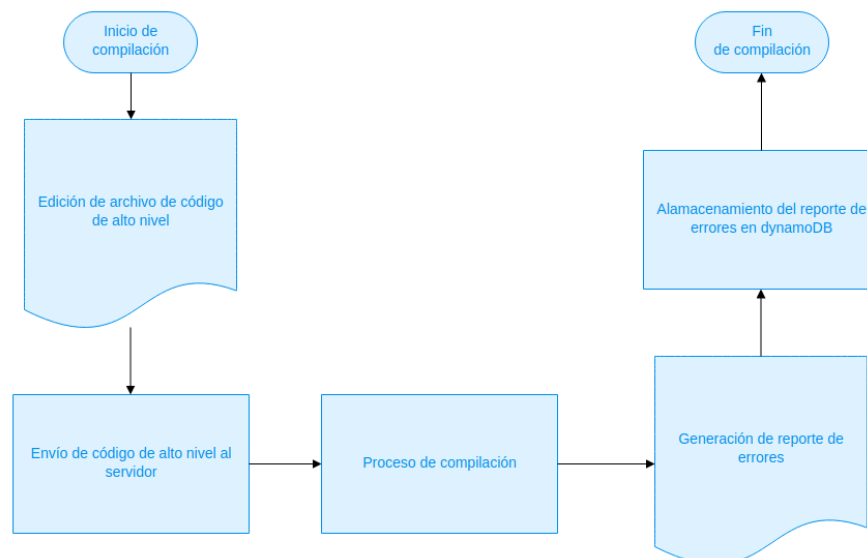
Figura 1: flujo por defecto del proceso de compilación  
Fuente: Elaboración propia.



## Flujo alternativo

1. Edición de código de alto nivel **Coline** en el editor descrito en la sección (§3.1).
2. Envío del código de alto nivel al servidor.
3. Procesamiento del código de alto nivel en el servidor, este procesamiento llevará a cabo los procesos de análisis léxico, sintáctico y semántico del código de alto nivel para creación del código de tres direcciones.
4. Detección de errores en el paso 3 (semánticos, léxicos o sintácticos) y generación del reporte de errores.
5. Almacenamiento del reporte de errores en DynamoDB.

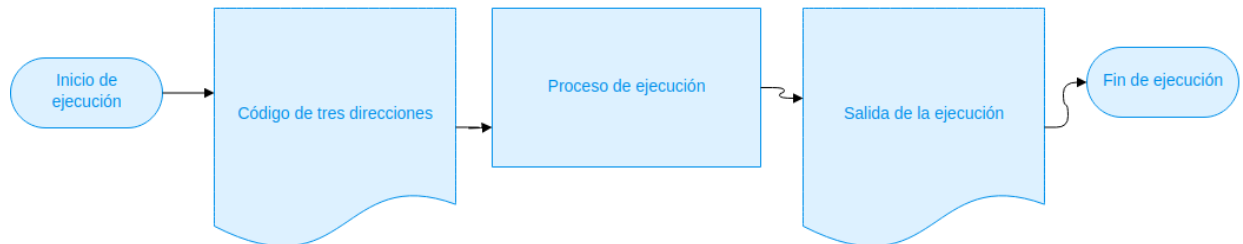
Figura 2: flujo alternativo del proceso de compilación  
Fuente: Elaboración propia.



### 2.3.1.2 Proceso de ejecución

El proceso de ejecución tendrá como entrada código de tres direcciones y como salida los resultados de la ejecución del código de tres direcciones impresos en la consola del editor en línea.

Figura 3: proceso de ejecución  
Fuente: Elaboración propia.



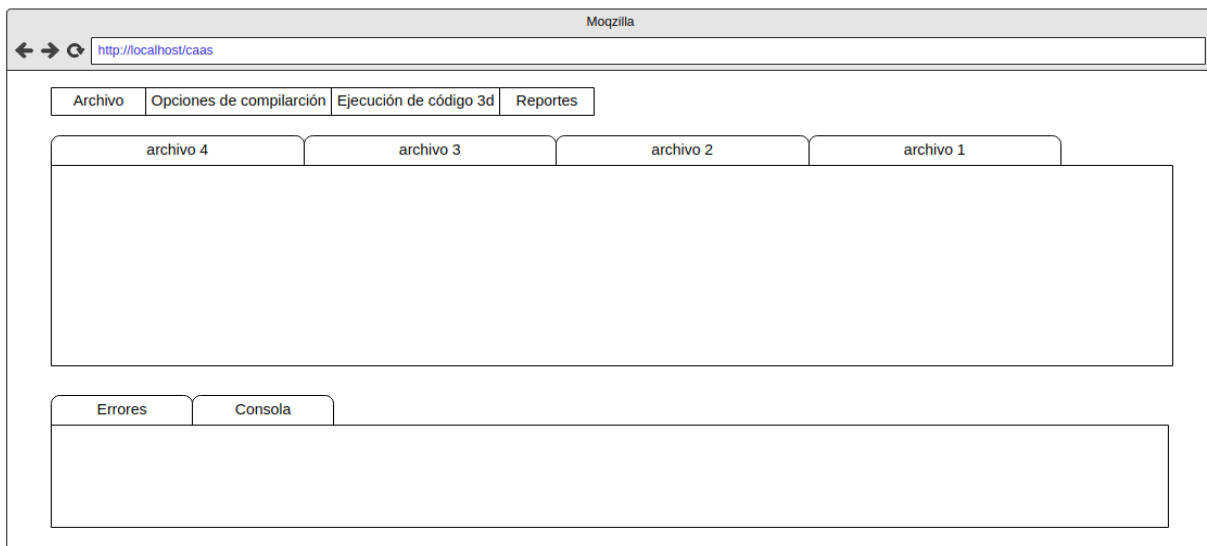
## 3 Componentes y servicios de la aplicación

A continuación, se describen los componentes de CAAS.

### 3.1 Editor en línea

CAAS tendrá un editor en línea desarrollado en con html y css, este editor permitirá crear archivos de código de alto nivel. Además, este editor podrá enviar el contenido de los archivos de código de alto nivel al servidor para que estos sean compilados. El resultado de la compilación será mostrado en una sección de resultados de compilación.

Figura 4: prototipo para el editor en línea  
Fuente: Elaboración propia.



El editor en línea estará conformado por las siguientes secciones:

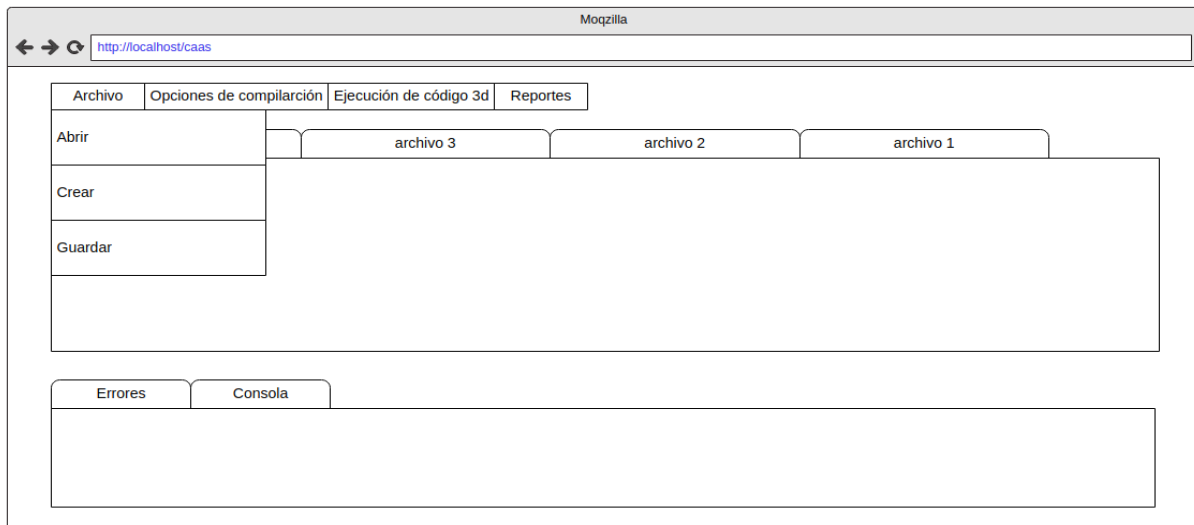


### 3.1.1 Archivo

Tendrá la capacidad de realizar las siguientes acciones:

- **Abrir:** abrirá un nuevo archivo.
- **Crear:** creará un nuevo archivo y lo colocará en una nueva pestaña.
- **Guardar:** guardará el archivo en el sistema en donde se esté ejecutando la aplicación.

*Figura 5: menú de archivo en editor en línea  
Fuente: Elaboración propia.*

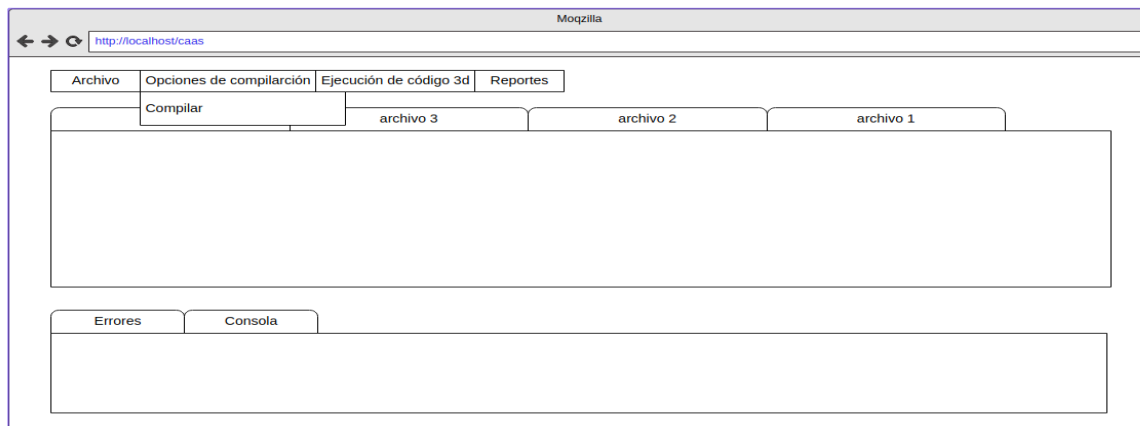


### 3.1.2 Opciones de compilación

Tendrá la capacidad de realizar las siguientes acciones:

- **Compilar:** realizará el análisis y todos los procesos de un compilador hasta obtener el código de tres direcciones.

*Figura 6: menú de opciones de compilación en editor en línea  
Fuente: Elaboración propia.*

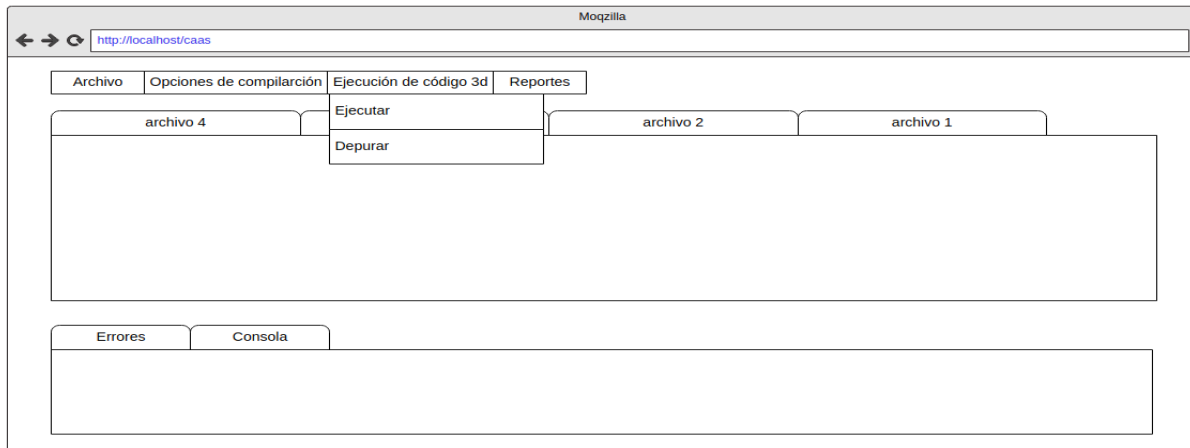


### 3.1.3 Ejecución de código 3D

Tendrá la capacidad de realizar las siguientes acciones:

- **Ejecutar:** realizará la ejecución del código 3D.
- **Depurar:** realizará la ejecución tomando en cuenta el debugger paso a paso, descrito más adelante.

Figura 7: menú de ejecución de código 3D en editor en línea  
Fuente: Elaboración propia.



### 3.1.4 Reportes

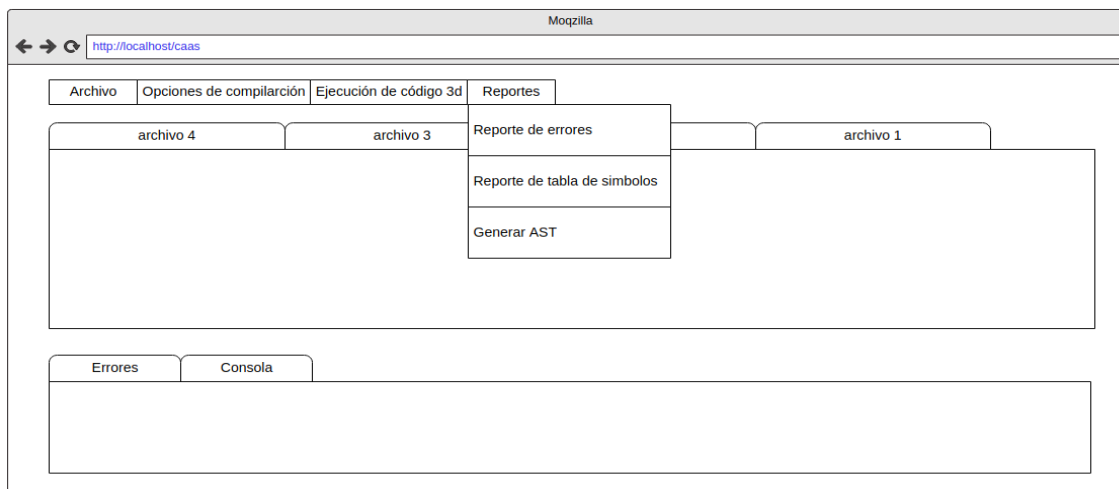
Una de las funcionalidades más innovadoras de CAAS es la generación de reportes y su respectivo almacenamiento en la nube, para lograr esto se usarán las tecnologías de Amazon Web Services como proveedor de la nube y su servicio gratuito de DynamoDB.

**DynamoDB**, es una base de datos de claves-valor, no relacional, que nos provee AWS con la particularidad que no hay servidores que aprovisionar, parchear o administrar, y no hay software que instalar, mantener o utilizar, dándonos así una gran herramienta fácil y rápida de implementar. El compilador y la ejecución será realizada por medio de un servidor local utilizando Node JS, por lo tanto, la generación de reportes sobre este proceso también será realizado por este medio, dando así la necesidad de hacer una conexión entre Node Js y DynamoDB.



AWS SDK para JavaScript es la conexión que nos provee Amazon Web Services y nos da un acceso directo para el almacenamiento y administración de la base de datos de DynamoDB.

Figura 8: menú de reportes en editor en línea  
Fuente: Elaboración propia.



A continuación, se listan los distintos reportes que será posible generar.

### 3.1.4.1 Reporte de errores de compilación

Este reporte mostrará si hubo errores durante el proceso de compilación, mostrando de cada uno, el tipo de error (léxico, sintáctico o semántico), la fila y columna de donde se detectó cada error.

Figura 9: prototipo de reporte de errores de compilación  
Fuente: Elaboración propia.

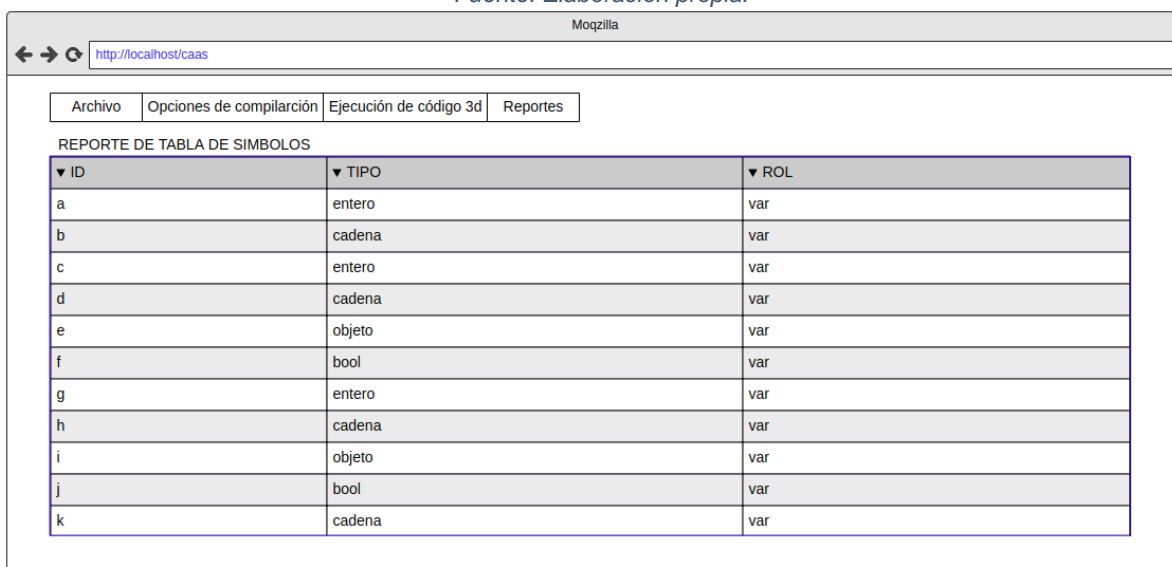
The screenshot shows the same web browser window as Figure 8, but now the 'Reporte de errores de compilación' is displayed. The table has three columns: 'Tipo', 'Fila', and 'Columna'. The data is as follows:

Tipo	Fila	Columna
semántico	2	3
léxico	5	6
sintáctico	8	9
semántico	2	3
léxico	5	6
sintáctico	8	9
semántico	2	3
léxico	5	6
sintáctico	8	9

### 3.1.4.2 Reporte de tabla de símbolos

Este reporte mostrará la tabla de símbolos del código fuente cuando este paso por el proceso de compilación, mostrará todas las variables globales de la clase principal, todos los métodos de la clase principal y todas las clases del código fuente, mostrando como mínimo el identificador, el tipo y el rol de cada uno.

Figura 10: prototipo de reporte de tabla de símbolos  
Fuente: Elaboración propia.

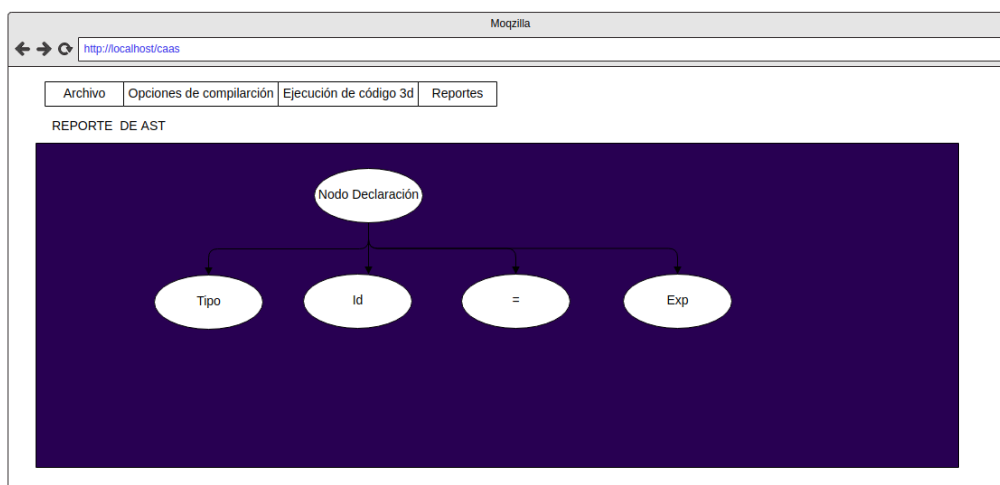


▼ ID	▼ TIPO	▼ ROL
a	entero	var
b	cadena	var
c	entero	var
d	cadena	var
e	objeto	var
f	bool	var
g	entero	var
h	cadena	var
i	objeto	var
j	bool	var
k	cadena	var

### 3.1.4.3 Reporte de AST

Este reporte mostrará el árbol de análisis sintáctico que se forma durante el análisis sintáctico en el proceso de compilación.

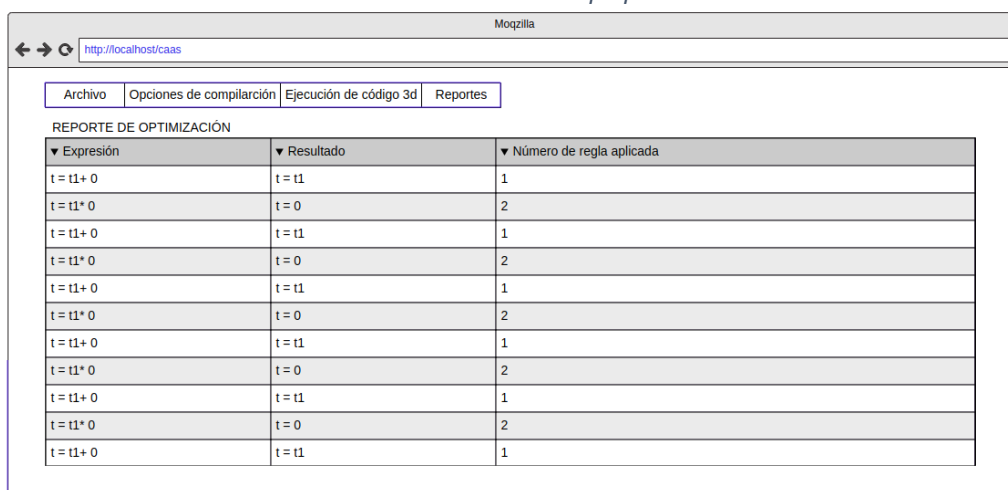
Figura 11: prototipo de reporte de AST  
Fuente: Elaboración propia.



### 3.1.4.4 Reporte de optimización

Este reporte mostrará todas las optimizaciones que fueron posible realizar en el código de tres direcciones generado por el proceso de compilación, este reporte mostrara como mínimo **la expresión original, la expresión optimizada, el número de línea de código y el número de la regla de optimización que se realizó**, las reglas de optimización son descritas más adelante.

