

# Documentación Compiladores e Intérpretes

## ETAPA 2: Analizador Sintáctico

Cejas Ramiro, LU: 120404

---

<b>Introducción</b>	<b>2</b>
<b>Fundamentos del Analizador Sintáctico</b>	<b>3</b>
Elementos de la gramática	3
Especificación de la gramática	3
<b>Implementación</b>	<b>4</b>
El módulo principal	4
Analizador Sintáctico	4
Excepciones	4
<b>¿Cómo compilar y utilizar el Analizador Sintáctico?</b>	<b>5</b>
<b>Logros</b>	<b>11</b>

## Introducción

Un analizador sintáctico es una herramienta que ayuda a entender la estructura gramatical de un texto o programa y se usa para verificar que el código esté escrito correctamente en términos de sintaxis. Es como un "verificador de gramática" para lenguajes humanos o de programación.

En este documento se verán los detalles del funcionamiento y la implementación de un analizador sintáctico para un lenguaje acotado de JAVA denominado Mini-JAVA.

Además se aclaran algunos aspectos a tener en cuenta a la hora de utilizarlo y cómo se debe hacer.

---

## Fundamentos del Analizador Sintáctico

### Elementos de la gramática

***terminal*** es un símbolo terminal

**<Clase>** es un símbolo no terminal (además la primer letra es una mayúscula)

$\epsilon$  representa la cadena vacía

**<X>::=a** representa una producción, con **a** una secuencia de terminales y no terminales

**<X>::=a / b** es una abreviación de **<X>::=a** y **<X>::=b**

### Especificación de la gramática

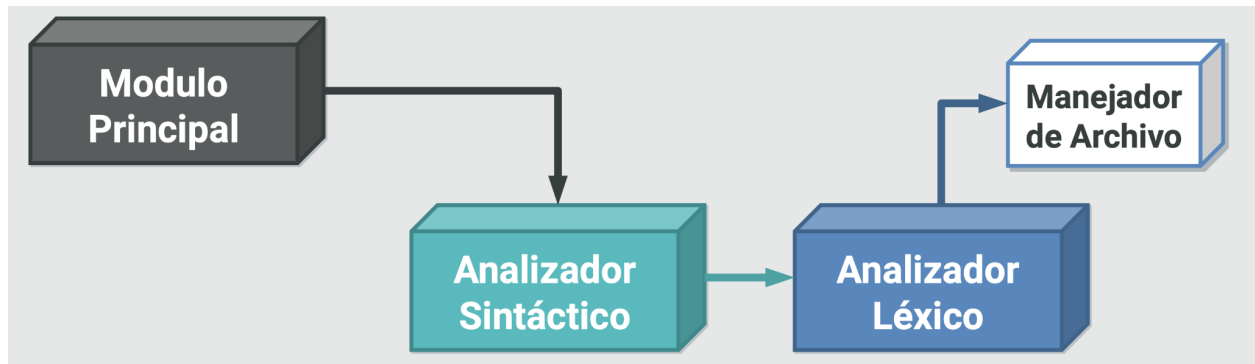
Presente en los archivos adjuntos, dentro del directorio **gramatica**, se encuentran las distintas versiones de la gramática.

gramaticaOriginal: La gramática original sin ningún cambio.

gramaticaSinRecursion: La gramática original sin recursión a izquierda.

gramaticaFactorizada: La gramática sin recursión a izquierda y factorizada, además se agregaron reglas y se modificaron algunas para cumplir con logros opcionales.

## Implementación



### El módulo principal

Proporciona una ruta a un archivo de texto que es suministrada al manejador de archivos, el cual es el que utiliza el Analizador Léxico como fuente para solicitar caracteres. Luego ese Analizador Léxico es suministrado al **Analizador Sintáctico** para realizar el chequeo.

En su método principal solicita al **Analizador Sintáctico** que realice el chequeo, además almacena todas las excepciones que se pueden producir y las muestra al finalizar.

### Analizador Sintáctico

La implementación fue llevada a cabo mediante un **Esquema Simple Descendente Recursivo**. El cual representa las producciones de la gramática LL(1) mediante llamados a métodos de manera encadenada y aprovecha el mecanismo de pila de llamados para representar la transformación total de la cadena, haciendo “match” con los tokens esperados y consumiendo los tokens suministrados por el Analizador Léxico.

### Excepciones

Las excepciones fueron modeladas extendiendo la clase Exception de JAVA con una nueva clase denominada SyntaxException, que se encarga de almacenar el lexema recibido y el nombre del token esperado, también se encarga de formatear el mensaje de error.

