Universidad de San Carlos de Guatemala Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2 Escuela de vacaciones de diciembre 2019
Catedrático: Ing. Carlos Olivio Natareno Yanes
Tutor académico: Ricardo Antonio Menchú Barrios.

yaba

Segundo proyecto de laboratorio

Contenido

Objetivos

Objetivo general

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

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 durante la carrera para el desarrollo de un intérprete.
- Que el estudiante sea capaz de implementar los conocimientos adquiridos en el curso tecnologías modernas.

Descripción

Yaba, es un software de compilación en línea, cuya implantación será hecha de forma local por los estudiantes. Yaba 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.

Yaba 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 **OLCEV** 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 **Yaba** 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.
- Reporte de tabla de símbolos: dará un reporte de la tabla de símbolos utilizada durante la compilación.
- Generación de AST: dará un reporte con el AST del lenguaje fuente.
- Reporte de optimización: dará un reporte de la optimización del código de tres direcciones generado.

Todos los anteriormente mencionados serán archivos generados (HTML) por yaba y podrán ser descargados localmente seguido de su generación.

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 **OLCEV**, y el segundo será la ejecución del código de tres direcciones generado a partir del proceso de compilación.

Proceso de compilación

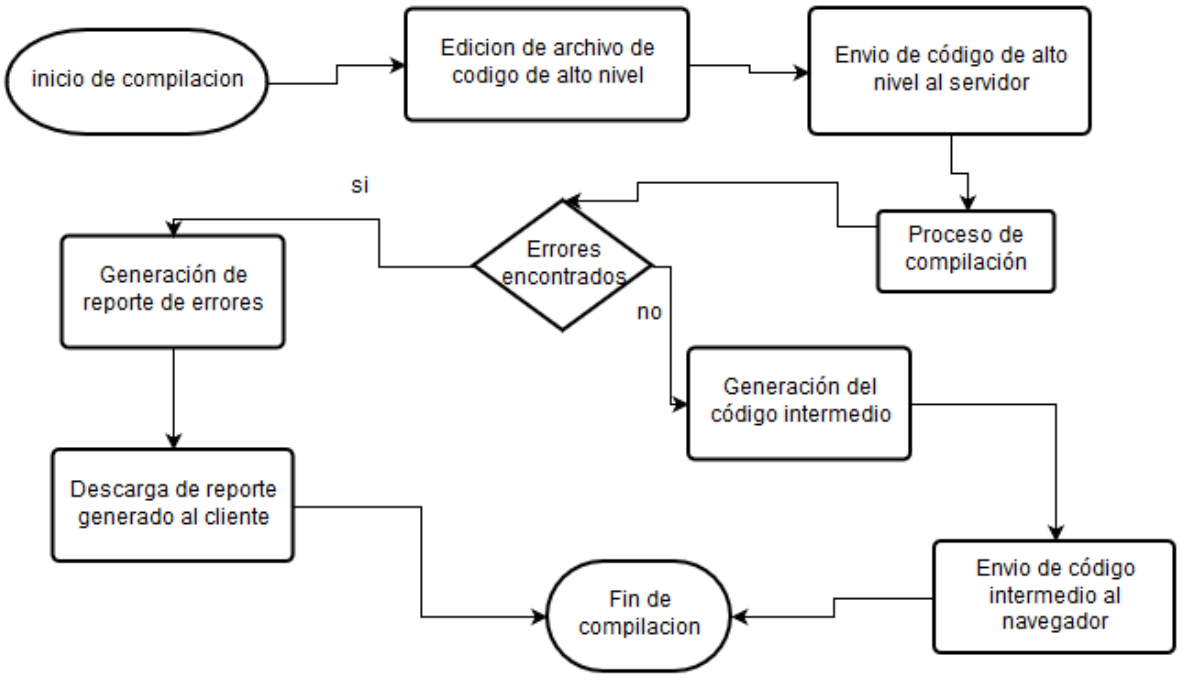
El proceso de compilación tendrá como entrada código de alto nivel y como salida la representación de esté en código de tres direcciones. El proceso de compilación

seguirá los siguientes pasos.

- Flujo normal:
  1. Edición de código de alto nivel **OLCEV** en el editor.
  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.

- Flujo alterno:

  1. Edición de código de alto nivel **OLCEV** 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. Descarga del reporte de errores.