Figura 12: prototipo de reporte de optimización de código 3D  
Fuente: Elaboración propia.



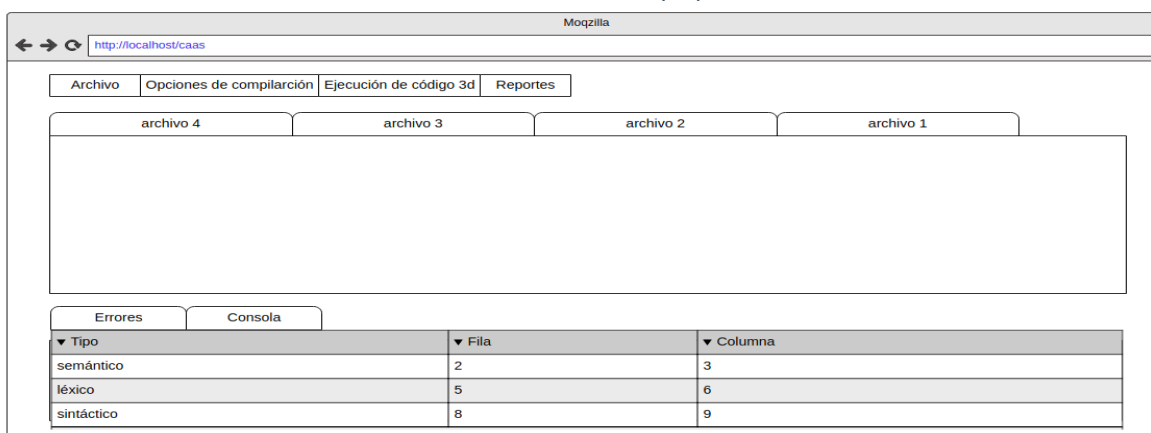
▼ Expresión	▼ Resultado	▼ Número de regla aplicada
$t = t1 + 0$	$t = t1$	1
$t = t1 * 0$	$t = 0$	2
$t = t1 + 0$	$t = t1$	1
$t = t1 * 0$	$t = 0$	2
$t = t1 + 0$	$t = t1$	1
$t = t1 * 0$	$t = 0$	2
$t = t1 + 0$	$t = t1$	1
$t = t1 * 0$	$t = 0$	2
$t = t1 + 0$	$t = t1$	1
$t = t1 * 0$	$t = 0$	2
$t = t1 + 0$	$t = t1$	1

### 3.1.5 Área de salida

Además, el editor en línea tendrá los siguientes componentes:

- **Consola:** en donde se mostrarán los resultados del proceso de ejecución del código de tres direcciones.
- **Área de reporte de errores:** en donde se mostrarán los errores detectados durante el proceso de compilación.

Figura 13: prototipo de área de salida del editor en línea  
Fuente: Elaboración propia.



▼ Tipo	▼ Fila	▼ Columna
semántico	2	3
léxico	5	6
sintáctico	8	9

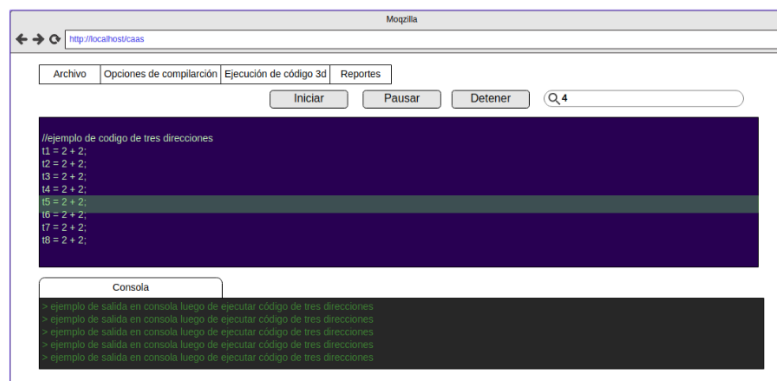
## 3.2 Depurador de código de tres direcciones

Esta será una herramienta que se usará observar el proceso de ejecución del código de tres direcciones en tiempo real. **En el debugger deberán mostrarse a discreción del estudiante, las estructuras de ejecución (Stack y Heap).**

### 3.2.1 Selección de línea de código para iniciar depuración

Para seleccionar la línea de código en donde se dese iniciar el proceso de depuración deberá introducirse el número de línea en la caja de texto que sirve para seleccionar el punto de partida.

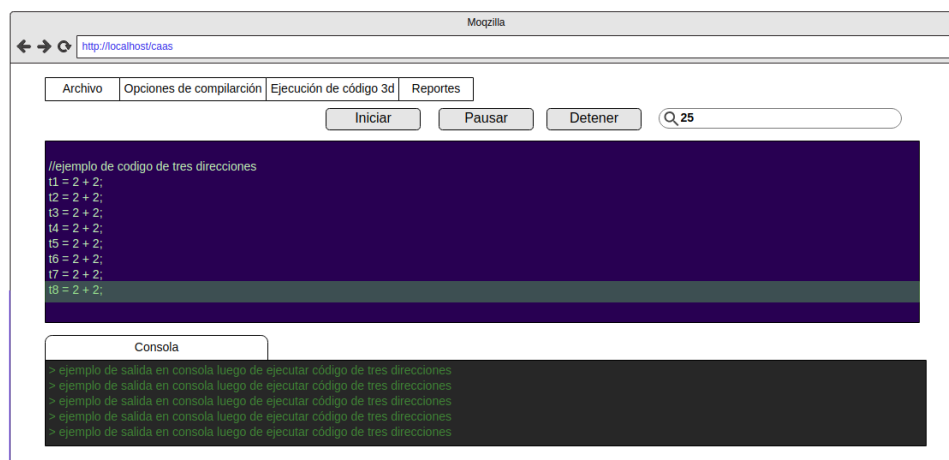
Figura 14: Selección de línea de depuración  
Fuente: Elaboración propia.



### 3.2.2 Iniciar depuración

Luego de haber seleccionado el número de línea en la que se desea iniciar la depuración se podrá iniciar el proceso de depuración al hacer click sobre el botón Iniciar. Al iniciarse la depuración la herramienta deberá de pintar la línea de código que se esté ejecutando, ver imagen 3.2.2.

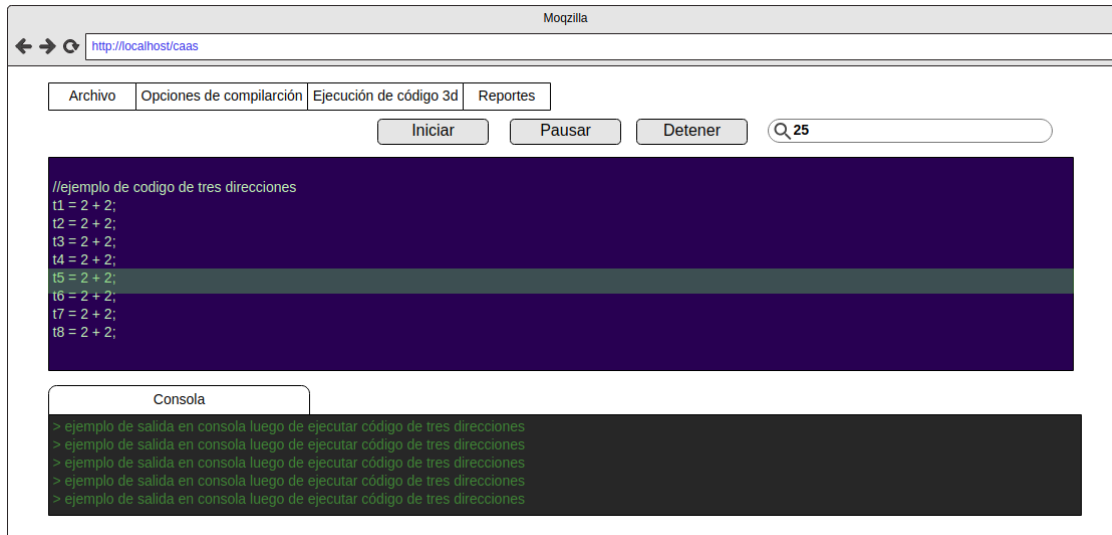
Figura 15: Inicio depuración  
Fuente: Elaboración propia.



### 3.2.3 Pausar depuración

Cuando se esté depurando y se presione el botón pausar el proceso de depuración se detendrá, pero mantendrá el estado en el que se encontraba antes de dar click sobre el botón de pausar.

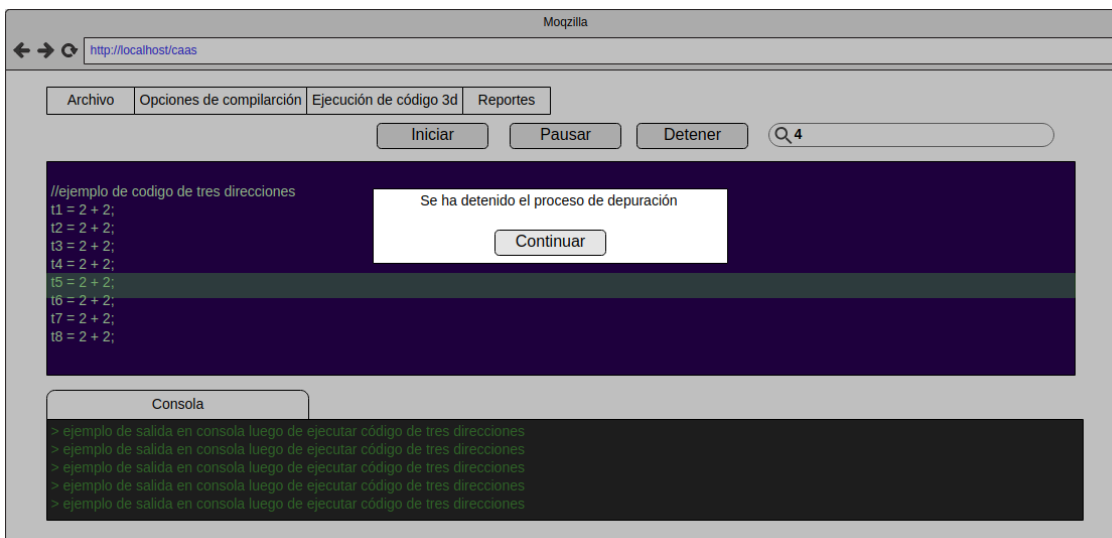
Figura 16: Pausar depuración  
Fuente: Elaboración propia.



### 3.2.4 Parar depuración

Al dar click sobre el botón detener el proceso de depuración se detendrá completamente el proceso de depuración.

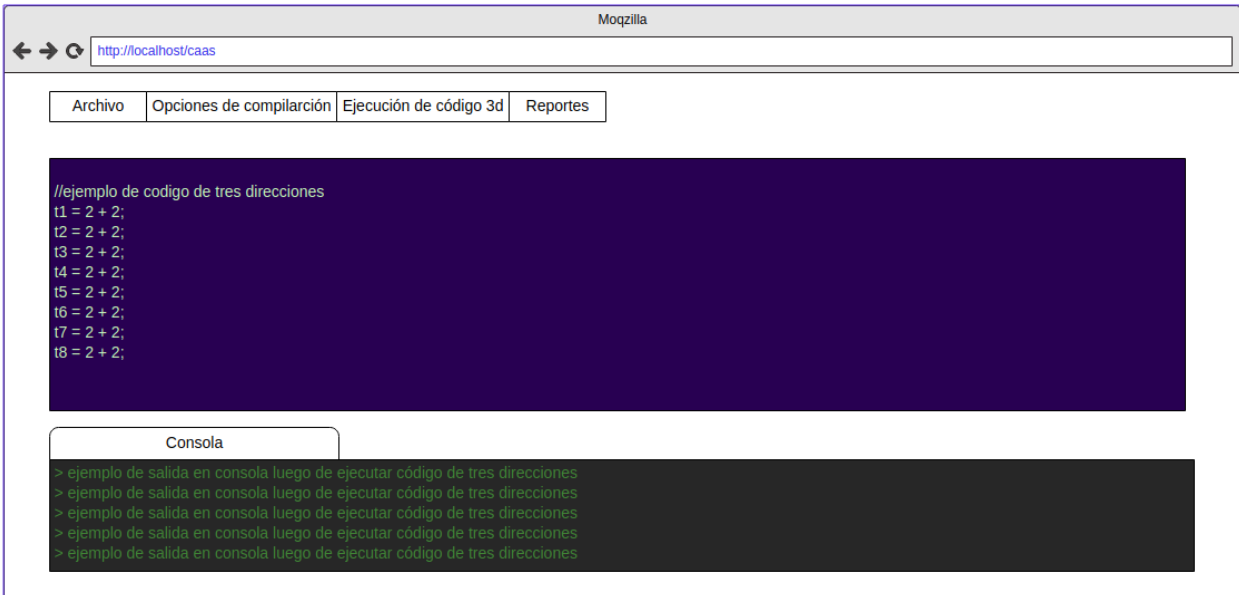
Figura 18: Detener depuración  
Fuente: Elaboración propia.



### 3.3 Interprete de código de tres direcciones.

El intérprete de código de tres direcciones será una herramienta que permitirá la ejecución del código de tres direcciones que será generado en el proceso de compilación.

Figura 14: prototipo de intérprete de código 3D  
Fuente: Elaboración propia.





## 4 Notación utilizada para la definición del lenguaje

Dentro del presente documento, se utilizarán las siguientes notaciones cuando se haga referencia al código de **Coline** o **código de tres direcciones**.

### 4.1 Sintaxis y ejemplos

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



### 4.2 Asignación de colores

Dentro del enunciado y en el editor de texto de la aplicación, para el código de alto nivel, seguirá el formato de colores establecido en la Tabla 1.

*Tabla 1: código de colores  
Fuente: Elaboración propia.*

COLOR	TOKEN
<b>AZUL</b>	Palabras reservadas
<b>NARANJA</b>	Cadenas, caracteres
<b>MORADO</b>	Números
<b>GRIS</b>	Comentario
<b>CAFÉ</b>	Salida por consola
<b>NEGRO</b>	Otro

### 4.3 Expresiones

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

- Una operación aritmética, relacional o lógica,
- Un acceso a un arreglo u objeto.

### 4.4 Cerradura positiva

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

### 4.5 Cerradura de Kleene

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

### 4.6 Opcionalidad

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

## 4.7 Salida por consola

Las salidas por consola del lenguaje **Coline** se denotarán iniciando con el símbolo mayor que (>).

```
> Esta es una salida por consola
```

## 5 Introducción a lenguaje Coline

Coline es un lenguaje derivado de Java, es decir, está conformado por un subconjunto de sus instrucciones y utiliza el paradigma de programación orientado a objetos.

### 5.1 Tipos

**Coline** es un lenguaje de programación de tipado estático, esto quiere decir que al momento de compilación ya se conoce el tipo de cada variable y cada expresión. También es un lenguaje de programación con tipado fuerte ya que el tipo de una variable define el rango de valores que esta puede tomar.

#### 5.1.1 Tipos primitivos

Se utilizarán los siguientes tipos de dato:

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

Tabla 2: rangos y requisitos de almacenamiento de cada tipo de dato  
Fuente: Elaboración propia.

TIPO DE DATO	RANGO	TAMAÑO
<code>int</code>	<code>[-2147483648, 2.147.483.647]</code>	4 bytes
<code>double</code>	<code>[-9223372036854775800, 9223372036854775800]</code>	8 bytes
<code>char</code>	<code>[0,255]</code> (ASCII)	1 byte
<code>boolean</code>	<code>true</code> , <code>false</code>	1 byte
<code>String</code>	<code>[0, 2.147.483.647]</code> caracteres	variable

#### NOTAS

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

---

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

## 5.1.2 Tipos de referencia

Estos tipos están dados por la creación de los siguientes elementos. Los valores de los tipos por referencia son las referencias a alguno de los siguientes:

- Objetos
- Arreglos

### 5.1.2.1 Clase *Object*

La clase *Object* es una superclase (§7.2.2.2) de todas las demás clases.

Todas las clases y arreglos heredan (§7.2.5.5) los métodos (§7.2.5) de la clase *Object*.

#### 5.1.2.1.1 Miembros

- El método `equals(Object o)` define una noción de igualdad de objeto, que se basa en la referencia, no en la comparación. Definirá la igualdad dada por el objeto al que pertenece y el objeto `o` recibido como parámetro.
- El método `getClass()` devuelve un `String` que representa el nombre de la clase del objeto.
- El método `toString()` devuelve una representación `String` del objeto.

### 5.1.2.2 Clase *String*

Las instancias de la clase *String* representan secuencias de caracteres ASCII.

- Un objeto `String` tiene un valor constante (invariable).
- Los literales de cadena (§6.8) son referencias a instancias de la clase *String*.
- El operador de concatenación de cadenas `+` (§7.7.8.1) crea implícitamente un nuevo objeto `String` cuando el resultado no es una expresión constante de tiempo de compilación.

#### 5.1.2.2.1 Miembros

- El método `toCharArray()` en la clase *String* devuelve un arreglo de caracteres que contiene la misma secuencia de caracteres que un *String*.
- El método `length()` devuelve la longitud de esta cadena.
- El método `toUpperCase()` devuelve la cadena en mayúsculas.
- El método `toLowerCase()` devuelve la cadena en minúsculas.

## 5.1.3 Tipo nulo

El tipo nulo, será utilizado para hacer referencia a la nada.

## 5.2 Paradigma de programación

**Coline** es un lenguaje totalmente orientado a Objetos. Todos los conceptos en los que se apoya esta técnica, encapsulación, herencia, polimorfismo, etc., están presentes en **Coline**.

### 5.3 Motor de Coline

El motor de coline es un sistema de procesamiento de lenguaje compuesto por un compilador híbrido, ya que posee

### 5.4 Archivo con extensión .coline

Debido a que **Coline** es un lenguaje completamente orientado a objetos, un archivo que contenga código **Coline** básicamente estará compuesto por dos elementos (explicados a detalle en el capítulo 6 del presente documento):

- Una lista de sentencias de importación.
- Una lista de clases.

### 5.5 Convención de nombres para métodos y funciones

En el presente documento se tomará como equivalente mencionar método y función, cuando sea necesario definir algo para cada objeto, se aclarará explícitamente, de lo contrario tomaremos método como el conjunto método y función.

## 6 Definición léxica

### 6.1 Finalización de línea

El compilador de **Coline** dividirá la entrada en líneas, estas líneas estarán divididas mediante componentes léxicos que determinen la finalización de las mismas y son los siguientes:

- Carácter salto de línea (LF ASCII)
- Carácter retorno de carro (CR ASCII)
- Carácter retorno de carro (CR ASCII) seguido de carácter salto de línea (LF ASCII)

### 6.2 Espacios en blanco

Los espacios en blancos definirán la división entre los componentes léxicos. Serán considerados espacios en blanco los siguientes caracteres:

- Espacio (SP ASCII)
- Tabulación horizontal (HT ASCII)
- Caracteres de finalización de línea

### 6.3 Comentarios

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

Existirán dos tipos de comentarios:

- Los comentarios de una línea que serán delimitados al inicio con los símbolos “//” y al final con un carácter de finalización de línea
- Los comentarios con múltiples líneas que empezarán con los símbolos “/\*” y terminarán con los símbolos “\*/”.

```
//este es un ejemplo de un lenguaje de una sola línea
/*
este es un ejemplo de un lenguaje de múltiples líneas.
*/
```

*Ejemplo 1: ejemplo de comentarios  
Fuente: Elaboración propia.*

### 6.4 Sensibilidad a mayúsculas y minúsculas

**Coline** será un lenguaje case sensitive, es decir, dicho lenguaje **si** será sensible a las mayúsculas y minúsculas, esto aplicará tanto para palabras reservadas propias del lenguaje como para identificadores. Por ejemplo, CoLiNe no es igual que Coline.

### 6.5 Identificadores

Un identificador será utilizado para dar un nombre a variables, métodos o estructuras.

Un identificador es una secuencia de caracteres alfabéticos [**A-Z a-z**] incluyendo el guion bajo [**\_**] o dígitos [**0-9**] que comienzan con un carácter alfabético o guion bajo. \

A continuación, se muestran ejemplos de identificadores validos:

```
este_es_un_identificador_valido_09
_este_tambien_09
Y_este_2018
```

*Ejemplo 2: identificadores validos*

*Fuente: Elaboración propia.*

A continuación, se muestran ejemplos de identificadores no validos:

```
0id
id-5
```

*Ejemplo 3: identificadores no válidos*

*Fuente: Elaboración propia.*

## 6.6 Palabras reservadas

Son palabras que no podrán ser utilizadas como identificadores ya que representan algo específico y necesario dentro de **Coline**.

*Tabla 3: lista de palabras reservadas*

*Fuente: Elaboración propia.*

abstract	double	new	super
boolean	else	pow	switch
break	extends	println	this
case	final	private	toChar
catch	for	protected	toDouble
char	graph	public	toInt
class	if	return	try
continue	import	read_file	while
default	instanceof	static	write_file
do	int	str	

## 6.7 Símbolos

*Tabla 4: lista de símbolos*

*Fuente: Elaboración propia.*

+	suma	-	resta negativo	*	multiplicación	/	división
%	modulo	++	incremento	--	decremento	=	igual
!=	diferente que	==	igual que	>=	mayor o igual que	>	mayor que
<=	menor o igual que	<	menor que	?	interrogante	:	dos puntos
&&	and		or	^	xor	!	not
(	paréntesis izquierdo	(	paréntesis derecho	[	corchete derecho	]	corchete derecho

	;		punto y coma		{		llave izquierda		}		llave derecha			
--	---	--	-----------------	--	---	--	--------------------	--	---	--	---------------	--	--	--

## 6.8 Literales

Representan el valor de un tipo primitivo

- Enteros: [0-9]+
- Decimales: [0-9]+(“.” 0-9)+)?
  - Toda expresión de este tipo utilizará 6 decimales con poda o corte.
  - Si se desea imprimir una expresión de tipo decimal y la parte entera cuenta con 8 dígitos o más, deberá imprimirse en notación científica.
- Caracter: “” <Carácter ASCII> “”
- Cadena: “” <Caracteres ASCII> “”
  - Será el único literal que si no se inicializa, su valor será **null**.
- Booleanos: [**true**, **false**]
- Nulo: **null**
  - Los literales de tipo char, int y double no podrán ser de tipo **null**.



## 7 Definición de sintaxis

### 7.1 Nombres

Los nombres se utilizan para referirse a entidades declaradas en un programa.