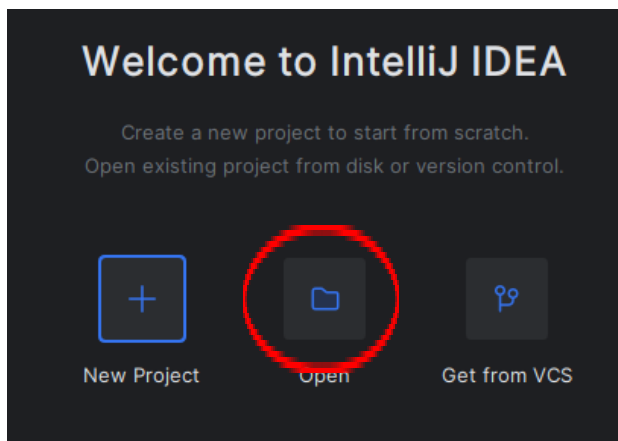
## ¿Cómo compilar y utilizar el Analizador Sintáctico?

Requisitos previos:

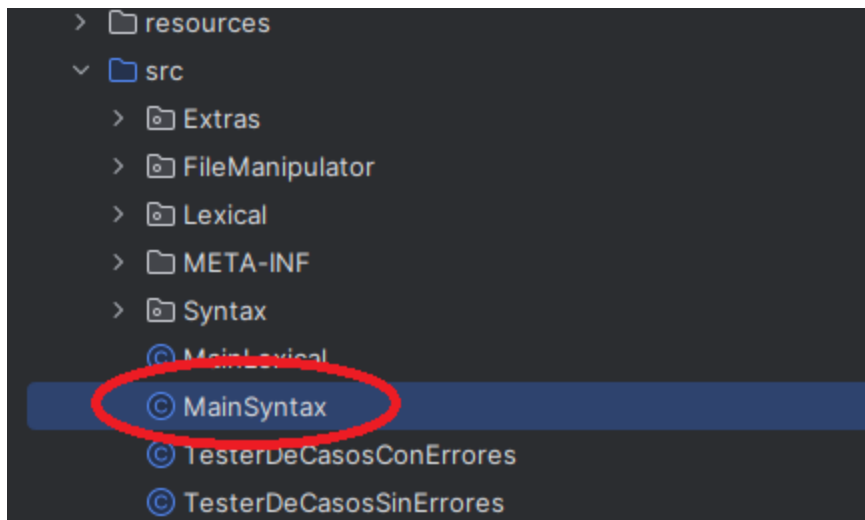
1. Instalar IntelliJ IDEA, se pueden usar cualquiera de las 2 variantes Ultimate o Community Edition, en este caso se utilizó la versión IntelliJ IDEA Community Edition 2023.2 (Podés descargarlo [acá](#)). ACLARACIÓN: El uso de este IDE no es restrictivo al funcionamiento del proyecto, pero la explicación se hará con IDEA.
2. Descargar el código fuente del siguiente [repositorio](#).
3. Descargar la versión de JDK 11 (Podés descargar la versión [Amazon Corretto](#)).

Ahora sí, manos a la obra:

1. Abrimos el proyecto descargado del repositorio con IDEA.



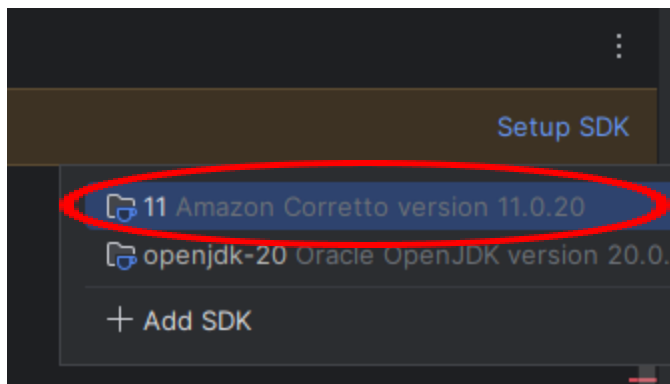
2. Veremos algo de este estilo, lo que tenemos que hacer es abrir el archivo Main.