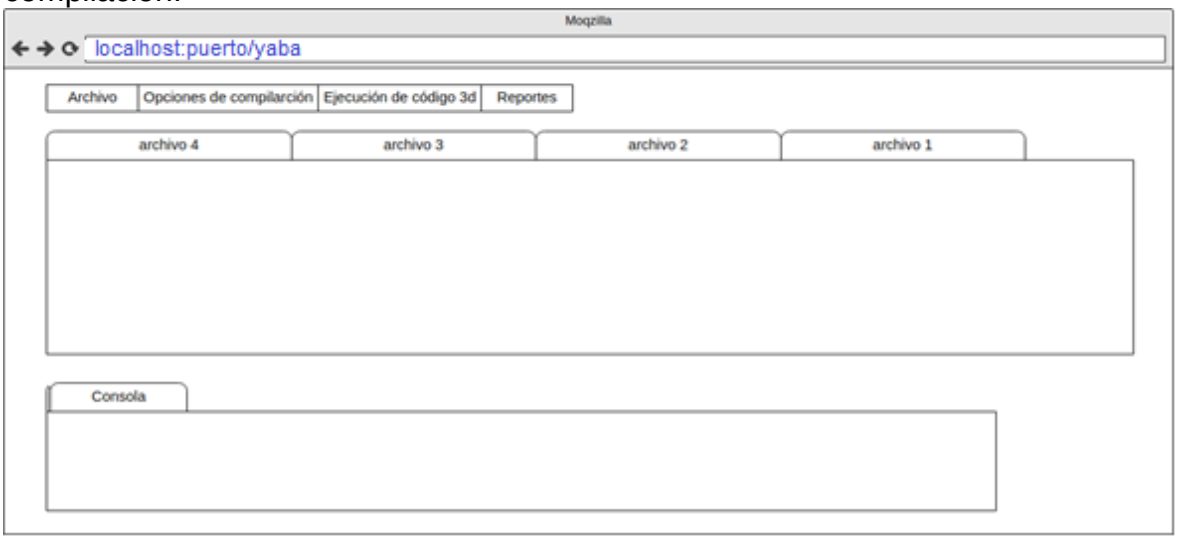


Componentes de la aplicación

A continuación, se describen los componentes de **Yaba**.

Editor en línea

Yaba 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.



El editor en línea estará conformado por las siguientes secciones:

- ### 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.

- ### 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.

- ### 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.

Reportes

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

Reporte de errores de compilación

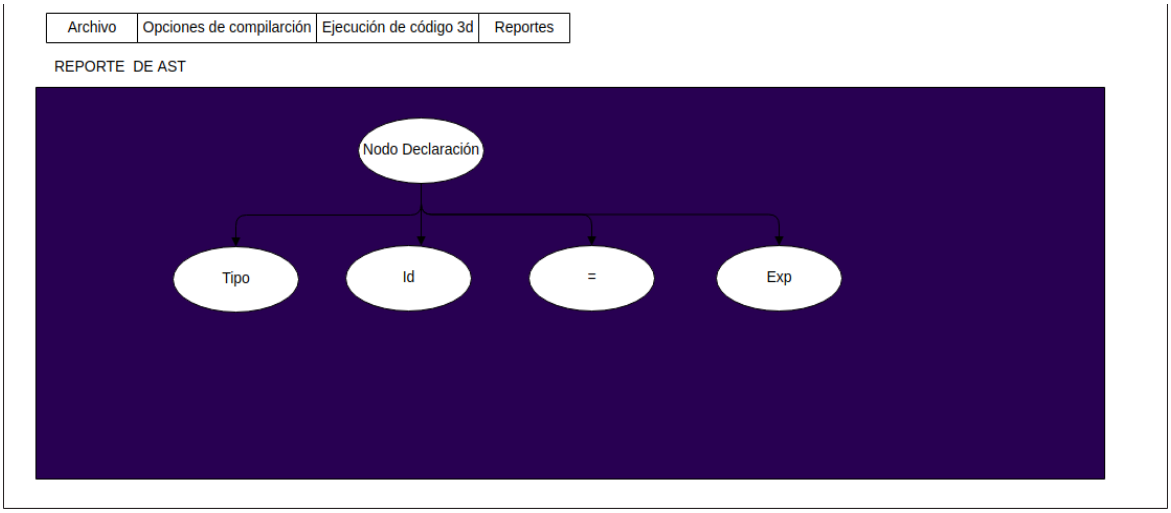
Este reporte mostrará si hubo errores durante el proceso de compilación, mostrando de cada uno, el tipo de error (léxico, sintáctico o semántico), la fila y columna de donde se detectó cada error.

Reporte 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.

Reporte de AST

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



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 optimizara, 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.