Una entidad declarada es un tipo de clase, miembro (clase, campo o método) de un tipo de referencia, parámetro (de un método, constructor o controlador de excepciones), o variable local. Los nombres en los programas pueden ser simples, consistentes en un solo identificador, o calificados, consistentes en una secuencia de identificadores separados por punto ("."). Cada declaración que introduce un nombre tiene un alcance, que es la parte del texto del programa dentro del cual se puede hacer referencia a la entidad declarada con un nombre simple.

#### 7.1.1 Variables

Una variable es una ubicación de memoria y tiene un tipo asociado, el mismo puede ser:

- Un tipo primitivo (§5.1.1): Una variable de un tipo primitivo siempre tiene un valor primitivo de ese tipo primitivo exacto, a lo que le llamamos literal (§6.8).
- Un tipo de referencia (§5.1.2): Una variable de un tipo de clase T puede contener una referencia nula o una referencia a una instancia de la clase T o de cualquier clase que es una subclase de T.
  - Si T es un tipo de referencia, a continuación, una variable de tipo "arreglo de T" puede contener una referencia nula o una referencia a cualquier arreglo de tipo "arreglo de S" de tal manera que el tipo S es una subclase.

El valor de una variable se cambia por una de las siguientes sentencias:

- Por una asignación (§7.6.4).
- Por un operador prefijo o postfijo de incremento o decremento (§7.7.9.2, §7.7.9.3, §7.7.10.1, §7.7.10.2).

#### 7.1.2 Tipos de variables

A continuación, se definen los tipos de variables:

##### 7.1.2.1 Variable de clase o estática

Una variable estática es un campo declarado utilizando el modificador `static` dentro de una declaración de clase (§7.2). Una variable estática se crea cuando se prepara su clase (§7.2.2) y se inicializa con un valor predeterminado (§7.1.4).

##### 7.1.2.2 Variable de instancia

Una variable de instancia es un campo declarado dentro de una declaración de clase sin usar la palabra clave `static`. Si una clase T tiene un campo que es una variable de instancia, entonces se crea una nueva variable de instancia y se inicializa a un valor predeterminado (§7.1.4) como parte de cada objeto recién creado de la clase T o de cualquier clase que sea una subclase de T.

### 7.1.2.3 Variable sin nombre

Los elementos de un arreglo son variables sin nombre que se crean y se inicializan a valores predeterminados (§7.1.4).

### 7.1.2.4 Parámetros de un método o función

Los parámetros de los métodos (§7.2.5) nombran valores de argumento pasados a un método.

Para cada parámetro declarado en una declaración de método, se crea una nueva variable de parámetro cada vez que se invoca ese método (§7.2.6). La nueva variable se inicializa con el valor de argumento correspondiente de la invocación del método. El parámetro del método deja de existir efectivamente cuando se completa la ejecución del cuerpo del método.

### 7.1.2.5 Parámetros del constructor

Los parámetros del constructor (§7.2.6) nombran valores de argumento pasados a un constructor.

Para cada parámetro declarado en una declaración de constructor, se crea una nueva variable de parámetro cada vez que una expresión de creación de instancia de clase (§) o una invocación explícita de constructor (§) invoca ese constructor. La nueva variable se inicializa con el valor de argumento correspondiente de la expresión de creación o invocación del constructor. El parámetro del constructor deja de existir efectivamente cuando se completa la ejecución del cuerpo del constructor.

### 7.1.2.6 Parámetro de excepción

Se crea un parámetro de excepción cada vez que se captura una excepción (§7.5) mediante una cláusula `catch` de una sentencia `try`.

La nueva variable se inicializa con el objeto real asociado con la excepción (§7.5). El parámetro de excepción deja de existir efectivamente cuando se completa la ejecución del bloque asociado con la cláusula `catch`.

### 7.1.2.7 Variables locales

Las variables locales se declaran mediante sentencias de declaración de variables locales (§7.6.3).

Cada vez que el flujo de control ingresa a un bloque (§7.6.1) o sentencia `for`, se crea una nueva variable para cada variable local declarada en una declaración de variable local contenida inmediatamente dentro de ese bloque o sentencia `for`.

La variable local deja de existir efectivamente cuando se completa la ejecución del bloque o de la instrucción.

### 7.1.3 Variables finales

Se puede declarar una variable `final`. Una variable `final` solo puede ser asignada una vez. Una vez que se ha asignado una variable `final`, está siempre contendrá el mismo valor.

Consideraciones:

- Si una variable `final` no se inicializa en su declaración, es necesario que se inicialice en el constructor de la clase.
- Si una variable `final` contiene una referencia a un objeto, entonces el estado del objeto puede cambiarse mediante operaciones en el objeto, pero la variable siempre se referirá al mismo objeto.
- Esto se aplica también para los arreglos, porque los arreglos son objetos; Si una variable `final` contiene una referencia a un arreglo, los elementos del arreglo pueden cambiarse mediante operaciones en el mismo, pero la variable siempre se referirá al mismo arreglo.
- Una variable de tipo o tipo primitivo `String`, que se define como `final` y se inicializa con una expresión, se denomina variable constante.

### 7.1.4 Valores predeterminados para las variables

Cada variable en un programa debe tener un valor antes de que se use su valor:

- Cada variable de clase, variable de instancia o componente de arreglo se inicializa con un valor predeterminado cuando se crea:
  - Para el tipo `int`, el valor predeterminado es cero, es decir, `0`.
  - Para el tipo `double`, el valor por defecto es cero positivo, es decir, `0.0`.
  - Para el tipo `char`, el valor por defecto es el carácter nulo, es decir, `'\0'`.
  - Para el tipo `boolean`, el valor predeterminado es `false`.
  - Para todos los tipos de referencia (§5.1.2) y `String`, el valor predeterminado es `null`.
- Cada parámetro del método o función (§7.2.5.1) se inicializa al valor de argumento correspondiente proporcionado por el invocador del método o función (§7.7.6).
- Cada parámetro del constructor se inicializa con el valor de argumento correspondiente proporcionado por una expresión de creación de instancia de clase (§7.7.3).
- Un parámetro de excepción (§7.6.8) se inicializa al objeto lanzado que representa la excepción (§7.5.4).
- A una variable local se le debe dar explícitamente un valor antes de que se use, ya sea por inicialización o asignación, de una manera que pueda verificarse usando las reglas para la asignación definida.

## 7.2 Clases

Las declaraciones de clase definen nuevos tipos de referencia y describen cómo se implementan (§7.2.2).

### 7.2.1 Introducción a las clases en Coline

En **Coline**, una clase puede ser de tres tipos:

- **Clase de nivel superior:** es una clase que no es una clase anidada.
- **Clase miembro:** es una clase cuya declaración se incluye directamente en otra clase.
- **Clase local:** es una clase que se encuentra declarada en un bloque de sentencias.

Una clase nombrada puede ser declarada utilizando el modificador `abstract` (§7.2.2.1.1), es decir, la clase será abstracta y debe declararse abstracta si se implementa de manera incompleta; tal clase no puede ser instanciada, pero puede ser extendida por subclases. Se puede declarar una clase `final` (§7.2.2.1.2), en cuyo caso no puede tener subclases. Si se declara una clase `public`, entonces se puede hacer referencia a ella desde cualquier lugar.

El cuerpo de una clase declara miembros (campos y métodos y clases anidadas), inicializadores estáticos y de instancia, y constructores (§7.2.6). El alcance de un miembro es el cuerpo completo de la declaración de la clase a la que pertenece el miembro.

Un campo, un método de la clase miembro y los constructores pueden incluir los modificadores de acceso `public`, `protected` y `private`. Los miembros de una clase incluyen tanto miembros declarados como heredados (§7.2.3). Los campos recién declarados pueden ocultar campos declarados en una superclase. Los miembros de la clase recientemente declarados pueden ocultar los miembros de la clase o la interfaz declarados en una superclase. Los métodos recientemente declarados pueden ocultar, implementar o sobrescribir los métodos declarados en una superclase.

Las declaraciones de campo describen variables de clase, que se definen una vez y variables de instancia, que se definen para cada instancia de la clase. Un campo puede ser declarado `final` (§7.2.4.1.2), en cuyo caso se puede asignar a una sola vez. Cualquier declaración de campo puede incluir un inicializador

Las declaraciones de clase de miembro (§7.2.3) describen las clases anidadas que son miembros de la clase circundante, estas pueden ser `static`, en cuyo caso no tienen acceso a las variables de instancia de la clase..

Las declaraciones de métodos (§7.2.5) describen el código que puede invocar las expresiones de invocación de métodos (§7.7.6). Un método de clase se invoca en

relación con el tipo de clase; se invoca un método de instancia con respecto a algún objeto particular que es una instancia de una clase, es por eso que cuando se genera código intermedio, es necesario saber la referencia del objeto que invoca dicho método. Un método declarado que no indique como se implementa debe ser declarado `abstract` (§7.2.5.2.1). Se puede declarar un método `final` (§7.2.5.2.3), en cuyo caso no se puede ocultar o anular.

Los nombres de los métodos pueden estar sobrecargados (§7.2.5.6).

Los constructores (§7.2.6) son similares a los métodos, pero no pueden invocarse directamente mediante una llamada de método; se utilizan para inicializar nuevas instancias de clase. Al igual que los métodos, pueden estar sobrecargados (§7.2.6.3).

## 7.2.2 Declaración de una clase

Una declaración de clase especifica un nuevo tipo de referencia.

```
<class_declaration> ::= <class_modifiers> "class" id "{"<class_body>"}"  
|<class_modifiers> id1 "extends" id2 "{"<class_body>"}"
```

*Sintaxis 1: declaración de una clase*

*Fuente: Elaboración propia.*

Consideraciones:

- El identificador en una declaración de clase especifica el nombre de la clase.
- Es un error en tiempo de compilación si una clase tiene el mismo nombre simple que cualquiera de sus clases adjuntas, es decir, las declaradas en su cuerpo.

### 7.2.2.1 Modificadores de clase

Una declaración de clase puede incluir modificadores de clase.

```
<class_modifiers> ::= <class_modifiers> <class_modifier>  
|<class_modifier>  
| €  
  
<class_modifier> ::= "public" | "protected" | "private"  
| "abstract" | "static" | "final"
```

*Sintaxis 2: modificadores de clase*

*Fuente: Elaboración propia.*

Consideraciones:

- El modificador de acceso `public` se refiere solo a clases de nivel superior no a clases locales o anidadas y define que se puede acceder a la clase desde cualquier lugar.
- El modificador de acceso `protected` se refiere solo a las clases miembro dentro de una clase que contiene directamente una declaración de clase y definirá clases miembro que podrán ser accedidas únicamente por subclases.

- El modificador de acceso `private` se refiere solo a las clases miembro dentro de una clase que contiene directamente una declaración de clase y definirá clases miembro que podrán ser accedidas únicamente por la misma clase.
- El modificador `static` se refiere solo a las clases miembro, no a las clases de nivel superior o local.
- Es un error en tiempo de compilación si el mismo modificador aparece más de una vez en una declaración de clase o si aparecen dos o más modificadores de acceso.
- Las variables locales y los parámetros de un procedimiento no tendrán modificador de acceso puesto que sólo podrán ser accedidos desde el propio procedimiento.
- Si una clase no tiene declarado un modificador de acceso esta se tomará como `public`.
- Si dos o más modificadores de clase (distintos) aparecen en una declaración de clase, entonces es habitual, aunque no es obligatorio, que aparezcan en el orden consistente con el que se muestra arriba en la producción de `<class_modifier>..`

#### 7.2.2.1.1 Clases abstractas

Una clase `abstract` es una clase que está incompleta, o que se considera incompleta.

```
abstract class Point {
    int x = 1, y = 1;
    void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
        alert();
    }
    abstract void alert();
}
abstract class ColoredPoint extends Point { int color;}
class SimplePoint extends Point {
    void alert() {
        println("Alerta");
    }
}
```

*Ejemplo 4: implementación de una clase abstracta  
Fuente: Elaboración propia.*

Consideraciones:

- Las clases pueden tener métodos `abstract` (§7.2.5.2.1), es decir, métodos que se declaran pero aún no se implementan, solo si son clases `abstract`.
- Si una clase no se declaró como `abstract` y contiene un método `abstract`, se produce un error en tiempo de compilación.
- Una clase C tiene métodos `abstract` si se cumple alguna de las siguientes condiciones:

- C contiene explícitamente una declaración de un método `abstract` (§7.2.5.2.1).
- Cualquiera de las superclases de C tiene un método `abstract` y C no declara ni hereda un método que lo implemente.
- Es un error en tiempo de compilación si se intenta crear una instancia de una clase `abstract` utilizando una expresión de creación de instancia de clase (§7.7.3).
- Se puede crear una instancia de una subclase de una clase `abstract` que no es en sí misma `abstract`, lo que da como resultado la ejecución de un constructor.

#### 7.2.2.1.2 Clases finales

Se puede declarar una clase `final` si su definición está completa y no se desean ni requieren subclases.

Consideraciones:

- Es un error en tiempo de compilación si el nombre de una clase `final` aparece en la cláusula `extends` (§7.2.2.2) de otra declaración de clase; esto implica que una clase `final` no puede tener ninguna subclase.
- Es un error en tiempo de compilación si una clase se declara a la vez `final` y `abstract`, porque la implementación de tal clase nunca podría completarse.
- Debido a que una clase `final` nunca tiene subclases, los métodos de una final clase nunca se anulan (§7.2.5.5).

#### 7.2.2.2 Superclases y subclases

La cláusula `extends` opcional en una declaración de clase normal especifica la superclase directa de la clase actual.

Consideraciones:

- Tras la cláusula `extends` se debe nombrar un tipo de clase accesible o se produce un error en tiempo de compilación.
- Si tras la cláusula `extends` se especifica un nombre de una clase que es `final`, entonces se produce un error en tiempo de compilación, ya que las clases `final` no pueden tener subclases.
- Es un error en tiempo de compilación si se nombra la misma clase.
- Se dice que una clase es una subclase directa de su superclase directa. La superclase directa es la clase de cuya implementación se deriva la implementación de la clase actual.

```
class Point {int x, y;}
final class ColoredPoint extends Point {int color;}
class Colored3DPoint extends Point {
    void alert() {
        println("Alerta");
    }
}
```

*Ejemplo 5: relación entre superclases y subclases*

Las relaciones en el Ejemplo 5 son las siguientes:

- La clase Point es una subclase directa de Object.
- La clase Object es la superclase directa de la clase Point.
- La clase ColoredPoint es una subclase directa de la clase Point.
- La clase Point es la superclase directa de la clase ColoredPoint.
- La declaración de clase Colored3dPoint provoca un error en tiempo de compilación porque intenta extender la clase final ColoredPoint.

### 7.2.2.3Cuerpo de clase y declaraciones de miembro

Un cuerpo de clase puede contener declaraciones de miembros de la clase, es decir, campo, métodos (§7.2.5), clases (§7.2). Un cuerpo de clase también puede contener declaraciones de constructores (§7.2.6) para la clase.

```
<class_body> ::= <class_body> <class_body_dec>
                | <class_body_dec>

<class_body_dec> ::= <field_declaration>
                    | <method_declaration>
                    | <class_declaration>
                    | <constructor_declaration>
```

*Sintaxis 3: modificadores de clase*  
*Fuente: Elaboración propia.*

## 7.2.3 Miembros de una clase

Los miembros de un tipo de clase son todos los siguientes:

- Miembros heredados de su superclase directa, excepto en la clase Object, que no tiene superclase directa
- Miembros declarados en el cuerpo de la clase

Consideraciones:

- Los miembros de una clase que se declaran `private` no son heredados por las subclases de esa clase.
- Solo los miembros de una clase declarados `protected` o `public` son heredados por subclases declaradas en un paquete que no sea aquel en el que se declara la clase.
- Los constructores no son miembros y, por lo tanto, no se heredan.

### 7.2.3.1Tipo de un miembro

Para un campo, su tipo.

Para un método, una dupla que consiste en:

- Tipos de argumento: una lista de los tipos de argumentos para el miembro del método.
- Tipo de retorno: el tipo de retorno del miembro del método.



### 7.2.3.2 Alcance de un miembro

Los campos, los métodos y los tipos de miembros de un tipo de clase **pueden tener el mismo nombre**, ya que se usan en diferentes contextos y están desambiguados por el procedimiento de búsqueda de **bloque anidado más cercano utilizando tablas de símbolos por alcance** (ver sección 2.71 de libro de texto del curso).

Consideraciones:

- El alcance de una declaración es la región del programa dentro de la cual se puede hacer referencia a la entidad declarada mediante un nombre simple, siempre que esté visible.
- Se dice que una declaración está dentro del alcance en un punto particular de un programa si y solo si el alcance de la declaración incluye ese punto.
- El alcance de un tipo importado por una declaración de importación de un solo tipo es todas las declaraciones de clase en la unidad de compilación (archivo) en la que aparece la sentencia `import`, así como cualquier anotación en la declaración del paquete (si corresponde) de la unidad de compilación.
- El alcance de una declaración de un miembro `m` declarado o heredado por una clase de tipo `C` es el cuerpo completo de `C`, incluidas las declaraciones de tipo anidadas.
- El alcance de un parámetro formal de un método (§8.4.1) o constructor (§8.8.1) es el cuerpo completo del método o constructor.
- El alcance de una declaración de clase local inmediatamente encerrada por un bloque (§7.6.1) es el resto del bloque de encierro inmediato, incluida su propia declaración de clase.
- El alcance de una declaración de variable local en un bloque (§14.4) es el resto del bloque en el que aparece la declaración, comenzando con su propio inicializador e incluyendo cualquier otro declarante a la derecha en la declaración de la declaración de variable local.
- El alcance de una variable local declarada en la parte `<for_init>` de una sentencia `for` iterador (§7.6.9.3.1) incluye todo lo siguiente:
  - Su propio inicializador
  - Cualquier otro declarante a la derecha en la parte `<for_init>` de la sentencia `for`.
  - Las partes `<expression>` y `<for_update>` de la sentencia `for`.
  - El bloque de sentencias
- El alcance de una variable local declarada en la parte `<formal_parameter>` de una sentencia `for each` (§7.6.9.3.2) es el bloque de sentencias.
- El alcance de un parámetro de un controlador de excepciones que se declara en una cláusula `catch` de una sentencia `try` (§7.6.8) es el bloque completo asociado con el `catch`.

### 7.2.3.3 Control de acceso

- Un miembro (clase, campo o método) de un tipo de referencia (clase o arreglo) o un constructor de un tipo de clase es accesible solo si el tipo es accesible y el miembro o constructor está declarado para permitir el acceso:
  - Si se declara el miembro o el constructor `public`, entonces se permite el acceso.
  - De lo contrario, si se declara el miembro o el constructor `private`, entonces se permite el acceso si y solo si ocurre dentro del cuerpo de la clase de nivel superior que encierra la declaración del miembro o constructor.
  - De lo contrario, decimos que hay acceso predeterminado, es decir, es `public`.

### 7.2.4 Declaración de un campo

Las variables de un tipo de clase son introducidas por declaraciones de un campo:

```
<field_declaration> ::= <field_modifiers> <type> <variable_declarators> ";"  
  
<variable_declarators> ::= <variable_declarators> "," <variable_declarator>  
                           | <variable_declarator>  
<variable_declarator> ::= <variable_declarator_id> "=" <variable_initializer>  
                           | <variable_declarator_id>  
<variable_declarator_id> ::= <variable_declarator_id> "[" "]"  
                           | id  
<variable_initializer> ::= <expression>  
                           | <array_initializer>
```

*Sintaxis 4: declaración de campo(s)*  
*Fuente: Elaboración propia.*

#### 7.2.4.1 Modificadores de campo

Una declaración de campo puede incluir modificadores.

```
<field_modifiers> ::= <field_modifiers> <field_modifier>  
                   | <field_modifier>  
                   | ε  
  
<field_modifier> ::= "public" | "protected" | "private"  
                   | "static" | "final"
```

*Sintaxis 5: modificadores de campo*  
*Fuente: Elaboración propia.*

Consideraciones:

- Los modificadores de acceso `public`, `protected` y `private` se discuten en §7.2.3.3.
- Es un error de tiempo de compilación si el mismo modificador aparece más de una vez en una declaración de campo, o si una declaración de campo tiene más de uno de los modificadores de acceso `public`, `protected` y `private`.

#### 7.2.4.1.1 Campos estáticos

Si se declara un campo `static`, existe exactamente una y solo una definición del campo, sin importar cuántas instancias (posiblemente cero) existan de la clase. Un campo estático, a veces llamado variable de clase, se define cuando se prepara la clase (§7.2.7).

Un campo que no está declarado `static` (a veces llamado campo non-static) se denomina variable de instancia. Cada vez que se crea una nueva instancia de una clase (§7.7.3), se crea una nueva variable asociada con esa instancia para cada variable de instancia declarada en esa clase o cualquiera de sus superclases.

```
class Point {
    int x, y, useCount;
    Point(int x, int y) { this.x = x; this.y = y; }
    static final Point origin = new Point(0, 0);
}
class Test {
    public static void main() {
        Point p = new Point(1,1);
        Point q = new Point(2,2);
        p.x = 3;
        p.y = 3;
        p.useCount++;
        p.origin.useCount++;
        println("(" + q.x + "," + q.y + ")");
        println(q.useCount);
        println(q.origin == Point.origin);
        println(q.origin.useCount);
    }
}
> (2,2)
> 0
> true
> 1
```

*Ejemplo 6: campos estáticos  
Fuente: Elaboración propia.*

#### 7.2.4.1.2 Campos finales

Un campo puede ser declarado final (§7.1.3). Se pueden declarar tanto las variables de clase como las de instancia (los campos static y los non-static) final.

#### 7.2.4.2 Inicialización de campos

Si un declarador de campo contiene un inicializador de variable, entonces tiene la semántica de una asignación (§7.6.4) a la variable declarada.

Consideraciones:

- Si el declarador es para una variable de clase (es decir, un campo static), entonces se evalúa el inicializador de la variable y la asignación se realiza exactamente una vez, cuando se prepara la clase (§7.2.7).

- Si el declarador es para una variable de instancia (es decir, un campo que no lo es static), entonces se evalúa el inicializador de la variable y la asignación se realiza cada vez que se crea una instancia de la clase (§7.7.3).
- Los inicializadores de variables también se utilizan en las sentencias de declaración de variables locales (§7.6.3), donde se evalúa el inicializador y la asignación se realiza cada vez que se ejecuta la declaración de variables locales.

```
class Point {
    int x = 1, y = 5;
}
class Test {
    public static void main() {
        Point p = new Point();
        println(p.x + ", " + p.y);
    }
}
> 1, 5
```

*Ejemplo 7: inicialización de campos  
Fuente: Elaboración propia.*

#### 7.2.4.2.1 Inicializadores para variables de clase

Consideraciones:

- Si una referencia por nombre simple a cualquier variable de instancia ocurre en una expresión de inicialización para una variable de clase, entonces ocurre un error en tiempo de compilación.
- Si la palabra clave `this` (§7.7.1) o la palabra clave `super`, aparecen en una expresión de inicialización para una variable de clase, entonces se produce un error en tiempo de compilación.
- En el tiempo de ejecución, los campos `static` que son `final` y son inicializados con expresiones constantes se inicializan primero.