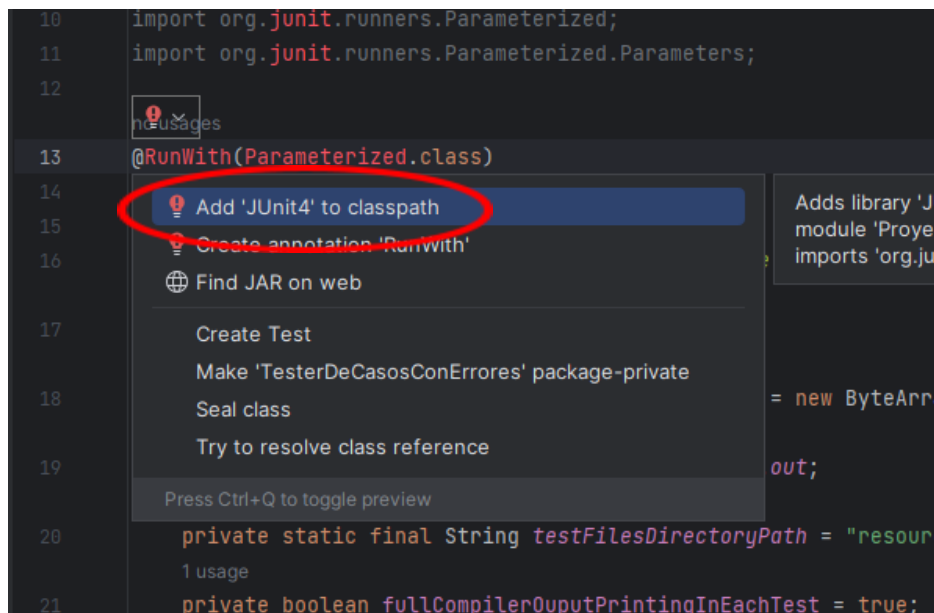
- Nos va a decir que el proyecto no tiene ningún JDK establecido. Entonces damos click en Setup SDK.



- En el menú desplegable elegimos la versión de JDK 11 que tengamos instalada. En este caso usamos Corretto Amazon. Y esperamos a que termine de configurar nuestro proyecto.

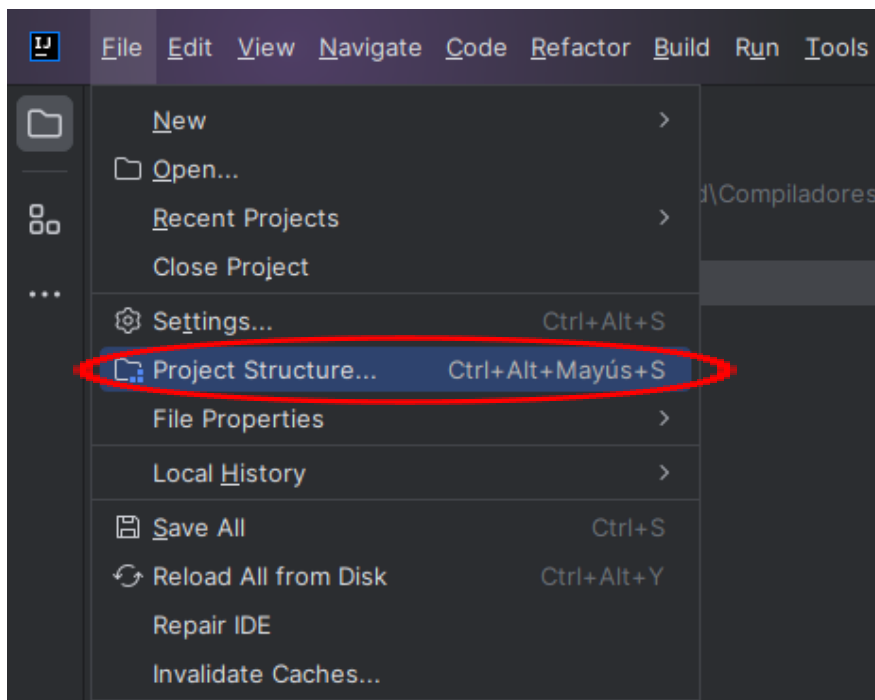


- En este paso se realiza lo necesario para disponer de testers para probar la funcionalidad, en caso de no quererlos, eliminar los archivos TesterDeCasosSinErrores y TesterDeCasosConErrores, además eliminar el directorio resources y saltar al paso 6. En caso de querer tener los testers lo tenemos que hacer es agregar JUnit4 a nuestro classpath. Para esto vamos al archivo TesterDeCasosSinErrores y presionamos cualquier error que nos arroje el editor de código, como sugerencia para solucionarlo nos va a dar la opción de agregar JUnit4 al classpath.