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.

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).

- ### 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.

- ### 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.

- ### 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.

- ### Parar depuración

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

- 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.

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.

COLOR	TOKEN
AZUL	Palabras reservadas
NARANJA	Cadenas, caracteres
MORADO	Números
GRIS	Comentario
CAFE	Salida por consola
NEGRO	Otro

Introducción a lenguaje OLCEV

OLCEV 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.

Tipos

OLCEV 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.

- ## 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 punto flotante.
- 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).

TIPO DE DATO	RANGO
`int`	[-2147483648, 2.147.483.647]
`double`	[-9223372036854775800, 9223372036854775800]
`char`	[0,255] (ASCII)
`boolean`	true, false
`String`	[-2147483648, 2.147.483.647] caracteres

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.

- ## 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

○ Clase Object

La clase Object una superclase de todas las demás clases.
Todas las clases y arreglos heredan los métodos de clase Object.

- Miembros
- El método equals() define una noción de igualdad de objeto, que se basa en el valor, no en la comparación, en la comparación.
- El método getClass() devuelve un String que representa el nombre de la clase del objeto.
- El método toString() devuelve una representación String del objeto.

Clase String

Las instancias de clase String representan secuencias de caracteres ASCII.

- Un objeto String tiene un valor constante (invariable).
- Los literales de cadena son referencias a instancias de clase String.
- El operador de concatenación de cadenas (+) crea implícitamente un nuevo objeto String cuando el resultado no es una expresión constante de tiempo de compilación.
  - Miembros
  - El método toCharArray() en 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.

Tipo nulo

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

Paradigma de programación

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

Archivo con extensión .cev

Debido a que **OLCEV** es un lenguaje completamente orientado a objetos, un archivo que contenga código **OLCEV** 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.

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.

Definición léxica

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 "*/".

Ejemplo:

```
//comentario de una sola linea

/*comentario multilinea
otra linea :v
*/
```

Sensibilidad a mayúsculas y minúsculas

OLCEV 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, OLCeV no es igual que olcev.

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:

```
ste_es_un_identificador_valido_12
_este_tambien_12
y_Este_2019
```

.
A continuación, se muestran ejemplos de identificadores no validos:

```
0id
id-5
```

Palabras reservadas

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

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

Símbolos

+	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		

Literales

Representan el valor de un tipo primitivo

- Enteros: [0-9]+
- Decimales: [0-9]+(“.” [WHITESPACE |0-9]+)?
  - Toda expresión de este tipo utilizará 6 decimales con poda o corte.
- 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.

Definición de sintaxis

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.

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.
- Por un operador prefijo o postfijo de incremento o decremento.

Tipos de variables

A continuación, se definen los tipos de variables:

- ### Variable de clase o estática

  Una variable estática es un campo declarado utilizando el modificador static dentro de una declaración de clase. Una variable estática se crea cuando se prepara su clase y se inicializa con un valor predeterminado.

- ### 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 como parte de cada objeto recién creado de la clase T o de cualquier clase que sea una subclase de T.

- ### Variable sin nombre

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

- ### Parámetros de un método o función

Los parámetros de los métodos 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. 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.

- ### Parámetros del constructor

Los parámetros del constructor 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.

- ### Parámetro de excepción

Se crea un parámetro de excepción cada vez que se captura una excepción mediante una cláusula catch de una sentencia try.
La nueva variable se inicializa con el objeto real asociado con la excepción. El parámetro de excepción deja de existir efectivamente cuando se completa la ejecución del bloque asociado con la cláusula catch.

- ### Variables locales

Las variables locales se declaran mediante sentencias de declaración de variables locales.
Cada vez que el flujo de control ingresa a un bloque 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.

- ### 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.

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 y String, el valor predeterminado es *null*.
- Cada parámetro del método o función se inicializa al valor de argumento correspondiente proporcionado por el invocador del método o función.
- Cada parámetro del constructor se inicializa con el valor de argumento correspondiente proporcionado por una expresión de creación de instancia de clase.
- Un parámetro de excepción se inicializa al objeto lanzado que representa la excepción.
- 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.

Clases

Las declaraciones de clase definen nuevos tipos de referencia y describen cómo se implementan.

- ### Introducción a las clases en OLCEV

En OLCEV, 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*, 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*, 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. 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. 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 sobreescribir 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*, en cuyo caso se puede asignar a una sola vez. Cualquier declaración de campo puede incluir un inicializador
Las declaraciones de clase de miembro 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 describen el código que puede invocar las expresiones de invocación de métodos. 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*. Se puede declarar un método *final*, en cuyo caso no se puede ocultar o anular.
Los nombres de los métodos pueden estar sobrecargados.
Los constructores 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.