#### 7.2.4.2.2 Inicializadores para variables de instancia

Las expresiones de inicialización para las variables de instancia pueden usar el nombre simple de cualquier variable estática declarada o heredada por la clase, incluso una cuya declaración ocurra textualmente más tarde.

```
class Test {
    double i = j;
    static int j = 1;
}
```

*Ejemplo 8: inicialización de variables de instancia  
Fuente: Elaboración propia.*

#### 7.2.4.2.3 Restricciones en el uso de los campos durante la inicialización

La declaración de un miembro debe aparecer textualmente antes de que se use solo si el miembro es una variable de instancia de una clase C y se cumplen todas las condiciones siguientes:

- El uso no está en el lado izquierdo de una asignación.
- El uso es a través de un nombre simple.
- C es la clase más interna que incluye el uso.

Es un error de tiempo de compilación si alguno de los tres requisitos anteriores no se cumple.

#### 7.2.5 Declaración de un método

Un método declara un código ejecutable que se puede invocar, pasando un número fijo de valores como argumentos.

```
<method_declaration> ::= <method_header> <statement_block>
                        | <method_header> ";"
<method_header> ::= <method_modifiers> <result> <method_declarator> id "("
<formal_parameters> ")"

<method_declarator> ::= <method_declarator> "[" "]"
                      | "[" "]"
                      | ε
```

*Sintaxis 6: declaración de un método*

*Fuente: Elaboración propia.*

Consideraciones:

- El no terminal `<formal_parameters>` se describe en §7.2.5.1, `<method_modifiers>` en §7.2.5.2, `<result>` en §7.2.5.3, `<statement_block>` en §7.6.1.
- El identificador en un `<method_declarator>` puede usarse para referirse al método.
- Es un error de tiempo de compilación para el cuerpo de una clase el declarar como miembros dos métodos con el mismo nombre.
- Es un error en tiempo de compilación si una declaración de método abstract tiene un bloque para su cuerpo.
- Es un error en tiempo de compilación si una declaración de método no es abstract tiene un punto y coma para su cuerpo.
- Si se debe proporcionar una implementación para un método declarado void, pero la implementación no requiere ningún código ejecutable, el cuerpo del método debe escribirse como un bloque que no contenga declaraciones: " {}".
- Si se declara un método void, entonces su cuerpo no debe contener ningún return (§7.6.5.3) que tenga una `<expression>`, o se produce un error en tiempo de compilación.

- Si se declara que un método tiene un tipo de retorno, entonces cada (§7.6.5.3) en su cuerpo debe tener una Expresión, o se produce un error en tiempo de compilación.

### 7.2.5.1 Parámetros formales

Los parámetros formales de un método o constructor, si los hay, se especifican mediante una lista de especificadores de parámetros separados por comas. Cada especificador de parámetro consta de un tipo (opcionalmente precedido por el modificador `final`) y un identificador (opcionalmente seguido de corchetes) que especifica el nombre del parámetro.

Si un método o constructor no tiene parámetros formales, solo aparece un par de paréntesis vacíos en la declaración del método o constructor.

```
<formal_parameters> ::= <formal_parameters> "<formal_parameter>
                        | <formal_parameter>

<formal_parameter> ::= "final" <type> <variable_declarator_id>
                        | <type> <variable_declarator_id>

<variable_declarator_id> ::= <variable_declarator_id> "[" "]"
                        | id
```

*Sintaxis 7: parámetros formales  
Fuente: Elaboración propia.*

Consideraciones:

- Es un error de tiempo de compilación para un método o constructor para declarar dos parámetros formales con el mismo nombre. (Es decir, sus declaraciones mencionan el mismo identificador).
- Es un error en tiempo de compilación si un parámetro formal que se declara `final` se asigna dentro del cuerpo del método o constructor.

### 7.2.5.2 Modificadores de un método

Una declaración de método puede incluir modificadores.

```
<method_modifiers> ::= <method_modifiers> <method_modifier>
                        | <method_modifier>
                        | ε

<method_modifiers> ::= "public" | "protected" | "private"
                        | "abstract" | "static" | "final"
```

*Sintaxis 8: modificadores de un método  
Fuente: Elaboración propia.*

Consideraciones:

- Los modificadores de acceso `public`, `protected` y `private` se discuten en §7.2.3.3.
- Es un error de tiempo de compilación si el mismo modificador aparece más de una vez en una declaración de campo, o si una declaración de campo tiene más de uno de los modificadores de acceso `public`, `protected` y `private`.

#### 7.2.5.2.1 Métodos abstract

Una declaración de método `abstract` introduce el método como miembro, pero no proporciona una implementación.

Consideraciones:

- La declaración de un método `abstract` m debe aparecer directamente dentro de una clase `abstract` (llámemosla A). De lo contrario se produce un error en tiempo de compilación.
- Cada subclase de A que no sea `abstract` debe proporcionar una implementación m, o se produce un error en tiempo de compilación.
- Sería imposible para una subclase implementar un método `abstract` que sea `private`, porque los métodos `private` no son heredados por las subclases; por lo tanto, tal método nunca podría ser utilizado.
- Una clase `abstract` puede sobrescribir (§7.2.5.5) un método `abstract` proporcionando otra declaración `abstract` de método.
- Un método de instancia que no es `abstract` puede ser sobrescrito por un `abstract` método.

#### 7.2.5.2.2 Métodos static

Un método que se declara `static` se llama un método de clase. Un método que no se declara `static` se denomina método de instancia y, a veces, se denomina método non-static.

Consideraciones:

- Un método de clase siempre se invoca sin hacer referencia a un objeto en particular. Es un error de tiempo de compilación intentar referenciar el objeto actual usando la palabra clave `this` (§7.7.1) o la palabra clave `super`.
- Un método de instancia siempre se invoca con respecto a un objeto, que se convierte en el objeto actual al que se refieren las palabras clave `this` (§7.7.1) o la palabra clave `super` durante la ejecución del cuerpo del método.

#### 7.2.5.2.3 Métodos final

Se puede declarar un método `final` para evitar que las subclases lo anulen u oculten.

Consideraciones:

- Es un error de tiempo de compilación intentar sobrescribir u ocultar un final método.
- Un método `private` y todos los métodos declarados inmediatamente dentro de una clase `final` ( §8.1.1.2 ) se comportan como si fueran `final`, ya que es imposible sobrescribirlos.
- En el tiempo de ejecución, un generador de código u optimizador puede poner "en línea" el cuerpo de un método `final`, reemplazando una invocación del método con el código en su cuerpo. El proceso de alineación debe preservar la semántica de la invocación del método.

### 7.2.5.3Firma de un método

Dos métodos tienen la misma firma si tienen el mismo nombre y tipo de argumento.

Dos declaraciones de método o constructor  $M$  y  $N$  tienen los mismos tipos de argumentos si se cumplen todas las condiciones siguientes:

- Tienen el mismo número de parámetros formales (posiblemente cero)
- Tienen el mismo número de parámetros de tipo (posiblemente cero)
- Sean  $A_1, \dots, A_n$  los tipos de los parámetros  $M$  y sean  $B_1, \dots, B_n$  los tipos de los parámetros de  $N$ . Para todo  $1 \leq i \leq n$  se cumple que  $A_i = B_i$ .

Es un error en tiempo de compilación declarar dos métodos con firmas iguales en una clase.

### 7.2.5.4Tipo de retorno

El tipo de retorno en una declaración de método declara el tipo de valor que el método devuelve (el tipo de retorno) o usa la palabra clave `void` para indicar que el método no devuelve un valor.

```
<result> ::= <type>  
          | "void"
```

*Sintaxis 9: tipo de retorno de un método  
Fuente: Elaboración propia.*

### 7.2.5.5Herencia, anular y esconder

Una clase  $C$  hereda de su superclase directa todos los métodos `abstract` y non-abstract de la superclase que son `public` o `protected` o son declaradas con acceso por defecto, esto si no son sobrescritos (§7.2.5.5.1) u ocultados (§7.2.5.5.2) por una declaración en la clase.

Consideraciones:

- Si, por ejemplo, una clase declara dos métodos `public` con el mismo nombre (§7.2.5.6), y una subclase invalida uno de ellos, la subclase todavía hereda el otro método.



- Si el método no heredado se declara en una clase, o el método no heredado se declara en una interfaz y la nueva declaración es `abstract`, entonces se dice que la nueva declaración lo anulará.
- Si el método no heredado es `abstract` y la nueva declaración no lo es `abstract`, se dice que la nueva declaración lo implementa.

#### 7.2.5.5.1 Sobre escritura de métodos

Un método de instancia  $m_1$ , declarado en la clase  $C$ , anula otro método de instancia  $m_2$ , anteponiendo al método `@Override`, permitiendo así, que se tengan dos métodos con la misma firma, el cual estará declarado en la clase  $A$  si todos los siguientes son verdaderos:

- $C$  es una subclase de  $A$ .
- La firma de  $m_1$  igual a la de  $m_2$
- Ya sea:
  - $m_2$  es `public`, `protected` o declarado con acceso predeterminado, o
  - $m_1$  sobrescribe un método  $m_3$  ( $m_3$  distinto de  $m_1$  y  $m_2$ ), tal que  $m_3$  sobrescribe a  $m_2$ .

Además, si  $m_1$  no es `abstract`, se dice que  $m_1$  implementa todas y cada una de las declaraciones de los métodos que sobrescribe.

Consideraciones:

- Es un error en tiempo de compilación si un método de instancia invalida un método `static`.
- Se puede acceder a un método sobrescrito utilizando una expresión de invocación de método (§7.7.6) que contiene la palabra clave `super`.

#### 7.2.5.5.2 Ocultamiento de métodos

Si una clase declara un método `static`  $m$ , entonces se dice que la declaración de  $m$  oculta cualquier método  $m'$  que llegará a poseer la misma firma

Consideraciones:

- Es un error en tiempo de compilación si un método `static` oculta un método de instancia.

#### 7.2.5.6 Sobrecarga de métodos

Si dos métodos de una clase (tanto si están declarados en la misma clase, o si son heredados por una clase, o uno declarado y uno heredado) tienen el mismo nombre, pero las firmas no son equivalentes para realizar una sobreescritura, entonces se dice que el nombre del método es sobrecargado. Este hecho no causa ninguna dificultad y nunca se traduce en un error de compilación.

Cuando se invoca un método (§7.7.6), el número de argumentos reales (y cualquier tipo de argumento explícito) y los tipos de tiempo de compilación de los argumentos se utilizan, en tiempo de compilación, para determinar la firma del método que se invocará (§7.2.5.3).

Si el método que se debe invocar es un método de instancia, el método real que se invocará se determinará en el tiempo de ejecución, utilizando la búsqueda dinámica de métodos<sup>2</sup>.

## 7.2.6 Constructor

Se utiliza un constructor en la creación de un objeto que es una instancia de una clase.

```
<constructor_declaration> ::=  
    <constructor_modifiers> <constructor_declarator> <statement_block>  
  
<constructor_declarator> ::= id "(" <formal_parameters> ")"
```

*Sintaxis 10: declaración de un constructor  
Fuente: Elaboración propia.*

```
class Point {  
    int x, y;  
    Point(int x, int y) { this.x = x; this.y = y; }  
}
```

*Ejemplo 9: declaración de un constructor  
Fuente: Elaboración propia.*

Consideraciones:

- El id en la producción <constructor\_declarator> debe ser el nombre simple de la clase que contiene la declaración del constructor; de lo contrario se produce un error en tiempo de compilación.
- Es un error en tiempo de compilación declarar dos constructores con firmas iguales.

### 7.2.6.1 Modificadores para el constructor

A continuación, se definen los posibles modificadores para un constructor

```
<constructor_modifiers> ::= "public" | "protected" | "private" | €
```

*Sintaxis 11: modificadores de un constructor  
Fuente: Elaboración propia.*

Consideraciones:

- Los modificadores de acceso `public`, `protected` y `private` se discuten en §7.2.3.3.

---

<sup>2</sup> En tiempo de ejecución, la invocación del método requiere cinco pasos. Primero, se puede calcular una referencia objetivo. En segundo lugar, se evalúan las expresiones de los argumentos. En tercer lugar, se comprueba la accesibilidad del método a invocar. Cuarto, se localiza el código real para el método a ejecutar. En quinto lugar, se crea un nuevo marco de activación, se realiza la sincronización si es necesario y el control se transfiere al código del método.

- Es un error de tiempo de compilación si el mismo modificador aparece más de una vez en una declaración de campo, o si una declaración de campo tiene más de uno de los modificadores de acceso `public`, `protected` y `private`.
- Si no se especifica ningún modificador de acceso para el constructor de una clase normal, el constructor tiene acceso predeterminado.
- Un constructor no se hereda, por lo que no es necesario declararlo `final`.
- Un constructor `abstract` nunca podría ser implementado.
- Un constructor siempre se invoca con respecto a un objeto, por lo que no tiene sentido que lo sea un constructor `static`.

### 7.2.6.2 Invocaciones explícitas de constructores

Es posible invocar un constructor explícitamente sin necesidad de instanciar un objeto. Las declaraciones explícitas de invocación del constructor se pueden dividir en dos tipos:

- Las invocaciones de constructor alternativo comienzan con la palabra clave `this` (posiblemente precedida por argumentos de tipo explícito). Se utilizan para invocar un constructor alternativo de la misma clase.
- Las invocaciones de constructores de superclase comienzan con la palabra clave `super` (posiblemente precedida por argumentos de tipo explícito) o una expresión primaria. Se utilizan para invocar a un constructor de la superclase directa.

```
<explicit_constructor_invocation> ::= "this" "(" <arguments> ")" ";"
                                     | "super" "(" <arguments> ")" ";"
```

*Sintaxis 12: invocaciones explícitas de constructores  
Fuente: Elaboración propia.*

### 7.2.6.3 Sobrecarga de constructores

Al igual que los métodos, los constructores pueden sobrecargarse ver sección §7.2.5.6.

### 7.2.6.4 Constructor predeterminado

Si una clase no contiene declaraciones de constructor, se declara implícitamente un constructor predeterminado sin parámetros formales y sin cláusula. Si la clase que se declara es la clase `Object`, entonces el constructor predeterminado tiene un cuerpo vacío. De lo contrario, el constructor predeterminado simplemente invoca al constructor de superclase sin argumentos.

## 7.2.7 Preparación de una clase

La preparación implica crear los campos `static` (variables de clase y constantes) para una clase e inicializar dichos campos a los valores predeterminados (§7.1.4). Esto no requiere la ejecución de ningún código fuente; los inicializadores explícitos para campos estáticos se ejecutan como parte de la inicialización, no de preparación.

## 7.3 Arreglos

En **Coline**, los arreglos son objetos y pueden asignarse a variables de tipo `Object` (§5.1.2.1). Todos los métodos de clase `Object` pueden ser invocados en un arreglo.

Un arreglo contiene una serie de variables. El número de variables puede ser cero, en cuyo caso se dice que el arreglo está vacío. Las variables contenidas en un arreglo no tienen nombres; en su lugar, se hace referencia a ellas mediante expresiones de acceso al arreglo que utilizan valores de índice entero no negativos. Estas variables se llaman los componentes del arreglo.

Si un arreglo tiene  $n$  componentes, decimos que  $n$  es la longitud del arreglo; se hace referencia a los componentes del arreglo utilizando índices enteros de  $0 \leq i < n$ .

Todos los componentes de un arreglo tienen el mismo tipo, denominado tipo de componente del arreglo. Si el tipo de componente de un arreglo es  $T$ , entonces el tipo del arreglo en sí está escrito  $T[]$ .

El tipo de componente de un arreglo puede ser un tipo de arreglo. Los componentes de dicho arreglo pueden contener referencias a subarreglos. Si a partir de cualquier tipo de arreglo, uno considera su tipo de componente, y luego (si también es un tipo de arreglo) el tipo de componente de ese tipo, y así sucesivamente, eventualmente debe alcanzar un tipo de componente que no sea un tipo de arreglo; esto se denomina tipo de elemento del arreglo original, y los componentes en este nivel de la estructura de datos se denominan elementos del arreglo original.

### 7.3.1 Tipos de arreglos

Los tipos de arreglos se utilizan en declaraciones y en expresiones de conversión (§7.7.7).

Un tipo de arreglo se escribe como el nombre de un tipo de elemento seguido de un número de pares vacíos de corchetes  $[]$ . El número de pares de corchetes indica la profundidad del anidamiento de arreglo. La longitud de un arreglo no es parte de su tipo.

El tipo de elemento de un arreglo puede ser cualquier tipo, ya sea primitivo o de referencia. En particular:

- Se permiten arreglos con un tipo de clase `abstract` como el tipo de elemento. Un elemento de un arreglo de este tipo puede tener como valor una referencia nula o una instancia de cualquier subclase de la clase `abstract` que no sea en sí misma `abstract`.

La superclase directa de un tipo de arreglo es `Object`.

### 7.3.2 Declaración de arreglos

Una variable de tipo arreglo contiene una referencia a un objeto. La declaración de una variable de tipo de arreglo no crea un objeto de arreglo ni asigna ningún espacio para los componentes del arreglo. Crea solo la variable, que puede contener una referencia a un arreglo. A continuación, se definirá la sintaxis para la declaración de arreglos u

opcionalmente arreglos de arreglos. Al momento de declarar arreglos será necesario definir el tipo de dato de los elementos que contendrá.

```
<array_declaration> ::= <type> id ("[" "]"")+;
```

*Sintaxis 13: declaración de arreglos  
Fuente: Elaboración propia.*

```
// este será un arreglo de decimales
double arr1[];

// este será un arreglo de arreglos de decimales
double arr2[][];

// este será un arreglo de arreglos de arreglos de decimales
double arr3[][][];
```

*Ejemplo 10: declaración de arreglos  
Fuente: Elaboración propia.*

### Consideraciones:

- Una vez que se crea un arreglo, su longitud nunca cambia. Para hacer que una variable de arreglo se refiera a un arreglo de diferente longitud, se debe asignar una referencia a un arreglo diferente a la variable.
- Una sola variable de tipo arreglo puede contener referencias a arreglos de diferentes longitudes, porque la longitud de un arreglo no es parte de su tipo.

### 7.3.3 Asignación de un valor a posiciones de un arreglo

Se accede a un componente de un arreglo mediante una expresión de acceso al arreglo (§7.7.4) que consiste en una expresión cuyo valor es una referencia de arreglo seguida de una expresión de indexación entre “[” y “]”, como en  $A[i]$ .

Un arreglo con longitud  $n$  puede ser indexada por los enteros  $0 \leq i < n$ .

```
class Gauss {
    public static void main() {
        int ia[] = new int[101];
        for (int i = 0; i < ia.length; i++){ ia[i] = i;}
        int sum = 0;
        for (int e : ia) {sum = sum + e;}
        println(sum);
    }
}
> 5050
```

*Ejemplo 11: asignación de un valor a posiciones de un arreglo  
Fuente: Elaboración propia.*

### 7.3.4 Inicialización de arreglos

A continuación, se definirá la sintaxis para la declaración de arreglos u opcionalmente arreglos de arreglos. Al momento de declarar arreglos será necesario definir el tipo de dato de los elementos que contendrá. Y la inicialización se podrá de hacer de dos formas, como se observa en el Ejemplo 12.

```
<arrayinit> ::= id "=" "new" <type> ("[" <expression> "]" )+;
```

*Sintaxis 14: inicialización de arreglos  
Fuente: Elaboración propia*

```
// declaración e inicialización de un  
// arreglo de decimales  
int x = 5 + 0;  
double arr1[] = new double [x];  
//alternativa de declaración e inicialización  
double arr2[][] = {{1,2},{3,4},{5,6}};  
// inicialización de un arreglo de decimales  
arr1 = new double [x];  
// alternativa para la inicialización de un  
// arreglo de decimales  
arr2 = {{1,2},{3,4},{5,6}};
```

*Ejemplo 12: inicialización de arreglo  
Fuente: Elaboración propia.*

### 7.3.5 Array Store Exception

Para una matriz cuyo tipo es A[] , donde A es un tipo de referencia, una asignación a un componente de la matriz se verifica en el tiempo de ejecución para garantizar que el valor que se asigna sea asignable al componente.

Si el tipo de valor que se está asignando no es compatible con la asignación con el tipo de componente, se lanza un ArrayStoreException.

```
class Point { int x, y; }  
class ColoredPoint extends Point { int color; }  
class Test {  
    public static void main() {  
        ColoredPoint cpa[] = new ColoredPoint[10];  
        Point pa[] = cpa;  
        println(pa[1] == null);  
        try {  
            pa[0] = new Point();  
        } catch (ArrayStoreException e) {  
            println(e);  
        }  
    }  
}  
> true  
> ArrayStoreException
```

*Ejemplo 13: excepción Array Store Exception*

### 7.3.6 Miembro length

El campo `public final` `length` contiene el número de componentes del arreglo. Puede ser positivo o cero.

```
class Gauss {
    public static void main() {
        int arr[][] = new int[2][3];
        println("La primera \"dimensión\" es de tamaño = \" + arr.length);
        println("La primera \"dimensión\" es de tamaño = \" + arr[0].length);
    }
}
> La primera "dimensión" es de tamaño = 2
> La segunda "dimensión" es de tamaño = 3
```

*Ejemplo 14: obtención de la longitud de un arreglo*  
Fuente: Elaboración propia.

### 7.3.7 Un arreglo de tipo char no es un String

En el lenguaje de programación **Coline**, a diferencia de C, un arreglo de `char` no es a `String`. Un objeto `String` es inmutable, es decir, su contenido nunca cambia, mientras que un arreglo de `char` tiene elementos mutables.

## 7.4 Collections de Coline

El lenguaje **Coline** cuenta con una librería llamada Collections que es un conjunto de estructuras dinámicas y genéricas que facilitarán el manejo de elementos de diferentes tipos (referencia y primitivos) las cuales se describirán a continuación.

### 7.4.1 LinkedList

Las LinkedList de **Coline** son listas doblemente enlazadas, las LinkedList una vez sean instanciadas se deberá indicar el tipo de los elementos que contendrá e inicialmente se encontrarán vacías.

#### 7.4.1.1 Declaración e instanciación

```
<linkedlist_declaration> ::= "LinkedList" "<" <type> ">" id ";"
| "LinkedList" "<" <type> ">" id "=" "new" "LinkedList" "<" ">" "(" ")" ";"
```

*Sintaxis 15: declaración e instanciación de una LinkedList*  
Fuente: Elaboración propia

```

public class Test {
    public static void main() {
        class Point {
            int x, y;
            public Point(int x, int y) {
                this.x = x;
                this.y = y;
            }
        }
        LinkedList<Point> points = new LinkedList<>();
        LinkedList<int> grades = new LinkedList<>();
    }
}

```

*Ejemplo 15: declaración e instanciación de una LinkedList*  
Fuente: Elaboración propia.

#### 7.4.1.2 Miembros

- `add(E e)`: Anexa el elemento especificado al final de esta lista. El elemento recibido como argumento debe ser del tipo definido en la declaración de la lista.
- `clear()`: Elimina todos los elementos de esta lista.
- `get(int index)`: Devuelve el elemento en la posición especificada en esta lista.
- `remove(int index)`: Elimina el elemento en la posición especificada en esta lista.
- `set(int index, E element)`: Reemplaza el elemento en la posición especificada en esta lista con el elemento especificado. El elemento recibido como argumento debe ser del tipo definido en la declaración de la lista.
- `size()`: Devuelve el número de elementos en esta lista.



```

public class Test {
    public static void main() {
        class Point {
            int x, y;
            public Point(int x, int y) {
                this.x = x;
                this.y = y;
            }
        }
        LinkedList<Point> points = new LinkedList<>();
        for (int i = 0; i < cadena.length(); i++) {
            for (int j = 0; j < cadena.length(); j++) {
                points.add(new Point(i, j));
            }
        }
        println("Se llenó la lista");
        for (int i = 0; i < points.size(); i++) {
            Point point = points.get(i);
            println("(" + point.x + " , " + point.y + ")");
        }
        points.set(0, new Point(100, 100));
        println("Se modifiko la primera posicion");
        for (Point point : points) {
            println("(" + point.x + " , " + point.y + ")");
        }
        println("Se eliminará el primer elemento de la lista");
        points.remove(0);
        for (Point point : points) {
            println("(" + point.x + " , " + point.y + ")");
        }
    }
}

```

```

> Se llenó la lista
> (0 , 0)
> (0 , 1)
> (0 , 2)
> (1 , 0)
> (1 , 1)
> (1 , 2)
> (2 , 0)
> (2 , 1)
> (2 , 2)
> Se modifiko la primera posicion
> (100 , 100)
> (0 , 1)
> (0 , 2)
> (1 , 0)
> (1 , 1)
> (1 , 2)
> (2 , 0)
> (2 , 1)
> (2 , 2)
> Se eliminará el primer elemento de la lista
> (0 , 1)
> (0 , 2)
> (1 , 0)
> (1 , 1)
> (1 , 2)
> (2 , 0)
> (2 , 1)
> (2 , 2)

```

*Ejemplo 16: uso de una LinkedList  
Fuente: Elaboración propia.*

## 7.5 Excepciones

Cuando un programa viola las restricciones semánticas del lenguaje de programación **Coline**, el compilador señala este error como una excepción. Un ejemplo de tal violación es un intento de indexar fuera de los límites de un arreglo.

En su lugar, el lenguaje de programación **Coline** especifica que se lanzará una excepción cuando se violen las restricciones semánticas y causará una transferencia de control no local desde el punto donde ocurrió la excepción hasta un punto que puede ser especificado por el programador.

Se dice que se lanza una excepción desde el punto donde ocurrió y se dice que se atrapa en el punto al que se transfiere el control.

Los programas también pueden lanzar excepciones explícitamente, usando la sentencia `throw` (§7.6.7).

El uso explícito de la sentencia `throw` proporciona una alternativa al estilo anticuado de manejar las condiciones de error al devolver valores arbitrarios, como el valor entero -1 donde normalmente no se esperaría un valor negativo.

Cada excepción está representada por una instancia de la clase Throwable o una de sus subclases (§). Un objeto de este tipo se puede usar para transportar información desde el punto en el que se produce una excepción al controlador que lo captura. Los manejadores están establecidos por cláusulas `catch` de sentencias `try` (§).

### 7.5.1 Los tipos de las excepciones

Una excepción está representada por una instancia de la clase Throwable (una subclase directa de Object) o una de sus subclases. Throwable y todas sus subclases son, colectivamente, las clases de excepción.

Las clases Exception y Error son subclases directas de Throwable. Exception es la superclase de todas las excepciones a partir de las cuales los programas ordinarios pueden desear recuperar.

Error es la superclase de todas las excepciones de las que normalmente no se espera que los programas ordinarios se recuperen. Error y todas sus subclases son, colectivamente, las clases de error.

La clase RuntimeException es una subclase directa de Exception. RuntimeException es la superclase de todas las excepciones que pueden lanzarse por muchas razones durante la evaluación de expresiones, pero a partir de las cuales aún es posible la recuperación. RuntimeException y todas sus subclases son, colectivamente, las clases de excepción en tiempo de ejecución.

### 7.5.2 La jerarquía

- Throwable
  - Error
    - MachineError
      - HeapOverflowError: ocurre cuando el espacio (64 Megabytes) del Heap se desborda.
      - StackOverflowError: ocurre cuando el espacio (64 Megabytes) del Stack se desborda.
  - Exception
    - IOException
      - FileNotFoundException: cuando mediante la sentencia `read_file` se intenta acceder a una dirección de un archivo inexistente.
    - ReflectiveOperationException
      - ClassNotFoundException: cuando se intenta instanciar un objeto de una clase inexistente
      - NoSuchFieldException: cuando se intenta acceder a un campo inexistente.
      - NoSuchMethodException: cuando se intenta acceder a un método inexistente.

- **RuntimeException**
  - **ArithmeticException**: se da cuando se evalúa una expresión que viola las reglas definidas sobre las operaciones aritméticas.
  - **ArrayStoreException**: ver sección (§7.3.5).
  - **ClassCastException**: se da cuando se realiza una operación de casteo no soportada.
  - **IllegalArgumentException**
    - **NumberFormatException**: Se da en las funciones de casteo explícito con cadenas cuando la misma no posee el formato que debería tener.
  - **IndexOutOfBoundsException**
    - **ArrayIndexOutOfBoundsException**: se da cuando se accede a una posición que no se encuentra entre el tamaño definido para un arreglo.
  - **NullPointerException**: se da cuando se desea utilizar miembros de un objeto que apunta a null.
  - **UncaughtException**: se da cuando una excepción no es manejada por una sentencia try.

#### Consideraciones:

- Una excepción puede declararse como parámetro de una cláusula catch y también puede crearse una nueva instancia de la misma, su único miembro es el atributo message que contendrá una cadena compuesta por su nombre y línea que produjo la excepción.

### 7.5.3 Causas de las excepciones

Se lanza una excepción por una de dos razones:

- Se ejecutó una sentencia **throw** (§).
- El compilador de **Coline** detectó una condición de ejecución anormal:
  - La evaluación de una expresión viola la semántica normal del lenguaje de programación **Coline** (§), como una división entera por cero.

Estas excepciones no se lanzan en un punto arbitrario del programa, sino en un punto donde se especifican como un posible resultado de una evaluación de expresión o ejecución de una declaración.

### 7.5.4 Manejo en tiempo de ejecución de una excepción

Decimos que una cláusula catch puede capturar su (s) clase (s) de excepción.

La clase de excepción detectable de una única cláusula catch es el tipo declarado de su parámetro de excepción (§).

Cuando se lanza una excepción (§), el control se transfiere desde el código que causó la excepción a la cláusula catch de cierre dinámico más cercana, si existe, de una sentencia try (§) que puede manejar la excepción.

Una cláusula o expresión está encerrada dinámicamente por una cláusula catch si aparece dentro del bloque try de la sentencia try de la que forma parte la cláusula catch.

Si una cláusula catch en particular puede manejar una excepción se determina comparando la clase del objeto que se lanzó con las clases de excepción detectables de la cláusula catch. La cláusula catch puede manejar la excepción si una de sus clases de excepción detectables es la clase de la excepción o una superclase de la clase de la excepción.

De manera equivalente, una cláusula catch capturará cualquier objeto de excepción que sea una instancia de (§) una de sus clases de excepción detectables.

La transferencia de control que se produce cuando se lanza una excepción provoca la finalización brusca de las expresiones (§) y las declaraciones (§) hasta que se encuentra una cláusula catch que puede manejar la excepción; la ejecución luego continúa ejecutando el bloque de esa cláusula catch. **El código que causó la excepción nunca se reanuda.** Todas las excepciones son precisas: cuando se produce la transferencia del control, todos los efectos de las declaraciones ejecutadas y las expresiones evaluadas antes del punto desde el cual se produce la excepción deben aparecer. No parece que se hayan evaluado expresiones, declaraciones o partes de las mismas que se produzcan después del punto desde el cual se lanzó la excepción.

Si el código optimizado ha ejecutado especulativamente algunas de las expresiones o declaraciones que siguen al punto en el que se produce la excepción, dicho código debe estar preparado para ocultar esta ejecución especulativa del estado visible para el usuario del programa.

Si no se puede encontrar una cláusula catch que pueda manejar una excepción, entonces se termina el hilo de ejecución del código intermedio (el hilo que encontró la excepción).

## 7.6 Sentencias

### 7.6.1 Bloques

Un bloque es una lista de sentencias, declaraciones de clases locales y sentencias de declaración de variables locales entre llaves.

```
<statement_block> ::= "{"<statement_list>"  
<statement_list> ::= <statement_list> <statement>  
                        | <statement>
```

*Sintaxis 16: bloque de sentencias*

*Fuente: Elaboración propia.*

### 7.6.2 Sentencia import

Instrucción que permitirá importar código de un archivo **Coline**, para así, poder utilizar los elementos del mismo.

```
<statement> ::= "import" string_terminal ";"
```

*Sintaxis 17: sentencia import*

*Fuente: Elaboración propia.*

```
import "archivo1.coline" ;  
println(x);  
> 69
```

*Ejemplo 17: sentencia import*

*Fuente: Elaboración propia.*

Consideraciones:

- Debe existir el archivo, de lo contrario lanzar una excepción.
- Si existieran elementos repetidos, debe lanzar una excepción.
- Se utilizará la clase que posea un método main en el archivo actual y se descartaran los métodos main encontrados en los archivos importados.

### 7.6.3 Declaración de variables

Para la declaración de variables será posible empezar con un modificador de acceso opcional, seguido de un modificador de variable (detallados en el apéndice B) opcional, seguido del tipo de dato de la variable y del identificador que esta tendrá, adicionalmente si se desea, podrá inicializarse con una expresión.

```
<local_var_declaration> ::= <type> <variable_declarators> ";"  
                        | "final" <type> <variable_declarators> ";"  
<variable_declarators> ::= <variable_declarators> ",", <variable_declarator>  
                        | <variable_declarator>  
<variable_declarator> ::= <variable_declarator_id> "=" <variable_initializer>  
                        | <variable_declarator_id>  
<variable_declarator_id> ::= <variable_declarator_id> "[" "]"  
                        | id  
<variable_initializer> ::= <expression>  
                        | <array_initializer>
```

*Sintaxis 18: declaración de una variable local*

*Fuente: Elaboración propia.*

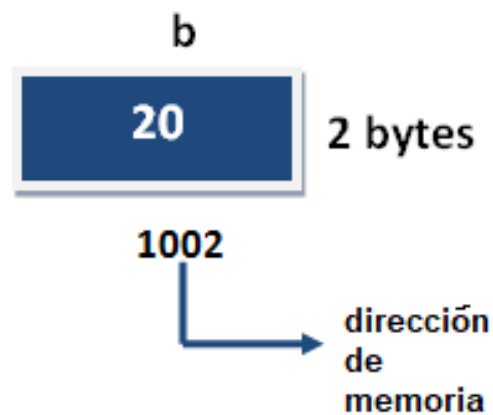
```

public class Test {
    public static void main() {
        boolean var1;
        final int a = 69;
        return 69;
    }
}

```

*Ejemplo 18: declaración de variables*  
*Fuente: Elaboración propia.*

A continuación, se muestra gráficamente la declaración de una variable:



*Figura 15: proceso interno al declarar una variable*  
*Fuente: Elaboración propia.*

Consideraciones:

- No puede declararse una variable con el mismo identificador de una variable existente en el ámbito actual o uno superior, de suceder debe reportarse un error.
- La inicialización es opcional y el tipo de la expresión debe ser el mismo que el declarado, de lo contrario debe reportarse un error.
- A pesar de que la inicialización de una variable al momento de declararla es opcional, para utilizar la misma es necesario se le asigne un valor para inicializarla, de lo contrario tendrá que mostrarse un error de semántica
- Una declaración de variable local, deberá aparecer en el cuerpo de un bloque de sentencias.

### 7.6.4 Asignación de variables

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

```
<assignment> ::= id "=" <expression> ";"
```

*Sintaxis 19: asignación de variables*

*Fuente: Elaboración propia.*

```
public class Test {  
    public static void main() {  
        int a = 20;  
        int b = 0;  
        // asignación directa  
        b = 25;  
        println(b);  
        > 25  
        // asignación en términos de otra variable  
        b = a;  
        println(b);  
        > 20  
        // asignación en términos de una expresión  
        b = 15 * 2 + 10;  
        println(b);  
        > 40  
    }  
}
```

*Ejemplo 19: asignación de variables*

*Fuente: Elaboración propia.*

A continuación, se muestra gráficamente la asignación de una expresión a una variable:



*Figura 16: proceso interno al asignar una variable*

*Fuente: Elaboración propia.*

Consideraciones:

- No puede asignarse un valor a una variable inexistente, el identificador debe coincidir con el identificador de una variable que exista en el ámbito actual o uno superior de suceder debe reportarse un error.
- El tipo de la expresión debe ser el mismo que el declarado, de lo contrario debe reportarse un error.



## 7.6.5 Sentencias de transferencia

**Coline** soportará tres sentencias de salto o transferencia: `break`, `continue` y `return`. Estas sentencias transferirán el control a otras partes del programa.

### 7.6.5.1 Break

La sentencia **break** tendrá dos usos:

- Terminar la ejecución de una sentencia `switch`
- Terminar la ejecución de un ciclo.

```
<break_statement> ::= "break" ";"
```

*Sintaxis 20: sentencia break  
Fuente: Elaboración propia.*

```
public class Test {  
    public static void main() {  
        int cuenta = 0;  
        while (cuenta < 5) {  
            cuenta++;  
            if (cuenta == 3) {  
                break;  
            }  
            println(cuenta);  
        }  
    }  
}  
> 1  
> 2
```

*Ejemplo 20: sentencia break  
Fuente: Elaboración propia.*

Consideraciones:

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

### 7.6.5.2 Continue

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

```
<continue_statement> ::= "continue" ";"
```

*Sintaxis 21: sentencia continue  
Fuente: Elaboración propia.*

```

public class Test {
    public static void main() {
        int cuenta = 0;
        while (cuenta < 5) {
            cuenta++;
            if (cuenta == 3) {
                continue;
            }
            println(cuenta);
        }
    }
}

```

> 1  
 > 2  
 > 4  
 > 5

*Ejemplo 21: sentencia continue  
Fuente: Elaboración propia.*

Consideraciones:

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

### 7.6.5.3 Return

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

```

<return_statement> ::= "return" ";"
                    | "return" <expression> ";"

```

*Sintaxis 22: sentencia return  
Fuente: Elaboración propia.*

```

public class Test {
    public static void main() {
        int x = factorial(5);
        println(x);
        if (x == 120) {
            return;
        }
        println("Esto ya no se imprimirá");
    }
    public static int factorial(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }
}

```

> 120

*Ejemplo 22: sentencia return  
Fuente: Elaboración propia.*

#### Consideraciones:

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

## 7.6.6 Sentencias de selección

### 7.6.6.1 Sentencia If

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

```

<if_statement> ::= <if_list> "else" <statement_block> | <if_list>
<if_list> ::= "if" "(" <expression> ")" <statement_block>
| <if_list> "else" "if" "(" <expression> ")" <statement_block>

```

*Sintaxis 23: sentencia if  
Fuente: Elaboración propia.*

```

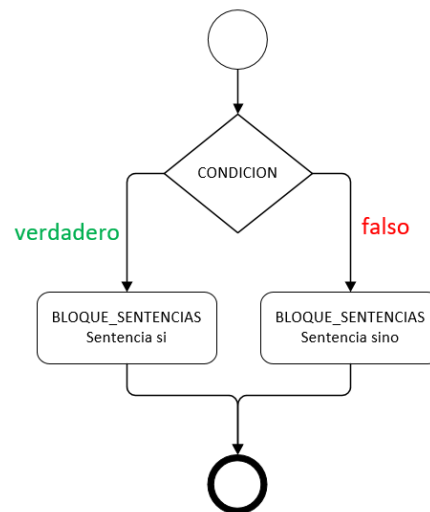
public class Test {
    public static void main() {
        println("Ingrese un numero entero: ");
        int numero = 5;
        if (numero > 0) {
            println("El numero ingresado es positivo");
        } else if (numero < 0) {
            println("El numero ingresado es negativo");
        } else {
            println("El numero ingresado es cero");
        }
    }
}
> Ingrese un numero entero: 5
> El numero ingresado es positivo

```

*Ejemplo 23: sentencia if  
Fuente: Elaboración propia.*

A continuación, se muestra el flujo de la sentencia:

*Figura 17: diagrama de flujo para la sentencia de selección if  
Fuente: Elaboración propia.*



### 7.6.6.2 Switch

Esta sentencia de selección será la bifurcación múltiple, es decir, proporcionará control sobre múltiples alternativas basadas en el valor de una expresión de control.

La sentencia switch compara con el valor seguido de cada `case`, y si coincide, se ejecuta el bloque de sentencias asociadas a dicho `case`, hasta encontrar una instrucción `break`.

```

<switch_statement> ::= "switch" "(" (<expression> ")" <switch_block>

<switch_block> ::=
    "{" (<switch_block_statement_groups>)? (<switch_labels>)? "}"

```

```

<switch_block_statement_groups>::= <switch_block_statement_groups>
    | <switch_block_statement_groups> <switch_block_statement_group>
<switch_block_statement_group>::= <switch_labels> <statement_block>

<switch_labels>::= <switch_label>
    | <switch_labels> <switch_label>

<switch_label>:
    "case" <literal> ":"
    "default" ":"

```

*Sintaxis 24: sentencia switch  
Fuente: Elaboración propia.*

```

public class Test {
    public static void main() {
        switch (y / 50) {
            case 1:
                // sentencias que se ejecutarán si y/50 = 1.
            case 2:
                // sentencias que se ejecutarán si y/50 = 2.
            default:
                // sentencias que se ejecutarán si y/50 no coincide ningún
                // caso.
        }
    }
}

```

*Ejemplo 24: sentencia switch  
Fuente: Elaboración propia.*

#### Consideraciones:

- Si  $i$  es el caso actual y  $n$  el número de casos, si la expresión del caso  $i$  llegara a coincidir con la expresión de control (para  $i < n$ ), se ejecutarán todos los bloques de sentencias entre los casos  $i$  y  $n$  mientras no se encuentre la instrucción `break`.
- Tanto la expresión de control como las expresiones asociadas a cada uno de los casos deberá ser de tipo primitivo, es decir, `char`, `int`, `double` o `boolean`.
- Si ninguno de los casos coincide con la expresión de control se ejecutará el bloque de sentencias asociadas a `default`.

#### 7.6.7 Sentencia throw

Una sentencia `throw` hace que se lance una excepción (§). El resultado es una transferencia inmediata de control que puede aparecer en cualquier bloque de sentencias mientras que se encuentre dentro del cuerpo de una sentencia `try` que capte el valor arrojado. Si no tal sentencia, entonces se lanza una `UncaughtException` y se detiene la ejecución del programa.

```
<throw_statement> ::= "throw" <expression> ";"
```

*Sintaxis 25: sentencia throw*  
*Fuente: Elaboración propia.*

```
public class Test {  
    public static void main() {  
        int divisor = 0;  
        int dividendo = 10;  
        try {  
            if (divisor == 0) {  
                throw new ArithmeticException();  
            }  
        } catch (ArithmeticException ae) {  
            print (e.message) ;  
        }  
    }  
}  
> ArithmeticException en linea 15
```

*Ejemplo 25: sentencia throw*  
*Fuente: Elaboración propia.*

### 7.6.8 Sentencia try catch

Una sentencia `try` ejecuta un bloque. Si se lanza un valor y la sentencia `try` tiene una cláusula `catch` que pueden atraparlo, entonces el control se transferirá a la cláusula `catch`.

```
<try_statement> ::=  
"try" <statement_block> "catch" "(" <declaration> ")"<statement_block>
```

*Sintaxis 26: sentencia try*  
*Fuente: Elaboración propia.*

```
public class Test {  
    public static void main() {  
        int divisor = 0;  
        int dividendo = 10;  
        try {  
            int cociente == dividendo / divisor;  
        } catch (ArithmeticException ae) {  
            print (e.message) ;  
        }  
    }  
}  
> ArithmeticException en linea 12
```

*Ejemplo 26: sentencia try*  
*Fuente: Elaboración propia.*

### 7.6.9 Sentencias cíclicas o bucles

#### 7.6.9.1 While

La sentencia cíclica mientras se utilizará para crear repeticiones de sentencias en el flujo del programa.

El bloque de sentencias se ejecutará mientras la condición sea verdadera (0 a n veces), de lo contrario el programa continuará con su flujo de ejecución normal.

```
<while_statement> ::= "while" "(" <expression> ")" <statement_block>
```

*Sintaxis 27: sentencia while*  
*Fuente: Elaboración propia.*

```
public class Test {  
    public static void main() {  
        int cuenta = 0;  
        while (cuenta < 5) {  
            cuenta++;  
            if (cuenta == 3) {  
                continue;  
            }  
            println(cuenta);  
        }  
    }  
}  
> 1  
> 2  
> 4  
> 5
```

*Ejemplo 27: sentencia while*  
*Fuente: Elaboración propia.*

### 7.6.9.2 Do While

La sentencia cíclica hacer-mientras se utilizará para crear repeticiones de sentencias en el flujo del programa.