- ## Declaración de una clase

Una declaración de clase especifica un nuevo tipo de referencia.

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.

    - ### Modificadores de clase

    Una declaración de clase puede incluir modificadores de clase.

        - Public
        - Protected
        - Private
        - Abstract
        - Static
        - Final

    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.*

○ Clases abstractas

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

```java
public abstract class Figura
{
    private String color;

    public Figura(String color)
    {
        this.color = color;
    }

    public abstract double calcularArea();

    public String getColor()
    {
        return color;
    }
}
```

Consideraciones:

- Las clases pueden tener métodos *abstract*, 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*.
    - 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.
- 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.

○ 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* 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.

○ 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");
    }
}
```

Las relaciones en el 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 la clase final ColoredPoint.

○ Cuerpo de clase y declaraciones de miembro

Un cuerpo de clase puede contener declaraciones de miembros de la clase, es decir, campo, métodos, clases. Un cuerpo de clase también puede contener declaraciones de constructores para la clase.

• 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.

Tipo 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.

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 o constructor es el cuerpo completo del método o constructor.
- El alcance de una declaración de clase local inmediatamente encerrada por un bloque 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 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.

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.*

○ 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.
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, se crea una nueva variable asociada con esa instancia para cada variable de instancia declarada en esa clase o cualquiera de sus superclases.

○ Campos finales

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

• Inicialización de campos

Si un declarador de campo contiene un inicializador de variable, entonces tiene la semántica de una asignación 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.
- 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.
- Los inicializadores de variables también se utilizan en las sentencias de declaración de variables locales, donde se evalúa el inicializador y la asignación se realiza cada vez que se ejecuta la declaración de declaración de variables locales.

○ 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* 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.

○ 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.

○ 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.

• Declaración de un método

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

Consideraciones:

- 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, esto produce un error en tiempo de compilación.

○ Firma 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 $A1, \ldots, An$ los tipos de los parámetros M y sean $B, \ldots, Bn$ los tipos de los parámetros de N. Para todo $1 \leq i \leq n$ se cumple que $Ai = Bi$.

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

○ 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.

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.

○ Modificadores de un método

Una declaración de método puede incluir modificadores.

Consideraciones:

- 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*.

○ 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 un método *abstract*
- proporcionando otra declaración *abstract* de método.
- Un método de instancia que no es *abstract* puede ser sobrescrito por un *abstract* método.

○ 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* 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 clave *this* o la palabra clave *super* durante la ejecución del cuerpo del método.

○ 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 sobreescribir u ocultar un final método.
- Un método *private* y todos los métodos declarados inmediatamente dentro de una clase final se comportan como si fueran final, ya que es imposible sobrescribirlos.

○ Tipo 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.

○ Herencia, 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 u ocultados por una declaración en la clase.

Consideraciones:

- Si, por ejemplo, una clase declara dos métodos *public* con el mismo nombre, 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.

○ Sobre escritura de métodos

Un método de instancia $m1$, declarado en la clase C, anula otro método de instancia
$m2$, declarado en la clase A si todos los siguientes son verdaderos:

- C es una subclase de A.
- La firma de $m1$ igual a la de $m2$
- Ya sea:
    - $m2$ es *public*, *protected* o declarado con acceso predeterminado, o
    - $m1$ sobrescribe un método $m3$ ($m3$ distinto de $m1$ y $m2$), tal que $m3$ sobrescribe a $m2$.

Además, si $m1$ no es *abstract*, se dice que $m1$ 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 que contiene la palabra clave super.

○ 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.

- ## 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 sobrescritura, 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, 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.
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.

- ## Constructor

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

Consideraciones:

- Es un error en tiempo de compilación declarar dos constructores con firmas iguales.

  - ### Modificadores para el constructor

  Los modificadores de acceso son public, protected y private.

  Consideraciones:

  - 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*.

  - ### Invocaciones explicitas 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.

Sobrecarga de constructores

Al igual que los métodos, los constructores pueden sobrecargarse.

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.

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. 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.

Arreglo

En OLCEV, los arreglos son objetos y pueden asignarse a variables de tipo *Object*. 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.

- ## Tipos de arreglos

Los tipos de arreglos se utilizan en declaraciones y en expresiones de conversión.
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*.

- ## 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.

```
double arr1[]; // una dimension
double arr2[][]; // dos dimensiones
int matriz [][][]; // tres dimensiones
```

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.

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

Se accede a un componente de un arreglo mediante una expresión de acceso al arreglo 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 $0 \leq i < n$.

```java
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 i = 0; i < ia.length; i++){
            sum = sum + ia[i];
        }
        println(sum);
    }
}
```

- Inicialización de arreglos

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

```java
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}};
```

- 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);
        // La primera "dimensión" es de tamaño = 2
        println("La primera \"dimensión\" es de tamaño = " + arr[0].length);
        // La primera "dimensión" es de tamaño = 3
    }
}
```

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

En el lenguaje de programación OLCEV, 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.

Sentencias

- ### Bloques

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

- ### Sentencia import

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

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.

- ### Declaración de variables

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

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.

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 (=).

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.

Sentencias de transferencia

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

Break

La sentencia break tendrá dos usos:

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

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.

Continue

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

Consideraciones:

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

```
class unaClase {
    public static void main() {
        int a = 0;
        while(a < 10){
            if (a == 5){
                println("a la mitad");
                a++;
                continue;
            }
            else if (a == 8){
                println("en ocho");
                a++;
                break;
            }
            a++;
        }
    }
}
```

- 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.

```
class unaClase {
    public int suma (int a, int b) {
        return a+b;
    }
}
```

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 está contenida únicamente en métodos.

- ## Sentencias de selección

Estas son sentencias que definen la línea de vida que debe tomar el programa según una condición.

La sintaxis y funcionamiento será el mismo utilizado en el lenguaje Java.

  - ### Sentencia If

  - ### switch

    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.

- ## Sentencias cíclicas o bucles

Estas son sentencias que permiten bucles de segmentos de código según una condición.

La sintaxis y funcionamiento será el mismo utilizado en el lenguaje Java.

  - ### While

  - ### Do while

  - ### For

- ## Sentencias de entrada/salida

  - ### Sentencia print

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

    Consideraciones:

    - Si se utiliza la sentencia print se incluirá un salto de línea al final, de lo contrario, si es una sentencia print 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.

Escribir archivo

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

```
public class Test {
    public static void main() {
        write_file("/ruta_absoluta/archivo.txt", "Contenido");
    }
}
```

Consideraciones:

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

Expresiones

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.
- 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.

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 la tabla de precedencia mas adelante

- ### 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.

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

- ### Expresiones de creación de arreglos

Para crear nuevos arreglos se define lo siguiente:

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

- ### Expresiones de acceso a un campo

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.

- ### 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.

- ### Expresiones de invocación a método

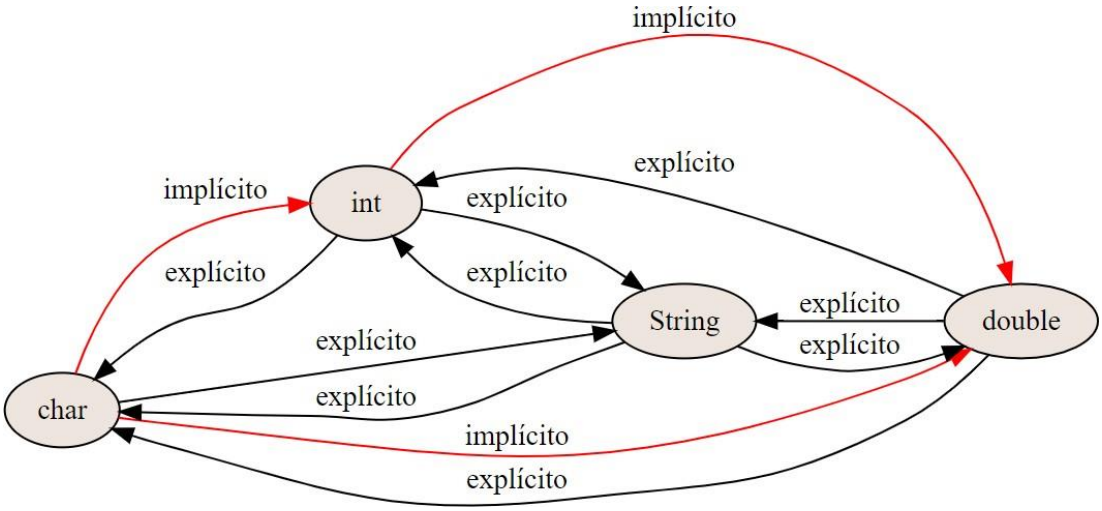
Una expresión de invocación de método se utiliza para invocar un método de clase o instancia.