El bloque de sentencias se ejecutará una vez y se seguirá ejecutando mientras la condición sea verdadera (1 a n veces), de lo contrario el programa continuará con su flujo de ejecución normal.

```
<do_statement> ::= "do" <statement_block> "while" "(" <expression> ")" ";"
```

*Sintaxis 28: sentencia do*  
*Fuente: Elaboración propia.*

```

public class Test {
    public static void main() {
        int cuenta = 0;
        do {
            cuenta++;
            if (cuenta == 3) {
                continue;
            }
            println(cuenta);
        } while (cuenta < 5);
    }
}

```

> 1  
> 2  
> 4  
> 5

*Ejemplo 28: sentencia do  
Fuente: Elaboración propia.*

### 7.6.9.3 For

#### 7.6.9.3.1 Iterador

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

```

<for_statement> ::=
"for" "(" <for_init> ";" "<expression>"; "<expression>" ")" <statement_block>

<for_init> ::= <declaration>
| <assignment>

```

*Sintaxis 29: sentencia for  
Fuente: Elaboración propia.*

```

public class Test {
    public static void main() {
        for (int cuenta = 0; cuenta < 5; cuenta++) {
            if (cuenta == 3) {
                continue;
            }
            println(cuenta);
        }
    }
}

```

> 0  
> 1  
> 2  
> 4

*Ejemplo 29: sentencia for  
Fuente: Elaboración propia.*



**Inicialización:** Declaración o asignación.

**Condición:** Expresión booleana que se evaluara cada iteración para decidir si se ejecuta el bloque de sentencias o no.

**Actualización:** Sentencia que actualiza la variable de control ya sea con una asignación o una expresión de incremento o decremento

### 7.6.9.3.2 For each

Esta sentencia será utilizada para recorrer arreglos y LinkedList.

```
<foreach_statement> ::=  
"for" "(" <formal_parameter> ":" <expression> ")" <statement_block>
```

*Sintaxis 30: sentencia for each  
Fuente: Elaboración propia.*

```
public class Test {  
    public static void main() {  
        int arreglo[][]={{1,2,3,4,5,6,7,8,9,10}};  
        print("Los elementos del arreglo son: ");  
        for (int i : arreglo[0]) {  
            print(i+" ");  
        }  
    }  
}  
> Los elementos del arreglo son: 1 2 3 4 5 6 7 8 9 10
```

*Ejemplo 30: sentencia for each  
Fuente: Elaboración propia.*

## 7.6.10 Sentencias de entrada/salida

### 7.6.10.1 *Sentencia print*

Instrucción que permitirá imprimir una expresión en la consola del editor en línea.

```
<statement> ::= "print" "(" <expression> ")" ";"  
| "println" "(" <expression> ")" ";"
```

*Sintaxis 31: sentencia println  
Fuente: Elaboración propia.*

```
public class Test {  
    public static void main() {  
        print("Hola ");  
        println("mundo");  
        print(69);  
    }  
}  
> Hola mundo  
> 69
```

*Ejemplo 31: sentencia println  
Fuente: Elaboración propia.*

Consideraciones:

- Si se utiliza la sentencia `print` se incluirá un salto de línea al final, de lo contrario, si es una sentencia `println` no se incluirá un salto de línea al final
- Debe verificarse que la expresión a imprimir tenga un equivalente en `String`.
- En caso que la expresión sea `null`, deberá imprimirse la palabra `null`.

### 7.6.10.2 Leer archivo

Esta función será encargada de leer un archivo del lado del servidor.

Para su traducción a código 3D se utilizará la misma palabra reservada enviando el puntero de la cadena como parámetro que representa la ruta relativa al archivo de aplicación (`server.js`), y el intérprete 3D realizará el acceso al sistema de archivos en el servidor y retornará el contenido del mismo.

```
<read_file_expression> ::= "read_file" "(" <expression> ")"
```

*Sintaxis 32: sentencia read\_file  
Fuente: Elaboración propia.*

```
public class Test {  
    public static void main() {  
        print(read_file("/home/Javier/archivo.txt"));  
    }  
}  
> Hola mundo
```

*Ejemplo 32: sentencia read\_file  
Fuente: Elaboración propia.*

Consideraciones:

- El parámetro es una ruta valida donde se encuentra el archivo a leer.
- Si la ruta del archivo es invalida, es decir, el archivo no existe deberá de lanzar una `FileNotFoundException`.

### 7.6.10.3 Escribir archivo

Esta función será encargada de escribir un archivo recibiendo la ruta y el contenido del mismo del lado del servidor.

Para su traducción a código 3D se utilizará la misma palabra reservada enviando el puntero de la cadena que representa la ruta relativa al archivo de aplicación (`server.js`), así como el que representa el contenido como parámetros y el intérprete 3D realizará el acceso al sistema de archivos en el servidor.

```
<write_file_expression> ::=  
    "write_file" "(" <expression> "," <expression> ")"
```

*Sintaxis 33: sentencia write\_file  
Fuente: Elaboración propia.*

```

public class Test {
    public static void main() {
        write file("/home/Javier/archivo.txt", "Contenido");
    }
}

```

*Ejemplo 33: sentencia write\_file  
Fuente: Elaboración propia.*

Consideraciones:

- El primer parámetro es una ruta valida donde se desea escribir el archivo, esta debe incluir el nombre del mismo.

#### 7.6.10.4 Lectura de datos de parte del usuario (No implementar)

En Coline existirá una función que interrumpirá el flujo normal del programa para pedir un valor al usuario, a través de la consola del editor, el valor ingresado deberá ser almacenado en la variable recibida como parámetro. Si el tipo de dato recibido es diferente del dato esperado deberá de mostrar un mensaje de error y volverá a solicitarlo.

```

<read_console_statement> ::=
    "read" "(" id ")"

```

*Sintaxis 34: sentencia read  
Fuente: Elaboración propia.*

```

public class Test {
    public static void main() {
        int Id_1;
        String Id_2;
        char Id_3;
        boolean Id_4;
        double Id_5;

        read (Id_1);
        read (Id_2);
        read (Id_3);
        read (Id_4);
        read (Id_5);
    }
}

```

*Ejemplo 34: sentencia read  
Fuente: Elaboración propia.*

#### 7.6.10.5 Sentencia graficar con graphviz

La sentencia recibirá la ruta del lado del servidor de la imagen .png a generar y el contenido .dot que representa la imagen.

Para su traducción a código 3D se utilizará la misma palabra reservada enviando el puntero de la cadena que representa la ruta así como el que representa el contenido dot como parámetros y el intérprete 3D realizará el acceso al sistema de archivos en el servidor..

```
<graph_expression> ::=  
    "graph" "(" "<expression> ", "<expression> ")"
```

*Sintaxis 35: sentencia graph*  
*Fuente: Elaboración propia.*

```
public class Test {  
    public static void main() {  
        graph("/home/Javier/imagen.png", "Contenido dot");  
    }  
}
```

*Ejemplo 35: sentencia graph*  
*Fuente: Elaboración propia.*

#### Consideraciones:

- El primer parámetro es una ruta valida donde se desea escribir el archivo de imagen, esta debe incluir el nombre del mismo.
- Si el contenido dot contiene errores deben mostrarse como errores semánticos.

## 7.7 Expresiones

### 7.7.1 this

La palabra clave `this` se puede usar solo en el cuerpo de un método de instancia o constructor. Si aparece en otro lugar, se produce un error en tiempo de compilación.

Consideraciones:

- Cuando se usa como una expresión, la palabra clave `this` denota un valor que es una referencia al objeto para el que se invocó el método de instancia o al objeto que se está construyendo, lo que permite acceder a los miembros para obtener su valor o incluso modificarlo.
- El tipo de `this` es la clase `C` dentro de la cual se produce la palabra clave `this`.
- En tiempo de ejecución, la clase del objeto real se hace referencia puede ser la clase `C` o cualquier subclase de `C`.
- La palabra clave `this` también se usa en una declaración especial de invocación de constructor explícita, que puede aparecer al principio del cuerpo de un constructor.

```
class IntVector {
    int v[];
    boolean equals(IntVector other) {
        if (this == other)
            return true;
        if (v.length != other.v.length)
            return false;
        for (int i = 0; i < v.length; i++) {
            if (v[i] != other.v[i]){
                return false;
            }
        }
        return true;
    }
}
```

*Ejemplo 36: this.  
Fuente: Elaboración propia*

### 7.7.2 Signos de agrupación

Para dar orden y jerarquía a ciertas operaciones se utilizarán los signos de agrupación. Para los signos de agrupación se utilizarán los paréntesis. Para la precedencia de operadores consultar Apéndice A (§12).

```
<expression> ::= "("<expression>")"
```

*Sintaxis 36: signos de agrupación  
Fuente: Elaboración propia.*

```
(85+3) / 8
```

*Ejemplo 37: signos de agrupación.  
Fuente: Elaboración propia*

### 7.7.3 Expresiones de creación de instancias de clase

Para crear objetos se deberá de instanciar una clase previamente creada. Para llamar clases de otro documento se necesitará hacer referenciado a dicho archivo, ya sea a través de la sentencia importar o llamar. Para llamar al constructor de un objeto será necesario colocar después de la asignación, la palabra reservada “nuevo” seguido del nombre de la clase.

```
<instance_creation_expression> ::=  
    "new" id "(" <argument_list> ")"  
  
<argument_list> ::= <argument_list> "," <expression>  
                  | <expression>
```

*Sintaxis 37: expresión de instancia de clase  
Fuente: Elaboración propia.*

```
public class Test {  
    public static void main() {  
        Estudiante est1 = new Estudiante();  
    }  
}
```

*Ejemplo 38: expresión de instancia de clase.  
Fuente: Elaboración propia*

### 7.7.4 Expresiones de creación de arreglos

Para crear nuevos arreglos se define la siguiente sintaxis:

```
<instance_creation_expression> ::=  
    "new" <type> <dims_expression>  
  
<dims_expression> ::= <dims_expression> "[" <expression> "]"  
                    | "[" <expression> "]"
```

*Sintaxis 38: expresión de creación de arreglos  
Fuente: Elaboración propia.*

```
public class Test {  
    public static void main() {  
        Estudiante est1 [] = new Estudiante[10];  
    }  
}
```

*Ejemplo 39: expresión de creación de arreglos.  
Fuente: Elaboración propia*

### 7.7.5 Expresiones de acceso a un campo

Se define la forma en la que se debe acceder a campos de un objeto o arreglo para obtener el valor o bien modificarlo.

```

<field_access> ::= "super" "." <expression>
                | id "." "super" "." <expression>
                | id "." <expression>

```

*Sintaxis 39: expresión de acceso a un campo  
Fuente: Elaboración propia.*

```

public class Test {
    public static void main() {
        Estudiante est1 [] = new Estudiante(201513630);
        print (est1.carnet);
    }
}
> 201513630

```

*Ejemplo 40: expresión de acceso a un campo.  
Fuente: Elaboración propia*

Consideraciones:

- Si un acceso retorna un objeto podrá realizarse un acceso sobre el mismo, es decir, será posible seguir realizando accesos.
- El super.id de se refiere al campo denominado "id" del objeto actual, pero con el objeto actual visto como una instancia de la superclase de la clase actual.

#### 7.7.5.1 Acceso a campos estáticos de una clase

El acceso estático se da sin necesidad de instanciar una clase únicamente colocando el identificador de la clase, seguido de punto (".") y seguido del id del campo.

#### 7.7.6 Expresión de invocación a método

Una expresión de invocación de método se utiliza para invocar un método de clase o instancia.

```

<method_invocation> ::= "super" "." id "(" <argument_list> ")"
                    | id "." "super" "." id "(" <argument_list> ")"
                    | id "." id "(" <argument_list> ")"

```

*Sintaxis 40: expresión de invocación de método  
Fuente: Elaboración propia.*

```

public class Test {
    public static void main() {
        Estudiante est1 [] = new Estudiante(201513630);
        print (est1.getCatedratico().getCarnet());
    }
}
> 199614536

```

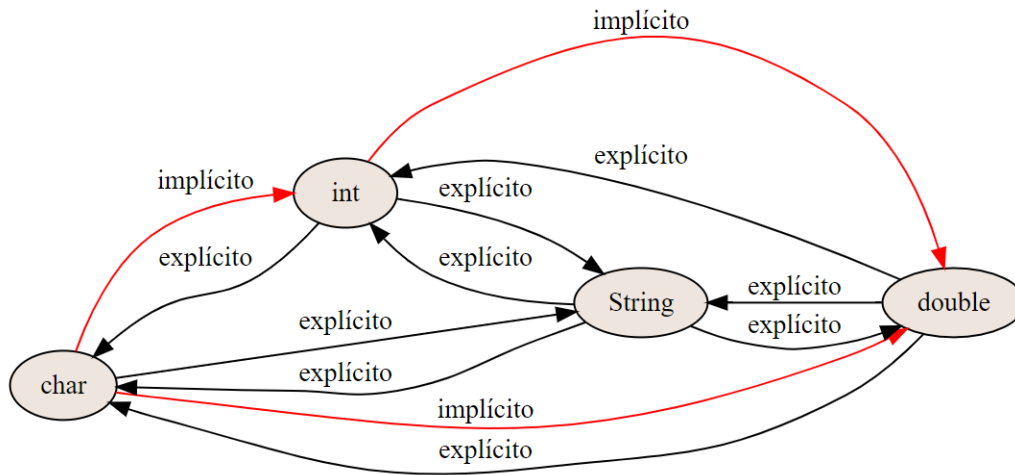
*Ejemplo 41: expresión de creación de arreglos.  
Fuente: Elaboración propia*

#### 7.7.7 Casteos

El casteo es una conversión de tipos en Coline, solo tipos primitivos. Para la precedencia de operadores consultar Apéndice A (§12).

### 7.7.7.1 Casteos permitidos en Coline

Figura 18: casteos permitidos  
Fuente: Elaboración propia.



### 7.7.7.2 Casteo implícito

Este tipo de casteo (también llamado conversión ancha) es una conversión entre dos tipos, sean los tipos de datos  $a$  y  $b$  si el rango de  $a$  es mayor que el de  $b$  entonces el dato puede pasar del tipo  $b$  al tipo  $a$  sin una expresión explícita que represente dicho proceso.

```
// int -> double
double decimal_de_entero = 45;
println(decimal_de_entero);
> 45.0
// char -> int
int entero_de_caracter = 'a';
println(entero_de_caracter);
> 97
double decimal_de_caracter = 'b';
println(decimal_de_caracter);
> 98.0
```

Ejemplo 42: casteos implícitos  
Fuente: Elaboración propia.

### 7.7.7.3 Casteo explícito

Este tipo de casteo (también llamado conversión estrecha) es una conversión entre dos tipos, sean los tipos de datos  $a$  (tipo de expresión) y  $b$  (tipo destino) si el rango de  $a$  es mayor que el de  $b$  entonces para que un dato de tipo  $a$  pueda convertirse en un dato de tipo  $b$ , es necesario.

```
<cast_expression> ::= "("<type_name>") "<expression>
```

Sintaxis 41: casteo explícito.  
Fuente: Elaboración propia.



```

double decimal = 97.37;
/**
 * double -> int (la parte entera del decimal
 * debe estar en el rango de los enteros)
 */
int entero = (int)decimal;
println(entero);
> 97
/**
 * double -> char (la parte entera del decimal
 * debe estar en el rango ASCII valido de los caracteres)
 */
char character = (char) decimal;

println(character);
> a
/**
 * int -> char (el entero debe estar en el
 * rango ASCII valido de los caracteres)
 */
entero++;
character = (char) entero;
println(character);
> b

```

*Ejemplo 43: casteo explicito  
Fuente: Elaboración propia.*

#### Consideraciones:

- Aplica para los tipos char, int y double.
- Si el valor de tipo *a* no puede ser contenido en el rango del tipo *b* debe mostrarse un error en tiempo de ejecución.
- Para castear una expresión de tipo char a su equivalente en tipo int o double se utilizará el numero ASCII que lo represente.

#### 7.7.7.3.1 Casteo explicito con cadenas

```

<cast_expression> ::= "str" "(" <expression> ")"
                    | "toDouble" "("string_terminal ")"
                    | "toInt" "("string_terminal ")"
                    | "toChar" "("string_terminal ")"

```

*Sintaxis 42: casteo explicito con cadenas  
Fuente: Elaboración propia.*

```

int entero = 45;
double decimal = entero;
// Casteo implícito int -> double
println("decimal: " + decimal);
> decimal: 45.0
/**
 * Se obtiene el equivalente en cadena
 * de decimal aumentado en uno
 */
String cadenita = str(++decimal);
println("cadenita: " + cadenita);
> cadenita: 46.0
// Se modifica el valor de decimal
decimal = decimal + 'a';
println("decimal: " + decimal);
> decimal: 143.0
/**
 * Se obtiene el equivalente en decimal
 * de cadenita
 */
decimal = toDouble(cadenita);
println("decimal: " + decimal);
> decimal: 46.0
/**
 * Se obtiene una cadena compuesta por
 * la parte entera de decimal
 */
cadenita = str((int) decimal);
println("cadenita: " + cadenita);
> cadenita: 46
/**
 * Se obtiene el equivalente en entero
 * de cadenita
 */
entero = toInt(cadenita);
println("entero: " + entero);
> entero: 46
/**
 * Se obtiene una cadena compuesta
 * por un caracter
 */
cadenita = str('a');
/**
 * Se obtiene el equivalente en
 * caracter de cadenita
 */

```

```

char character = toChar(cadenita);
println("character: " + character);
> character: a
/**
 * Esto no es posible y se debe
 * arrojar una excepción y detener la ejecución
 */
int a = toInt("1.5");
> Number Format Exception en línea 69, se intentó
convertir a entero una expresión

```

*Ejemplo 44: casteo explícito con cadenas  
Fuente: Elaboración propia.*

#### Consideraciones:

- La primera producción define como debe ser la conversión de cualquier expresión al tipo String, encontrando su equivalente en cadena.
- La segunda producción define como debe ser la conversión de tipo String al tipo double, es posible que no pueda encontrarse un equivalente en double de la cadena, en dicho caso deberá reportarse una excepción.
- La tercera producción define como debe ser la conversión de tipo String al tipo int, es posible que no pueda encontrarse un equivalente en int de la cadena, en dicho caso deberá reportarse una excepción.
- La cuarta producción define como debe ser la conversión de tipo String al tipo char, es posible que no pueda encontrarse un equivalente en char de la cadena, en dicho caso deberá reportarse una excepción.
- Si el valor de tipo  $a$  no puede ser contenido en el rango del tipo  $b$  debe mostrarse un error en tiempo de ejecución.

#### 7.7.8 Operadores aritméticos

Una operación aritmética es un conjunto de reglas que permiten obtener otras cantidades o expresiones a partir de datos específicos. Cuando el resultado de una operación aritmética sobrepase el rango establecido para los tipos de datos deberá mostrarse un error en tiempo de ejecución. Para la precedencia de operadores consultar Apéndice A (§12).

A continuación, se definen las operaciones aritméticas soportadas por el lenguaje.

### 7.7.8.1 Suma

Operación aritmética que consiste en reunir varias cantidades (sumandos) en una sola (la suma). El operador de la suma es el signo más (+). En la Tabla 5 se encuentran las diferentes posibilidades de una suma.

Tabla 5: sistema de tipos para la suma.  
Fuente: Elaboración propia.

Operandos	Tipo resultante	Ejemplos
int + double double + int double + char char + double double + double	double	5 + 4.5 = 9.5 7.8 + 3 = 10.8 15.3 + 'a' = 112.3 'b' + 2.7 = 100.7 3.56 + 2.3 = 5.86
int + char char + int int + int char + char	int	7 + 'c' = 106 'c' + 7 = 74 4 + 5 = 9 'b' + 'f' = 200
String + int int + String String + char char + String String + double double + String String + boolean boolean + String String + String	String	"1" + 2 = "12" 1 + "2" = "12" "1" + '2' = "12" '1' + "2" = "12" "1" + 2.0 = "12.0" 1.0 + "2" = "1.02" "es " + true = "es true" false + " es" = "false es" "es " + "true" = "es true"

Consideraciones:

- Al sumar un dato numérico con un dato de tipo caracter el resultado será la suma del código ASCII del carácter y el número.
- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.
- El operador de suma (+) esta sobrecargado, ya que cuando el tipo de alguno de los dos operandos sea de tipo String, se realizará una concatenación.

### 7.7.8.2 Resta

Operación aritmética que consiste en quitar una cantidad (sustraendo) de otra (minuendo) para averiguar la diferencia entre las dos. El operador de la resta es el signo menos (-). En la Tabla 6 se encuentran las diferentes posibilidades de una resta.

Tabla 6: sistema de tipos para la resta  
Fuente: Elaboración propia.

Operandos	Tipo resultante	Ejemplos
int - double double - int double - char char - double double - double	double	5 - 4.5 = 0.5 7.8 - 3 = 4.8 197.0 - 'a' = 100.0 'b' - 88.0 = 10.0 3.56 - 2.36 = 1.2
int - char char - int int - int char - char	int	7 - 'a' = -90 'a' - 7 = 90 4 - 5 = -1 'f' - 'b' = 4

Consideraciones:

- Al restar un dato numérico con un dato de tipo carácter el resultado será la resta del código ASCII del carácter y el número.
- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.

### 7.7.8.3 Multiplicación

Operación aritmética que consiste en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). El operador de la multiplicación es el asterisco (\*). En la Tabla 7 se encuentran las diferentes posibilidades de una multiplicación.

Tabla 7: sistema de tipos para la multiplicación  
Fuente: Elaboración propia.

Operandos	Tipo resultante	Ejemplos
int * double double * int double * char char * double double * double	double	1 * 4.5 = 4.5 5.1 * 3 = 15.3 1.0 * 'a' = 97.0 'b' * 2.0 = 196.0 3.0 * 2.5 = 7.5
int * char char * int int * int char * char	int	2 * 'd' = 200 'd' * -3 = -300 4 * 5 = -1 'f' * 'b' = 9996

Consideraciones:

- Al multiplicar un dato numérico con un dato de tipo carácter el resultado será la multiplicación del código ASCII del carácter y el número.
- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.

#### 7.7.8.4Potencia

Operación aritmética que consiste en una multiplicación de multiplicandos iguales abreviada. Matemáticamente tiene la siguiente forma  $x^n$ , se multiplica  $n$  (exponente) veces  $x$  (base) y esta se realizará mediante la función pow que recibirá dos parámetros:  $pow(x, n)$

```
<pow_expression> ::= "pow" "(" <expression1> "," <expression2> ")"
```

*Sintaxis 43: operación aritmética de potencia  
Fuente: Elaboración propia.*

En la Tabla 8 se encuentran las diferentes posibilidades de una multiplicación.

*Tabla 8: sistema de tipos para la multiplicación  
Fuente: Elaboración propia.*

Operandos	Tipo resultante	Ejemplos
<code>pow(int, double)</code> <code>pow(double, int)</code> <code>pow(double, char)</code> <code>pow(char, double)</code> <code>pow(double, double)</code> <code>pow(int, char)</code> <code>pow(char, int)</code> <code>pow(int, int)</code> <code>pow(char, char)</code>	<code>double</code>	<code>pow(1, 4.5) = 1.0</code> <code>pow(5.1, 3) = 132.650999</code> <code>pow(1.0, 'a') = 1.0</code> <code>pow('b', 2.0) = 9604.0</code> <code>pow(3.0, 2.5) = 15.588457</code> <code>pow(2, 'd') = 1.26765060E30</code> <code>pow('d', -3) = 1.0E-6</code> <code>pow(4, 5) = 1024.0</code> <code>pow('f', 'b') = 6.96332767E196</code>

Consideraciones:

- El primer parámetro es la base y el segundo es el exponente.
- Al elevar un dato numérico con un dato de tipo caracter el resultado será la potencia del código ASCII del carácter y el número.
- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.

#### 7.7.8.5Modulo

Operación aritmética que consiste en partir un todo en varias partes, al todo se le conoce como dividendo, al total de partes se le llama divisor y el resultado será el residuo de realizar la división entre el dividendo y el divisor. El operador del módulo es el signo de porcentaje (%). En la Tabla 9 se encuentran las diferentes posibilidades un módulo.

*Tabla 9: sistema de tipos para el modulo  
Fuente: Elaboración propia.*

Operandos	Tipo resultante	Ejemplos
<code>int % double</code> <code>double % int</code>	<code>double</code>	<code>10 % 2.5 = 0.0</code> <code>5.0 % 2 = 1.0</code>

double % char char % double double % double		194.0 % 'a' = 0.0 'b' % 2.0 = 0.0 10.0 % 7 = 3.0
int % char char % int int % int char % char	int (se toma solo la parte entera)	194 % 'a' = 0 'b' % 5 = 3 10 % 4 = 2 'z' % 'a' = 25

#### Consideraciones:

- Al dividir un dato numérico con un dato de tipo carácter el resultado será el residuo de la división del código ASCII del carácter y el número.
- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.
- El divisor debe ser diferente de cero, de lo contrario tendrá que reportarse un error semántico.

#### 7.7.8.6 División

Operación aritmética que consiste en partir un todo en varias partes, al todo se le conoce como dividendo, al total de partes se le llama divisor y el resultado recibe el nombre de cociente. El operador de la división es la diagonal (/). En la Tabla 10 se encuentran las diferentes posibilidades de una división.

Tabla 10: sistema de tipos para la división  
Fuente: Elaboración propia.

Operandos	Tipo resultante	Ejemplos
int / double double / int double / char char / double double / double	double	10 / 2.5 = 4.0 5.0 / 2 = 2.5 194.0 / 'a' = 97.0 'b' / 2.0 = 49.0 10.0 / 2.5 = 4.0
int / char char / int int / int char / char	int (se toma solo la parte entera)	194 / 'a' = 2 'b' / 5 = 19 10 / 4 = 2 'z' / 'a' = 1

#### Consideraciones:

- Al dividir un dato numérico con un dato de tipo carácter el resultado será la división del código ASCII del carácter y el número.
- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.
- El divisor debe ser diferente de cero, de lo contrario tendrá que reportarse un error semántico.

### 7.7.9 Operadores prefijos

Para la precedencia de operadores consultar Apéndice A (§12).

#### 7.7.9.1 Menos unario

Operador unario que retorna el inverso multiplicativo de una expresión.

```
<unary_minus> ::= "-" <expression>
```

*Sintaxis 44: operador prefijo menos unario*

*Fuente: Elaboración propia.*

```
char a = 'a';  
int b = -a;  
println(b);  
> -97
```

*Ejemplo 45: operador prefijo menos unario*

*Fuente: Elaboración propia.*

Consideraciones:

- El inverso multiplicativo de un carácter es el inverso multiplicativo de su equivalente ASCII.
- Únicamente aplica para tipos numéricos (int, double, char).
- El tipo resultante será el tipo del operando unario.

#### 7.7.9.2 Incremento

Operador unario que incrementa el valor del operando en uno. El operador de incremento son dos signos más (++).

```
<increment> ::= "++" <expression>
```

*Sintaxis 45: operador prefijo de incremento*

*Fuente: Elaboración propia.*

```
int contador = 10;  
println(++contador);  
println(contador);  
> 11  
> 11
```

*Ejemplo 46: operador prefijo de incremento*

*Fuente: Elaboración propia.*

Consideraciones:

- El operando deberá ser un identificador de una variable previamente definida.
- Este operador realizara dos acciones específicas:
  1. Modificar el valor de la variable almacenada en la tabla de símbolos, aumentado en uno dicho valor.



2. Retornar el nuevo valor de la variable.
- El valor retornado, podrá o no ser utilizado.

### 7.7.9.3 Decremento

Operador unario que disminuye el valor del operando en uno. El operador del decremento son dos signos menos (--).

```
<increment> ::= "--" <expression>
```

*Sintaxis 46: operador prefijo de decremento  
Fuente: Elaboración propia.*

```
int contador = 10;  
println(--contador);  
println(contador);  
> 9  
> 9
```

*Ejemplo 47: operador postfijo de decremento  
Fuente: Elaboración propia.*

Consideraciones:

- El operando deberá ser un identificador de una variable previamente definida.
- Este operador realizara dos acciones específicas:
  1. Modificar el valor de la variable almacenada en la tabla de símbolos, disminuyendo en uno dicho valor.
  2. Retornar el nuevo valor de la variable.
- El valor retornado, podrá o no ser utilizado.

## 7.7.10 Operadores postfijos

Para la precedencia de operadores consultar Apéndice A (§12).

### 7.7.10.1 Incremento

Operador unario que incrementa el valor del operando en uno. El operador del aumento son dos signos más (++).

```
<increment> ::= <expression> "++"
```

*Sintaxis 47: operador postfijo de incremento  
Fuente: Elaboración propia.*

```
int contador = 10;  
println(contador++);  
println(contador);  
> 10  
> 11
```

*Ejemplo 48: operador postfijo de incremento  
Fuente: Elaboración propia.*

Consideraciones:

- El operando deberá ser un identificador de una variable previamente definida.
- Este operador realizara dos acciones específicas:
  1. Modificar el valor de la variable almacenada en la tabla de símbolos, aumentado en uno dicho valor.
  2. Retornar el antiguo valor de la variable, es decir, sin realizar el aumento.
- El valor retornado, podrá o no ser utilizado.

### 7.7.10.2 Decremento

Operador unario que disminuye el valor del operando en uno. El operador del decremento son dos signos menos (--).

```
<increment> ::= <expression> "--"
```

*Sintaxis 48: operador postfijo de decremento  
Fuente: Elaboración propia.*

```
int contador = 10;  
println(contador--);  
println(contador);  
> 10  
> 9
```

*Ejemplo 49: operador postfijo de decremento  
Fuente: Elaboración propia.*

Consideraciones:

- El operando deberá ser un identificador de una variable previamente definida.
- Este operador realizara dos acciones específicas:
  1. Modificar el valor de la variable almacenada en la tabla de símbolos, disminuyendo en uno dicho valor.
  2. Retornar el antiguo valor de la variable, es decir, sin realizar el decremento.
- El valor retornado, podrá o no ser utilizado.

### 7.7.11 Operadores relacionales

Una operación relacional es una operación de comparación entre dos valores, siempre devuelve un valor de tipo lógico (`boolean`) según el resultado de la comparación. Una operación relacional es binaria, es decir tiene dos operandos siempre.

### 7.7.11.1 Operadores de comparación numérica y de igualdad

En la tabla 11 se muestran los ejemplos de los operadores relacionales. Para la precedencia de operadores consultar Apéndice A (§12).

Tabla 11: operadores relacionales  
Fuente: Elaboración propia.

Operador relacional	Operador relacional	Ejemplos de uso
<pre>int [&gt;,&lt;,&gt;=,&lt;=] double double [&gt;,&lt;,&gt;=,&lt;=] int double [&gt;,&lt;,&gt;=,&lt;=] char char [&gt;,&lt;,&gt;=,&lt;=] double int [&gt;,&lt;,&gt;=,&lt;=] char char [&gt;,&lt;,&gt;=,&lt;=] int double [&gt;,&lt;,&gt;=,&lt;=] double int [&gt;,&lt;,&gt;=,&lt;=] int char [&gt;,&lt;,&gt;=,&lt;=] char</pre>	>,<,>=,<=	<pre>5 &gt; 4.5 = true 4.5 &lt; 3 = false 5.5 &gt;= 'a' = false 'a' &lt;= 97.5 = true 5 &gt;= 'a' = false 'a' &lt;= 100 = true 45.2 &gt; 52.2 = false 4 &lt;= 4 = true 'b' &lt; 'a' = false</pre>
<pre>int [==,!=] double double [==,!=] int double [==,!=] char char [==,!=] double int [==,!=] char char [==,!=] int double [==,!=] double int [==,!=] int char [==,!=] char String [==,!=] String boolean [==,!=] boolean</pre>	==,!=	<pre>5 == 4.5 = false 4.5 != 3 = true 5.5 == 'a' = false 'a' == 97.0 = true 5 == 'a' = false 'a' != 100 = true 45.2 == 52.2 = false 4 != 4 = false 'b' == 'a' = false "abc" != "ABC" = true true == false = false</pre>

Consideraciones:

- Será válido comparar datos numéricos (entero, decimal) entre sí, la comparación se realizará utilizando el valor numérico con signo de cada dato.
- Será válido comparar cadenas de caracteres entre sí, la comparación se realizará lexicográficamente, tal como se ordenan las palabras en un diccionario.
- Será válido comparar datos numéricos con datos de carácter en este caso se hará la comparación del valor numérico con signo del primero con el valor del código ASCII del segundo.
- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.

### 7.7.11.2 Operador de comparación de tipos instanceof

Este operador definirá si una variable es instancia de una clase específica retornando true de cumplirse y de lo contrario retornara false.

```
<instanceof_expression> ::= id1 "instanceof" id2;
```

*Sintaxis 49: operador instanceof  
Fuente: Elaboración propia.*

```
// Se verificará si el estudiante1 es instancia de Estudiante
if ( estudiante1 instanceof Estudiante){
    println("estudiante1 es instancia de la clase
    Estudiante");
}else{
    println("estudiante1 no es instancia de la clase
    Estudiante");
}
```

*Ejemplo 50: declaración de arreglos  
Fuente: Elaboración propia.*

Consideraciones:

- El *id1* debe representar una variable, una instancia de clase.
- El *id2* debe representar una clase.
- Si el objeto que es representado por él *id1* es instancia de la clase representada por el segundo id

### 7.7.12 Operaciones lógicas

Las operaciones lógicas son expresiones matemáticas cuyo resultado será valor lógico (boolean). Las operaciones lógicas se basan en el álgebra de Boole. Para la precedencia de operadores consultar Apéndice A (§12).

En la tabla 12 se muestra las tablas de verdad para las operaciones lógicas. Los operadores disponibles son:

- Suma lógica o disyunción, conocida como OR (||)
- Multiplicación lógica o conjunción, conocida como AND (&&)
- Suma exclusiva, conocido como XOR (^)
- Negación, conocida como NOT (!)

*Tabla 12: tabla de verdad para operadores booleanos  
Fuente: Elaboración propia.*

OPERANDO A	OPERANDO B	A    B	A && B	A ^ B	!A
false	false	false	false	false	true
false	true	true	false	true	true
true	false	true	false	true	false
true	true	true	true	false	false

Consideraciones:

- Ambos operadores deberán ser de tipo booleano.

## 7.7.13 Operador ternario

Este operador **retornará** una expresión en función de una condición lógica. Para la precedencia de operadores consultar Apéndice A (§12).

<ternary> ::= <expression1>"?"<expression2>":"<expression3>

*Sintaxis 50: operador ternario*

*Fuente: Elaboración propia.*

Si la condición es verdadera entonces el valor resultante será expression1, de lo contrario si la condición fuera falsa el valor retornado será expression2.

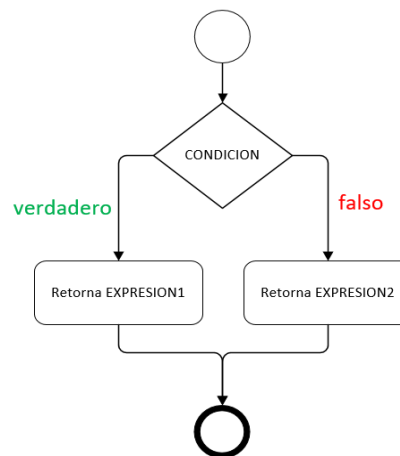
```
int var1 = 0;
println("var1 = " + var1);
> 0
/**
 * La condición no se cumplirá ya que el operador
 * postfijo retornara el valor antiguo de var1 (0)
 */
int var2 = (var1++ == 1 ? 12 : 69) + 1;
/**
 * Posterior a evaluar la condición el operador
 * postfijo modifica el valor de var1 aumentándolo en uno
 */
println("var1 = " + var1);
> 1
/**
 * Como la condición no se cumplió, el operador
 * ternario, retornó 69 luego ese 69 se sumó con el 1
 */
println("var2 = " + var2);
> 70
/**
 * Para notar la diferencia entre pre y postfijo,
 * regresamos el valor de var1 a 0, observemos
 * que no se utiliza el valor retornado
 */
var1--;
println("var1 = " + var1);
> 0
/**
 * Ya el operador prefijo realiza la modificación y retorna el
 * nuevo valor, en esta ocasión la condición si se cumple, es
 * decir, el operador ternario retorna 12 y lo suma con el 1
 */
int var3 = (++var1 == 1 ? 12 : 69) + 1;
```

```
println("var3 = " + var3);  
> 13
```

*Ejemplo 51: operador ternario  
Fuente: Elaboración propia.*

Consideraciones:

- El valor implícito de `expression1` deberá ser de tipo booleano.
- Si `expression1` es verdadera se retornará el valor implícito de `expression2`, de lo contrario se retornará el valor de la `expression3`.



*Figura 19: diagrama de flujo para un operador ternario  
Fuente: Elaboración propia.*

## 8 El formato de código intermedio

El formato estándar de código de tres direcciones es una secuencia de proposiciones que tiene formato general  $x = y \text{ op } z$ , donde  $x$ ,  $y$ ,  $z$  son nombres, constantes o variables temporales que han sido generadas por el compilador, y  $op$  representa a operadores aritméticos o relacionales. Este código deberá generarse de acuerdo al estándar de código intermedio tres direcciones visto en clase.

### 8.1 Definición léxica

#### 8.1.1 Finalización de línea

El intérprete de **C3D** dividirá la entrada en líneas, estas líneas estarán divididas mediante componentes léxicos que determinen la finalización de las mismas y son los siguientes:

- Carácter salto de línea (LF ASCII)
- Carácter retorno de carro (CR ASCII)
- Carácter retorno de carro (CR ASCII) seguido de carácter salto de línea (LF ASCII)

#### 8.1.2 Espacios en blanco

Los espacios en blancos definirán la división entre los componentes léxicos. Serán considerados espacios en blanco los siguientes caracteres:

- Espacio (SP ASCII)
- Tabulación horizontal (HT ASCII)
- Caracteres de finalización de línea

#### 8.1.3 Comentarios

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

Existirán dos tipos de comentarios:

- Los comentarios de una línea que serán delimitados al inicio con los símbolos “//” y al final con un carácter de finalización de línea
- Los comentarios con múltiples líneas que empezarán con los símbolos “/\*” y terminarán con los símbolos “\*/”.

```
//este es un ejemplo de un lenguaje de una sola línea
/*
este es un ejemplo de un lenguaje de múltiples líneas.
*/
```

*Ejemplo 52: ejemplo de comentarios  
Fuente: Elaboración propia.*

#### 8.1.4 Temporales

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

```
t[0-9] +
```

*Ejemplo 53: expresión regular para los temporales  
Fuente: Elaboración propia.*

A continuación, se muestran ejemplos de temporales validos:

```
t1  
t152
```

*Ejemplo 54: temporales válidos  
Fuente: Elaboración propia.*

### 8.1.5 Etiquetas

Las etiquetas serán creadas por el compilador en el proceso de generación de código intermedio. Las etiquetas deberán empezar con la letra “L” seguida de un número.

```
L[0-9] +
```

*Ejemplo 55: expresión regular para los etiquetas  
Fuente: Elaboración propia.*

A continuación, se muestran ejemplos de etiquetas validas:

```
L12  
L25
```

*Ejemplo 56: etiquetas válidos  
Fuente: Elaboración propia.*

### 8.1.6 Identificadores

Un identificador será utilizado para dar un nombre a variables, métodos o estructuras.

Un identificador es una secuencia de caracteres alfabéticos **[A-Z a-z]** incluyendo el guion bajo **[\_]** o dígitos **[0-9]** que comienzan con un carácter alfabético o guion bajo.

A continuación, se muestran ejemplos de identificadores validos:

```
este_es_un_identificador_valido_09  
_este_tambien_09  
Y_este_2018
```

*Ejemplo 57: identificadores validos  
Fuente: Elaboración propia.*

A continuación, se muestran ejemplos de identificadores no validos:

```
0id  
id-5
```

*Ejemplo 58: identificadores no válidos  
Fuente: Elaboración propia.*



### 8.1.7 Palabras reservadas

Son palabras que no podrán ser utilizadas como identificadores ya que representan algo específico y necesario dentro del **C3D**.

Tabla 13: lista de palabras reservadas para el código 3D  
Fuente: Elaboración propia.

var	stack	heap	proc
begin	end	call	println
goto	if	then	ifFalse
			null

### 8.1.8 Símbolos

Tabla 14: lista de símbolos del código 3D  
Fuente: Elaboración propia.

+	suma	-	resta negativo	*	multiplicación	/	división
%	modulo	;	punto y coma	,	coma	=	igual
!=	diferente que	==	igual que	>=	mayor o igual que	>	mayor que
<=	menor o igual que	<	menor que	[	corchete derecho	]	corchete derecho
(	paréntesis izquierdo	(	paréntesis derecho				

### 8.1.9 Literales

Representan el valor de un tipo primitivo

- Enteros: [0-9]+
- Decimales: [0-9]+(“.” [0-9])?
  - Toda expresión de este tipo utilizará 6 decimales con poda o corte.
  - Si se desea imprimir una expresión de tipo decimal y la parte entera cuenta con 8 dígitos o más, deberá imprimirse en notación científica.

## 8.2 Definición de sintaxis

### 8.2.1 Declaración de variables

Se podrán declarar variables de la siguiente forma

```
<var_declaration> ::= "var" <id_list> ";"  
                    | "var" "stack" "[" "]" ";"  
                    | "var" "heap" "[" "]" ";"
```

*Sintaxis 51: declaración de variables C3D  
Fuente: Elaboración propia.*

```
var P = 0;  
var H = 0;
```

*Ejemplo 59: declaración de variables C3D  
Fuente: Elaboración propia.*

### 8.2.2 Operadores aritméticos

Las operaciones aritméticas serán las operaciones básicas que aparecen en la tabla 15.

*Tabla 15: operaciones aritméticas permitidas en el código 3D  
Fuente: Elaboración propia.*

OPERADOR	OPERACIÓN
+	suma
-	resta
*	multiplicación
%	modulo
/	división

### 8.2.3 Operadores relacionales

Las operaciones relacionales serán las operaciones básicas que aparecen en la tabla 16.

*Tabla 16: operaciones relacionales permitidas en el código 3D  
Fuente: Elaboración propia.*

OPERADOR	OPERACIÓN
<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que
==	igual que
!=	diferente que

## 8.2.4 Operadores lógicos

Los operadores lógicos &&, || y ^ serán implementados por ifFalse, como se ha visto en el laboratorio.

## 8.2.5 Asignación

```
<assignment> ::= id1 "=" id2 op id3 ";"  
                | id1 "=" id2 ";"  
                | id "=" num_literal ";"
```

*Sintaxis 52: asignación C3D*  
*Fuente: Elaboración propia.*

```
t1 = t1 + 1;  
t2 = t1 + 0;  
t3 = 10 + 5;  
t4 = t2 + t3;
```

*Ejemplo 60: asignación C3D*  
*Fuente: Elaboración propia.*

## 8.2.6 Destino de un salto

El destino de un salto condicional o incondicional será una etiqueta.

```
<destiny_of_jump> ::= label ":"
```

*Sintaxis 53: destino de un salto*  
*Fuente: Elaboración propia.*

```
L1:
```

*Ejemplo 61: destino de un salto*  
*Fuente: Elaboración propia.*

## 8.2.7 Salto incondicional

Sentencia que indicará un salto a la etiqueta que se especifique.

```
<inconditional_jump> ::= "goto" label ";"
```

*Sintaxis 54: salto incondicional*  
*Fuente: Elaboración propia.*

```
L1:  
    t1 = t1 + 1;  
    t2 = t1 + 0;  
    t3 = 10 + 5;  
    t4 = t2 + t3;  
    goto L2;  
    t5 = t5 + t4;  
    t6 = t5;  
L2:
```

*Ejemplo 62: salto incondicional*  
*Fuente: Elaboración propia.*

## 8.2.8 Salto condicional

### 8.2.8.1 If then

Sentencia que indicará un salto a la etiqueta que se especifique si se cumple la condición.

```
<conditional_jump_if> ::= "if" "(" <rel_op> ")" "goto" label ";"
```

*Sintaxis 55: salto condicional if  
Fuente: Elaboración propia.*

```
L2:
    t1 = t1 + 1;
    t2 = t1 + 0;
    t3 = 10 + 5;
    if (t2 < t3) goto L2;
    t5 = t5 + t4;
    t6 = t5;
L2:
```

*Ejemplo 63: salto condicional if  
Fuente: Elaboración propia.*

### 8.2.8.2 If False

Sentencia que indicará un salto a la etiqueta que se especifique si no se cumple la condición.

```
<conditional_jump_ifFalse> ::=
    "ifFalse" "(" <rel_op> ")" "goto" label ";"
```

*Sintaxis 56: salto condicional ifFalse  
Fuente: Elaboración propia.*

```
L2:
    t1 = t1 + 1;
    t2 = t1 + 0;
    t3 = 10 + 5;
    ifFalse (t2 < t3) goto L2;
    t5 = t5 + t4;
    t6 = t5;
L2:
```

*Ejemplo 64: salto condicional ifFalse  
Fuente: Elaboración propia.*

## 8.2.9 Declaración de métodos

Los métodos podrán ser declarados de la siguiente forma:

```
<method_declaration> ::= "proc" id "begin" <3D_statement_block> "end"
```

*Sintaxis 57: declaración de métodos  
Fuente: Elaboración propia.*

```

proc metodo begin
  L2:
    t1 = t1 + 1;
    t2 = t1 + 0;
    t3 = 10 + 5;
    ifFalse (t2 < t3) goto L2;
    t5 = t5 + t4;
    t6 = t5;
  L2:
end

```

*Ejemplo 65: declaración de métodos*  
*Fuente: Elaboración propia.*

## 8.2.10 Invocación a métodos

Se podrá invocar métodos de la siguiente forma:

```
<method_invocation> ::= "call" id ";"
```

*Sintaxis 58: invocación de métodos*  
*Fuente: Elaboración propia.*

```
call metodo;
```

*Ejemplo 66: invocación de métodos*  
*Fuente: Elaboración propia.*

## 8.2.11 Sentencia print

Esta sentencia recibirá un parámetro de tipo cadena y un identificador que tendrá el valor de lo que se mostrará en la consola de salida. En la tabla 17 se muestran los parámetros permitidos para esta sentencia.

```

<print_statement> ::=
  "print" "(" " " " " " %" char_terminal " " " " " ," id ")" " ";"

```

*Sintaxis 59: sentencia print*  
*Fuente: Elaboración propia.*

*Tabla 17: parámetros permitidos para sentencia print C3D*  
*Fuente: Elaboración propia.*

PARAMETRO	ACCION
"%c"	Imprime el valor en formato de carácter del id
"%e"	Imprime el valor en formato de entero del id
"%d"	Imprime el valor en formato de decimal del id

```

t1 = 4;
print("%d" , t1);
> 4.0

```

*Ejemplo 67: sentencia print*

## 9 Entorno de ejecución

Como ya se expuso en otras secciones de este documento, el proceso de compilación genera el código intermedio y este es el único que se va a ejecutar, siendo indispensable para el flujo de la aplicación, 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.