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

- Casteos

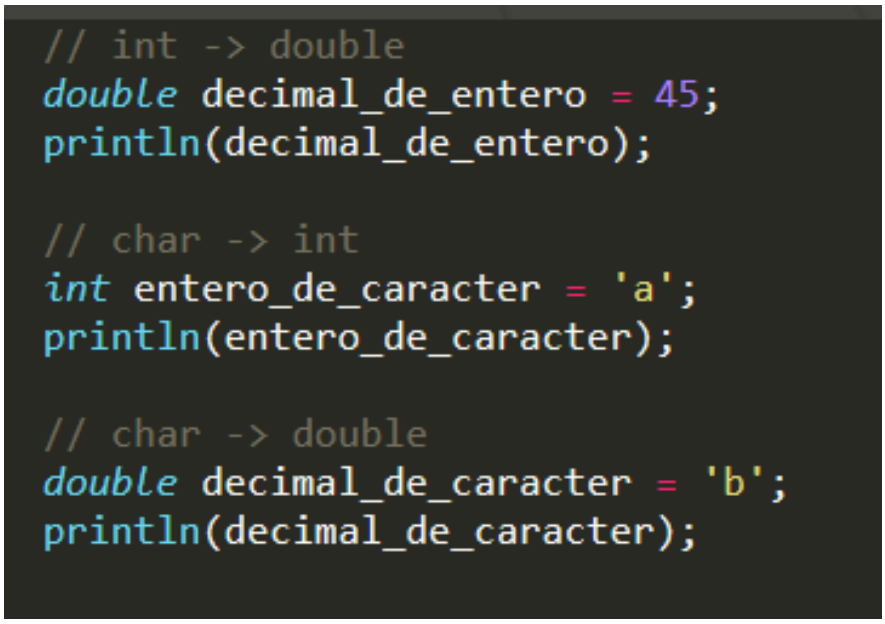
El casteo es una conversión de tipos en OLCEV, solo tipos primitivos.

  - Casteos permitidos en OLCEV



  - Casteos implícito

Este tipo de casteo (también llamado conversión ancha) es una conversación 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 explicita que represente dicho proceso.



```
// int -> double
double decimal_de_entero = 45;
println(decimal_de_entero);

// char -> int
int entero_de_caracter = 'a';
println(entero_de_caracter);

// char -> double
double decimal_de_caracter = 'b';
println(decimal_de_caracter);
```

- Casteos explícito

Este tipo de casteo (también llamado conversión estrecha) es una conversación entre dos tipos, sean los tipos de datos a ($tipo\ de\ expression$) 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.

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.

```
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);

/**
*   double -> char (la parte entera del decimal
*   debe estar en el rango ASCCI valido de los caracteres)
*/
char caracter = (char) decimal;
println(caracter);
/**
*   int -> char (el entero debe estar en el
*   rango ASCCI valido de los caracteres)
*/
entero++;
caracter = (char) entero;
println(caracter);
```

- Casteos explícito con cadenas

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

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.

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

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 se encuentran las diferentes posibilidades de una suma.

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.

○ 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 se encuentran las diferentes posibilidades de una resta.

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 caracter 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.

○ 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 se encuentran las diferentes posibilidades de una multiplicación.

Operandos	Tipo resultante	Ejemplos

`int * double`
`double * int`
`double * char`
`char * double`
	`double`	`1 * 4.5 = 4.5`
		`5.1 * 3 = 15.3`
		`1.0 * 'a' = 97.0`
		`'b' * 2.0 = 196.0`

Consideraciones:

- Al multiplicar un dato numérico con un dato de tipo caracter 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.

○ Potencia

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

Operandos	Tipo resultante	Ejemplos

`pow(int,double)`
`pow(double,int)`
`pow(double,char)`
`pow(char,double)`
`pow(double,double)`
`pow(int,char)`
`pow(char,int)`
`pow(int,int)`
`pow(char,char)`
	`double`	`pow(1,4.5) = 1.0`
		`pow(5.1,3) = 132.650999`
		`pow(1.0,'a') = 1.0`
		`pow('b',2.0) = 9604.0`
		`pow(3.0,2.5) = 15.588457`
		`pow(2,'d') = 1.26765060E30`
		`pow('d',-3) = 1.0E-6`
		`pow(4,5) = 1024.0`
		`pow('f','b') = 6.96332767E196`

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.

○ Modulo

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 se encuentran las diferentes posibilidades un módulo.