### 9.1 Estructuras del entorno de ejecución

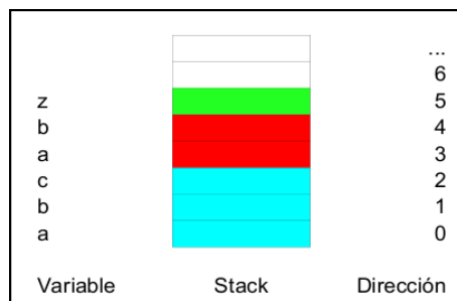
Las estructuras del entorno de ejecución no son más que arreglos de bytes que emplean ingeniosos mecanismos para emular las sentencias de alto nivel. Este proyecto consistirá de dos estructuras indispensables para el manejo de la ejecución, siendo estas, Stack y Heap, las cuales al ser empleadas en el código de tres direcciones determinará el flujo y la memoria de la ejecución, estas 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.

#### 9.1.1 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 de **Coline** cuenta con un espacio propio en memoria para comunicarse con otros métodos y administrar sus variables locales. 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.

Figura 20: Representación gráfica del Stack  
Fuente: Elaboración propia.



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 una sentencia nativa.

```
<clean_scope> ::=
    "$$_clean_scope" "(" int_literal "," int_literal ")"
```

Sintaxis 60: sentencia clean scope  
Fuente: Elaboración propia.

Cuando se genere código de tres direcciones para **Coline** se deberá 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.

```
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();
. . .
P = P - K;

$$_clean_scope(tx, K); /* Llamada a SGC para limpiar el
ámbito terminado */
```

Ejemplo 68: sentencia clean scope

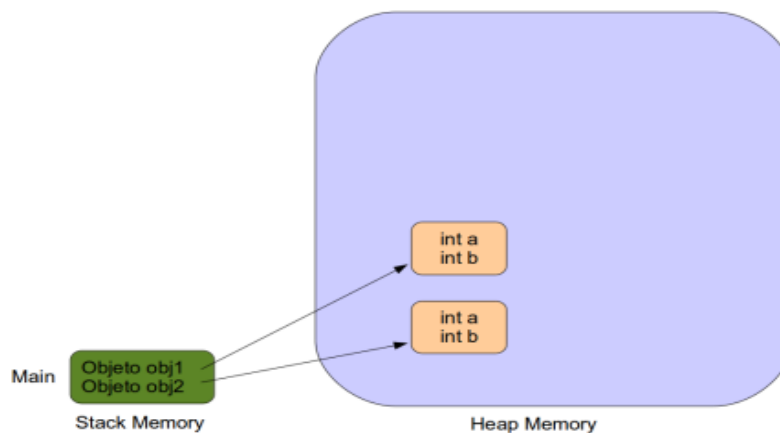
### 9.1.2 El Heap y su puntero

En **Coline** el Heap (o en español, montículo) es la estructura de control del entorno de ejecución encargada de guardar los objetos o referencias a variables globales.

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 Objetos.

Esta estructura también será la encargada de almacenar las cadenas de texto, guardando únicamente en cada posición el ASCII de cada uno de los caracteres que componen la cadena a guardar.

Figura 21: Representación gráfica del Heap  
Fuente: Elaboración propia.



### 9.1.3 Acceso a estructuras del entorno de ejecución

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

```
Stack[id_o_valor] = id_o_valor;  
Heap[id_o_valor] = id_o_valor;
```

Sintaxis 61: acceso a estructuras de ejecución (set)  
Fuente: Elaboración propia.

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



```
id = Stack[id_o_valor];  
id = Heap[id_o_valor];
```

*Sintaxis 62: acceso a estructuras de ejecución (get)*  
*Fuente: Elaboración propia.*

## 10 Optimización de código intermedio

**CAAS** deberá optimizar el código de tres direcciones como parte de la compilación para su posterior ejecución. Para el proceso de optimización se utilizará el método conocido como mirilla (Peephole).

El método de mirilla consiste en utilizar una ventana que se mueve a través del código de 3 direcciones, la cual se le conoce como mirilla, en donde se toman las instrucciones dentro de la mirilla y se sustituyen en una secuencia equivalente que sea de menor longitud y lo más rápido posible que el bloque original. El proceso de mirilla permite que por cada optimización realizada con este método se puedan obtener mejores beneficios.

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

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

Cada uno de estos tipos de transformación utilizados para la optimización por mirilla, tendrán asociadas reglas las cuales en total son 18, estas se detallan a continuación:

### 10.1 Eliminación de instrucciones redundantes de carga y almacenamiento.

#### 10.1.1 Regla 1

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

*Tabla 18: regla 1 de optimización*  
*Fuente: Elaboración propia.*

Ejemplo	Optimización
$t2 = b;$ $b = t2;$	$b = t2$

### 10.2 Eliminación de código inalcanzable

Consistirá en eliminar las instrucciones que nunca serán utilizadas. Por ejemplo, instrucciones que estén luego de un salto condicional, el cual direcciona el flujo de

ejecución a otra parte y nunca llegue a ejecutar las instrucciones posteriores al salto condicional. Las reglas aplicables son las siguientes:

### 10.2.1 Regla 2

Si existe un salto condicional de la forma Lx y exista una etiqueta Lx:, todo código contenido entre el goto Lx y la etiqueta Lx, podrá ser eliminado siempre y cuando no exista una etiqueta en dicho código.

*Tabla 19: regla 2 de optimización  
Fuente: Elaboración propia.*

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

### 10.2.2 Regla 3

Si existe un salto condicional de la forma if <cond> goto Lv; goto Lf; inmediatamente después de sus etiquetas Lv: <instrucciones> Lf: se podrá reducir el número de saltos negando la condición, cambiando el salto condicional hacia la etiqueta falsa Lf: y eliminando el salto condicional innecesario a goto Lf y quitando la etiqueta Lv:.

*Tabla 20: regla 3 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
if a == 10 goto L1; goto L2; L1: <instrucciones> L2:	if a != 10 goto L2; <instrucciones> L2:

### 10.2.3 Regla 4

Si se utilizan valores constantes dentro de las condiciones de la forma if <cond> goto Lv; goto Lf; y el resultado de la condición es una constante verdadera, se podrá transformar en un salto incondicional y eliminarse el salto hacia la etiqueta falsa Lf.

*Tabla 21: regla 4 de optimización  
Fuente: Elaboración propia.*

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

### 10.2.4 Regla 5

Si se utilizan valores constantes dentro de las condiciones de la forma `if <cond> goto Lv; goto Lf;` y el resultado de la condición es una constante falsa, se podrá transformar en un salto incondicional y eliminarse el salto hacia la etiqueta verdadera Lv.

*Tabla 22: regla 5 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
<code>if 1 == 0 goto L1; goto L2;</code>	<code>goto L2;</code>

## 10.3 Optimizaciones de Flujo de control

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

### 10.3.1 Regla 6

Si existe un salto incondicional de la forma `goto Lx` donde existe la etiqueta Lx: y la primera instrucción, luego de la etiqueta, es otro salto, de la forma `goto Ly;` se podrá realizar la modificación al primer salto para que sea dirigido hacia la etiqueta Ly: , para omitir el salto condicional hacia Lx.

*Tabla 23: regla 6 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
<code>goto L1; &lt;instrucciones&gt; L1: goto L2;</code>	<code>goto L2; &lt;instrucciones&gt; L1: goto L2;</code>

### 10.3.2 Regla 7

Si existe un salto incondicional de la forma `if <cond> goto Lx;` y existe la etiqueta Lx: y la primera instrucciones luego de la etiqueta es otro salto, de la forma `goto Ly;` se podrá realizar la modificación al primer salto para que sea dirigido hacia la etiqueta Ly: , para omitir el salto condicional hacia Lx.

*Tabla 24: regla 7 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
---------	--------------

<pre>if t9 &gt;= t10 goto L1; &lt;instrucciones&gt; L1: goto L2;</pre>	<pre>if t9 &gt;= t10 goto L2; &lt;instrucciones&gt; L1: goto L2;</pre>
--	--

## 10.4 Simplificación algebraica y reducción por fuerza

La optimización por mirilla podrá utilizar identidades algebraicas para eliminar las instrucciones ineficientes.

### 10.4.1 Regla 8

Eliminación de las instrucciones que tenga la siguiente forma:

*Tabla 25: regla 8 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
$x = x + 0;$	

Debido que no genera ningún cambio dentro del código.

### 10.4.2 Regla 9

Eliminación de las instrucciones que tenga la siguiente forma:

*Tabla 26: regla 9 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
$x = x - 0;$	

Debido que no genera ningún cambio dentro del código.

### 10.4.3 Regla 10

Eliminación de las instrucciones que tenga la siguiente forma:

*Tabla 27: regla 10 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
$x = x * 1;$	

Debido que no genera ningún cambio dentro del código.

### 10.4.4 Regla 11

Eliminación de las instrucciones que tenga la siguiente forma:

*Tabla 28: regla 11 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
$x = x / 1;$	

Debido que no genera ningún cambio dentro del código.

Para las reglas 12, 13, 14, 15 es aplicable la eliminación de instrucciones con operaciones de variable distinta a la variable de asignación y una constante, sin modificación de la variable involucrada en la operación, por lo que la operación se descartará y se convertirá en una asignación.

#### 10.4.5 Regla 12

*Tabla 29: regla 12 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
$x = y + 0;$	$x = y$

#### 10.4.6 Regla 13

*Tabla 30: regla 13 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
$x = y - 0;$	$x = y$

#### 10.4.7 Regla 14

*Tabla 31: regla 14 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
$x = y * 1;$	$x = y$

#### 10.4.8 Regla 15

*Tabla 32: regla 15 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
$x = y / 1;$	$x = y$

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

#### 10.4.9 Regla 16

*Tabla 33: regla 16 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
$x = y * 2$	$x = y + y$

#### 10.4.10 Regla 17

*Tabla 34: regla 17 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
$x = y * 0$	$x = 0$

#### 10.4.11 Regla 18

*Tabla 35: regla 18 de optimización  
Fuente: Elaboración propia.*

Ejemplo	Optimización
$x = 0 / y$	$x = 0$

## 11 Manejo de errores

La herramienta deberá de ser capaz de identificar todo tipo de errores (léxicos, sintácticos y semánticos) al momento de interpretar el lenguaje de entrada (Celine o 3D-CAAS). Estos errores se almacenarán y se mostrarán en la aplicación. Este reporte deberá contener el día y hora de ejecución, seguido de una tabla con la descripción detallada de cada uno de los errores.

Los tipos de errores se deberán de manejar son los siguientes:

- Errores léxicos.
- Errores sintácticos.
- Errores semánticos

Un error léxico es aquel que ocurre cuando se encuentra un carácter que no está permitido como parte de un lexema.

### Ejemplo 1

```
in$t nombre = "Julius";
```

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

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

### Ejemplo 2

```
int clase var = "Julio"
```

En la entrada del ejemplo anterior se encuentra la palabra reservada "clase" cuando aparentemente sería una declaración y esta no hace match con ninguna producción de la gramática.

### Ejemplo 3

```
int numero = 10;
```

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

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

```
...  
int numero1 = numero2 + 10;  
...
```

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

#### Consideraciones

- Si se existiesen varios errores léxicos se podrá continuar con el análisis de la cadena de entrada y los errores detectados se mostrarán al final del análisis.
- Si se encontrara algún error sintáctico en la cadena este debe tratar de recuperarse con un símbolo de recuperación, en caso que no se pueda recuperar mostrara el error y se detendrá el análisis, si se puede recuperar llevara consigo una lista de errores que se mostraran al final del análisis.
- Si se hallará un error de tipo semántico se procederá a parar el proceso de análisis y se reportará el error de inmediato.

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

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

#### Ejemplo

Tipo	Descripción	Fila	Columna
<b>Léxico</b>	El carácter "o" no pertenece al alfabeto del lenguaje	<b>3</b>	<b>1</b>
<b>Sintáctico</b>	Se esperaba ID y se encontró "/"	<b>74</b>	<b>15</b>
<b>Semántico</b>	La variable "nombre" no ha sido declarada	<b>89</b>	<b>12</b>



## 12 Apéndice A: Precedencia y asociatividad de operadores

Para saber el orden jerárquico de las operaciones se define la siguiente precedencia de operadores. La precedencia de los operadores irá de menor a mayor según su aparición en la tabla 12.

A continuación, se presenta la precedencia de operadores lógicos.

*Tabla 36: precedencia y asociatividad de operadores lógicos  
Fuente: Elaboración propia.*

NIVEL	OPERADOR	DESCRIPCION	ASOCIATIVIDAD
13	[]	Acceso a elemento de arreglo	Izquierda
	.	Acceso a miembro de un objeto	
	()	paréntesis	
12	++	incremento postfijo	no asociativo
	--	decremento postfijo	
11	++	incremento prefijo	Derecha
	--	decremento prefijo	
	-	menos unario	
	!	not	
10	(<type>)	casteo explícito	Derecha
	new	creación de objetos	
9	* / %	multiplicativas	Izquierda
8	+ -	aditivas	Izquierda
	+	concatenación de cadenas	
7	< <=	relacionales	no asociativo
	> >=		
	instanceof		
6	==	igualdad	izquierda
	!=		
5	^	Xor	izquierda
4	&&	And	Izquierda
3		Or	Izquierda
2	? :	ternario	Derecha
1	=	asignación	derecha

## 13 Apéndice B: Archivo de prueba

Los archivos de prueba se encontrarán alojados en la siguiente dirección: [Link archivos de prueba](#).

## 14 Apéndice C: Referencias

### 14.1 Jison

Esta es la página oficial de Jison, en esta podrán encontrar toda la documentación oficial, instalación y algunos ejemplos de su uso.

<http://zaa.ch/jison/docs/>

### 14.2 AWS

Página oficial de AWS, con link directo a DynamoDB

<https://aws.amazon.com/es/dynamodb/>

### 14.3 SDK

SDK AWS for javascript es la conexión directa entre javascript (node js) y diferentes servicios de AWS, entre ellos DynamoDB.

[https://docs.aws.amazon.com/es\\_es/amazondynamodb/latest/developerguide/GettingStarted.NodeJs.html](https://docs.aws.amazon.com/es_es/amazondynamodb/latest/developerguide/GettingStarted.NodeJs.html)

<https://aws.amazon.com/es/sdk-for-node-js/>

### 14.4 Tutorial manejo base de datos DynamoDB

En este link encontraran como crear tablas, como cargar datos, consultas y demás ejemplos necesarios para su proyecto.

[https://docs.aws.amazon.com/es\\_es/amazondynamodb/latest/developerguide/GettingStarted.NodeJs.html](https://docs.aws.amazon.com/es_es/amazondynamodb/latest/developerguide/GettingStarted.NodeJs.html)

## 15 Entregables y Restricciones

### 15.1 Entregables

- Código fuente
- Aplicación funcional
- Gramáticas
- Manual técnico
- Manual de usuario

Deberán entregarse todos los archivos necesarios para la ejecución de la aplicación, así como el código fuente, la gramática y la documentación. La calificación del proyecto se realizará con los archivos entregados en la fecha establecida. El usuario es completa y

únicamente responsable de verificar el contenido de los entregables. La calificación se realizará sobre archivos ejecutables. Se proporcionarán archivos de entrada al momento de calificación.

## 15.2 Restricciones

- La aplicación deberá ser desarrollada utilizando el lenguaje javascript haciendo uso de Node Js
- Está permitido el uso de TypeScript.
- Para el almacenamiento de los reportes deberá ser utilizado DynamoDB de AWS.
- Los analizadores léxicos y sintácticos para el compilador deberá ser con Jison.
- El intérprete de la representación intermedia de tres direcciones deberá ser con Jison.
- Copias de proyectos tendrán de manera automática una nota de 0 puntos y serán reportados a la Escuela de Ingeniería en Ciencias y Sistemas los involucrados.
- El lenguaje Coline es case sensitive.

## 15.3 Requisitos mínimos

Los requerimientos mínimos del proyecto son funcionalidades del sistema que permitirán un ciclo de ejecución básica, para tener derecho a calificación se deben cumplir con lo siguiente:

- 3 Componentes y servicios de la aplicación
  - 3.1 Editor en línea
  - 3.3 Intérprete de código de tres direcciones.
- 5 Introducción a lenguaje Coline
  - 5.1 Tipos
    - 5.1.1 Tipos primitivos
    - 5.1.2 Tipos de referencia
      - 5.1.2.1 Clase Object
      - 5.1.2.2 Clase String
    - 5.1.3 Tipo nulo
- 6 Definición léxica
  - 6.1 Finalización de línea
  - 6.2 Espacios en blanco
  - 6.3 Comentarios
  - 6.4 Sensibilidad a mayúsculas y minúsculas
  - 6.5 Identificadores
  - 6.6 Palabras reservadas
  - 6.7 Símbolos
  - 6.8 Literales
- 7 Definición de sintaxis
  - 7.1 Nombres

- 7.1.1 Variables
- 7.1.2 Tipos de variables
  - 7.1.2.2 Variable de instancia
  - 7.1.2.3 Variable sin nombre
  - 7.1.2.4 Parámetros de un método o función
  - 7.1.2.5 Parámetros del constructor
  - 7.1.2.7 Variables locales
- 7.1.4 Valores predeterminados para las variables
- 7.2 Clases
  - 7.2.1 Introducción a las clases en Coline
  - 7.2.2 Declaración de una clase
    - 7.2.2.2 Superclases y subclases
    - 7.2.2.3 Cuerpo de clase y declaraciones de miembro
  - 7.2.3 Miembros de una clase
    - 7.2.3.1 Tipo de un miembro
    - 7.2.3.2 Alcance de un miembro
  - 7.2.4 Declaración de un campo
    - 7.2.4.1 Modificadores de campo
    - 7.2.4.2 Inicialización de campos
  - 7.2.5 Declaración de un método
    - 7.2.5.1 Firma de un método
    - 7.2.5.2 Parámetros formales
    - 7.2.5.3 Modificadores de un método
    - 7.2.5.4 Tipo de retorno
    - 7.2.5.5 Herencia, anular y esconder
  - 7.2.6 Constructor
    - 7.2.6.1 Modificadores para el constructor
    - 7.2.6.2 Invocaciones explícitas de constructores
    - 7.2.6.3 Sobrecarga de constructores
    - 7.2.6.4 Constructor predeterminado
  - 7.2.7 Preparación de una clase
- 7.3 Arreglos
  - 7.3.1 Tipos de arreglos
  - 7.3.2 Declaración de arreglos
  - 7.3.3 Asignación de un valor a posiciones de un arreglo
  - 7.3.4 Inicialización de arreglos
  - 7.3.6 Miembro length
  - 7.3.7 Un arreglo de tipo char no es un String
- 7.6 Sentencias
  - 7.6.1 Bloques
  - 7.6.2 Sentencia import
  - 7.6.3 Declaración de variables
  - 7.6.4 Asignación de variables

- 7.6.5 Sentencias de transferencia
  - 7.6.5.1 Break
  - 7.6.5.3 Return
- 7.6.6 Sentencias de selección
- 7.6.9 Sentencias cíclicas o bucles
- 7.6.10 Sentencias de entrada/salida
  - 7.6.10.1 Sentencia print
- 7.7 Expresiones
  - 7.7.1 this
  - 7.7.2 Signos de agrupación
  - 7.7.3 Expresiones de creación de instancias de clase
  - 7.7.4 Expresiones de creación de arreglos
  - 7.7.5 Expresiones de acceso a un campo
  - 7.7.6 Expresión de invocación a método
  - 7.7.7 Casteos
    - 7.7.7.1 Casteos permitidos en Coline
    - 7.7.7.2 Casteo implícito
    - 7.7.7.3 Casteo explícito
  - 7.7.8 Operadores aritméticos
  - 7.7.9 Operadores prefijos
  - 7.7.10 Operadores postfijos
  - 7.7.11 Operadores relacionales
  - 7.7.12 Operaciones lógicas
  - 7.7.13 Operador ternario
- 8 El formato de código intermedio
- 9 Entorno de ejecución
  - 9.1 Estructuras del entorno de ejecución
    - 9.1.1 Stack y su puntero
    - 9.1.2 Heap y su puntero
- 11 Manejo de errores
- 12 Apéndice A: Precedencia y asociatividad de operadores

## 15.4 Entrega del proyecto

1. La entrega será virtual y por medio de la plataforma Classroom.
2. La entrega de cada uno de los proyectos es individual.
3. Para la entrega del proyecto se deberá cumplir con todos los requerimientos mínimos.
4. No se recibirán proyectos después de la fecha ni hora de la entrega estipulada.
5. La entrega del proyecto será mediante un archivo comprimido de extensión rar o zip.
6. Entrega del proyecto:

**8 de mayo 2019 antes de las 23:59 horas**