Operandos	Tipo resultante	Ejemplos

`int % double`
`double % int`
`double % char`
`char % double`
`double % double`
	`double`	`10 % 2.5 = 0.0`
		`5.0 % 2 = 1.0`
		`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 caracter 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.

○ 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 se encuentran las diferentes posibilidades de una división.

Operandos	Tipo resultante	Ejemplos

int / double double / int double / char char / double	double	10 / 2.5 = 4.0 5.0 / 2 = 2.5 194.0 / 'a' = 97.0 'b' / 2.0 = 49.0

double / double		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 caracter 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.

- Operadores prefijos

  - Menos unario

Operador unario que retorna el inverso multiplicativo de una expresión.

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

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.

  - Incremento

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

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

Consideraciones:

.

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

Decremento

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

```
<increment> ::= “--” <expression>
```

Consideraciones:

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

Operadores postfijos

Incremento

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

```
<increment> ::= <expression> “++”
```

Consideraciones:

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

aumento.
- El valor retornado, podrá o no ser utilizado.

○ Decremento

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

```
<increment> ::= <expression> “--”
```

Consideraciones:

- El operando deberá ser un identificador de una variable previamente definida.
- Este operador realizara dos acciones específicas:

- Modificar el valor de la variable almacenada en la tabla de símbolos, disminuyendo en uno dicho valor.
- Retornar el antiguo valor de la variable, es decir, sin realizar el decremento.
- El valor retornado, podrá o no ser utilizado.

• 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.

○ Operadores de comparación numérica y de igualdad

En la tabla se muestran los ejemplos de los operadores relacionales.

Operador relacional Operador relacional
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 caracter 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.

Operador relacional	Operador relacional	Ejemplos de uso
int [>,<,>=,<=] double double [>,<,>=,<=] int double [>,<,>=,<=] char char [>,<,>=,<=] double int [>,<,>=,<=] char char [>,<,>=,<=] int double [>,<,>=,<=] double int [>,<,>=,<=]int char [>,<,>=,<=]char	>,<,>=,<=	5 > 4.5 = true 4.5 < 3 = false 5.5 >= 'a' = false 'a' <= 97.5 = true 5 >= 'a' = false 'a' <= 100 = true 45.2 > 52.2 = false 4 <= 4 = true 'b' < 'a' = false
int [==,!=] double double [==,!=] int double [==,!=] char char [==,!=] double int [==,!=] char char [==,!=] int double [==,!=] double int [==,!=] int char [==,!=] char String [==,!=] String boolean [==,!=] boolean	==,!=	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

○ Operadores de comparación de tipos instanceof

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

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

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

○ Operaciones lógicas

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

En la tabla 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 (&&)
- Negación, conocida como NOT (!)

OPERANDO A	OPERANDO B	A | | B	A && B	!A
false	false	false	false	true
false	true	true	false	true
true	false	true	false	false
true	true	true	true	false

Consideraciones:

- Ambos operadores deberán ser de tipo booleano.

○ Operador ternario

Este operador retornará una expresión en función de una condición lógica.

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);
/**
*   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);
/**
*   Como la condición no se cumplió, el operador
*   ternario, retornó 69 luego ese 69 se sumó con el 1
*/
println("var2 = " + var2);
/**
*   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);
/**
*   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);
```

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.

Formato de código intermedio

El formato estándar de código de tres direcciones es una secuencia de proposiciones que tiene formato general x = y op z, donde x, y, z son 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.

Definición léxica

- #### Comentarios

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

- Los comentarios de una línea que serán delimitados al inicio con los símbolos “**;**” y al final con un carácter de finalización de línea

```
t1 = t1 + 1
t2 = t1 + 0
t3 = 10 + 5 ; comentario en esta linea
;otro comentario
t4 = t2 + t3
```

- #### 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.

- ## 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.

- ## 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.

- ## Palabras reservadas

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

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

- ## Símbolos

+	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				

- ## 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.

Definición de sintaxis

- Declaración de variables

Se podrán declarar variables de la siguiente forma

```
var p = 0
var h = 0
```

- Operadores aritméticos

OPERADOR	OPERACIÓN
+	suma
-	resta
*	multiplicación
/	división

- Operadores relacionales

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

- Operadores lógicos

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

- Destino de un salto

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

- Salto incondicional

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

- Salto condicional

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

- If

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

- ### IfFalse

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

- ### Declaración de métodos

Los métodos podrán ser declarados de la siguiente forma:

```
proc id {
    t1 = 10 + 11
    t2 = p + 0
    stack[t2] = t1
}
```

- ### Invocación a métodos

Se podrá invocar métodos de la siguiente forma:

```
call id
```

- ### 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.

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

```
t32 = 15.15
print('%d', t32)
```

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

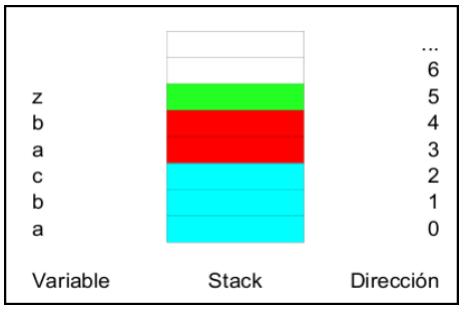
el código intermedio busca ser una representación próxima al código máquina, por lo que todas las sentencias de las que se compone se deben basar en las estructuras que componen el entorno de ejecución. Típicamente se puede decir que los lenguajes de programación cuentan con dos estructuras para realizar la ejecución de sus programas en bajo nivel, la pila (Stack) y el montículo (Heap), en la siguiente sección se describen estas estructuras.

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.

- ### El Stack y su puntero

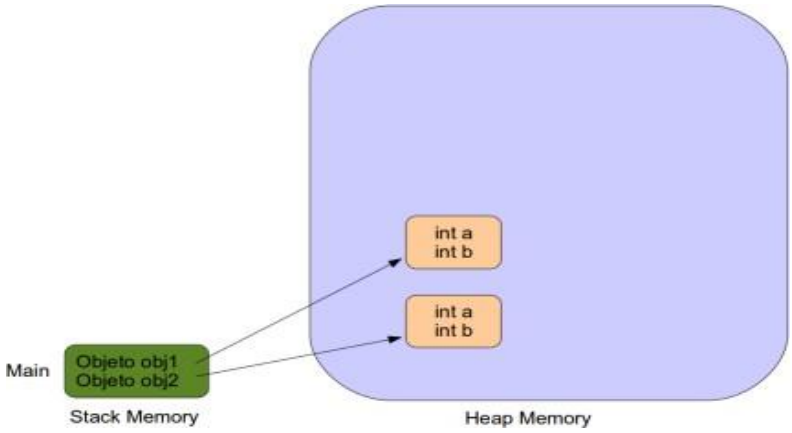
  El stack es la estructura que se utiliza en código intermedio para controlar las variables locales, y la comunicación entre métodos (paso de parámetros y obtención de retornos en llamadas a métodos). Se compone de ámbitos, que son secciones de memoria reservados exclusivamente para cierto grupo de sentencias.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



- ### El Heap y su puntero

  En OLCEV 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.



- 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[0] = t1
heap[t2] = 123.5
```

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

```
a = stack[3]
t1 = heap[t2]
```

Optimización de código intermedio

YABA 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:

Eliminación de instrucciones redundantes de carga y almacenamiento.

- ### Regla 1

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

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

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:

- ### 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.

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

- Regla 3

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.

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

- 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 falsa, se podrá transformar en un salto incondicional y eliminarse el salto hacia la etiqueta verdadera Lv.

Ejemplo	Optimización
`if 1 == 0 goto L1;` `goto L2;`	`goto L2;`

Simplificación algebraica y reducción por fuerza

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

- Regla 5

Eliminación de las instrucciones que tenga la siguiente forma:
Esto se realiza con todos los operandos algebraicos (+,-,*,/)

Ejemplo	Optimización
`x = y + 0;`	

- Regla 6

Eliminación de las instrucciones que tenga la siguiente forma:
Esto se realiza con todos los operandos algebraicos (+,-,*,/)

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

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 (OLCEV o 3D-YABA). 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.

Consideraciones

- Si se existiesen varios errores léxicos se podrá continuar con el análisis de la cadena de entrada y los errores detectados se mostrarán al final del análisis.
- Si se encontrara algún un error sintáctico en la cadena 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.

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 siguiente tabla.

A continuación, se presenta la precedencia de operadores lógicos.

NIVEL	OPERADOR	DESCRIPCION	ASOCIATIVIDAD

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

Entregables y Restricciones

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.

Restricciones

- La aplicación deberá ser desarrollada utilizando el lenguaje javascript (queda a discreción el framework o herramienta, recomendable hacer uso de Nodejs)
- Está permitido el uso de TypeScript.
- 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.

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:

TODO (por el momento, esto puede llegar a disminuir)

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 enero 2020 antes de las 23:59 horas