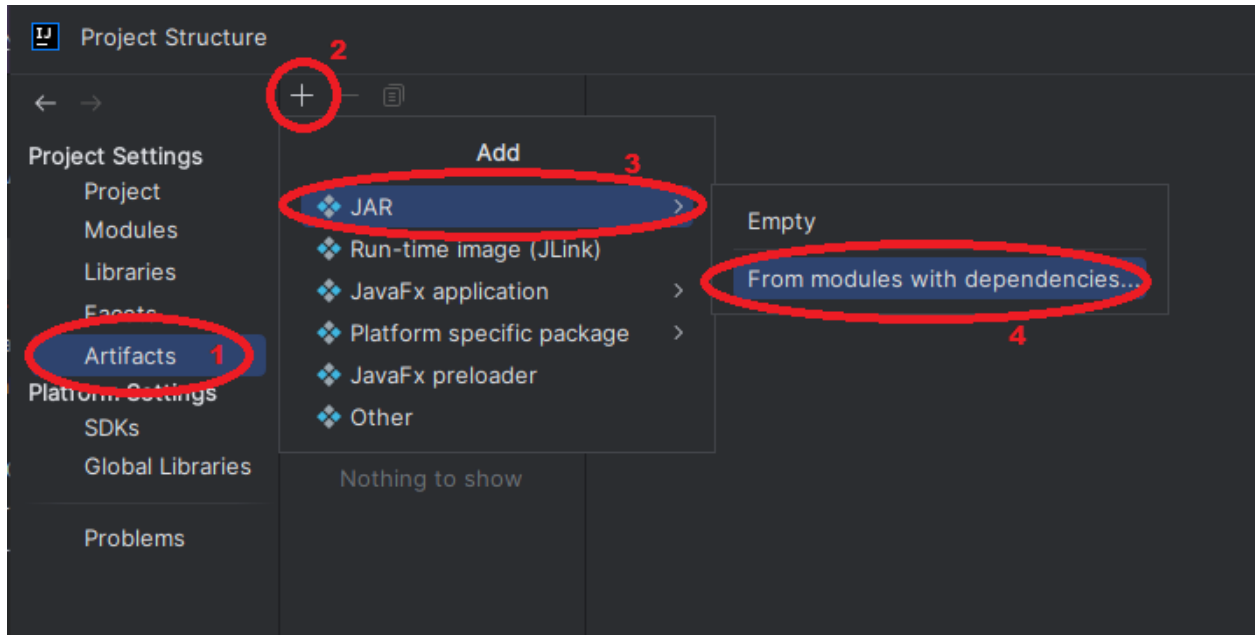
Ahora tenemos todo lo necesario para poder testear el código del analizador léxico y podemos ejecutar los test de JUnit. (Si nos sigue arrojando el error de JUnit, reiniciamos el IDE para que recargue el proyecto y se debería solucionar).

6. Para compilar el archivo principal de nuestro proyecto vamos a ir a File > Project Structure.

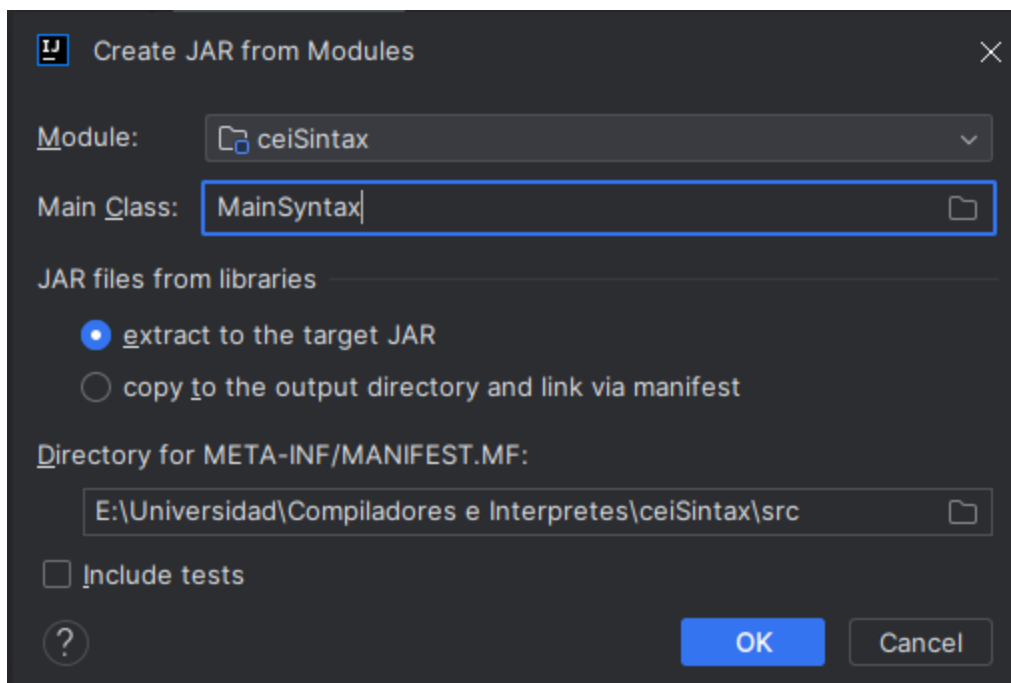




7. En la sección de Artifacts(1) vamos a presionar el botón +(2), luego en JAR(3) y por último en From modules with dependencies...(4).

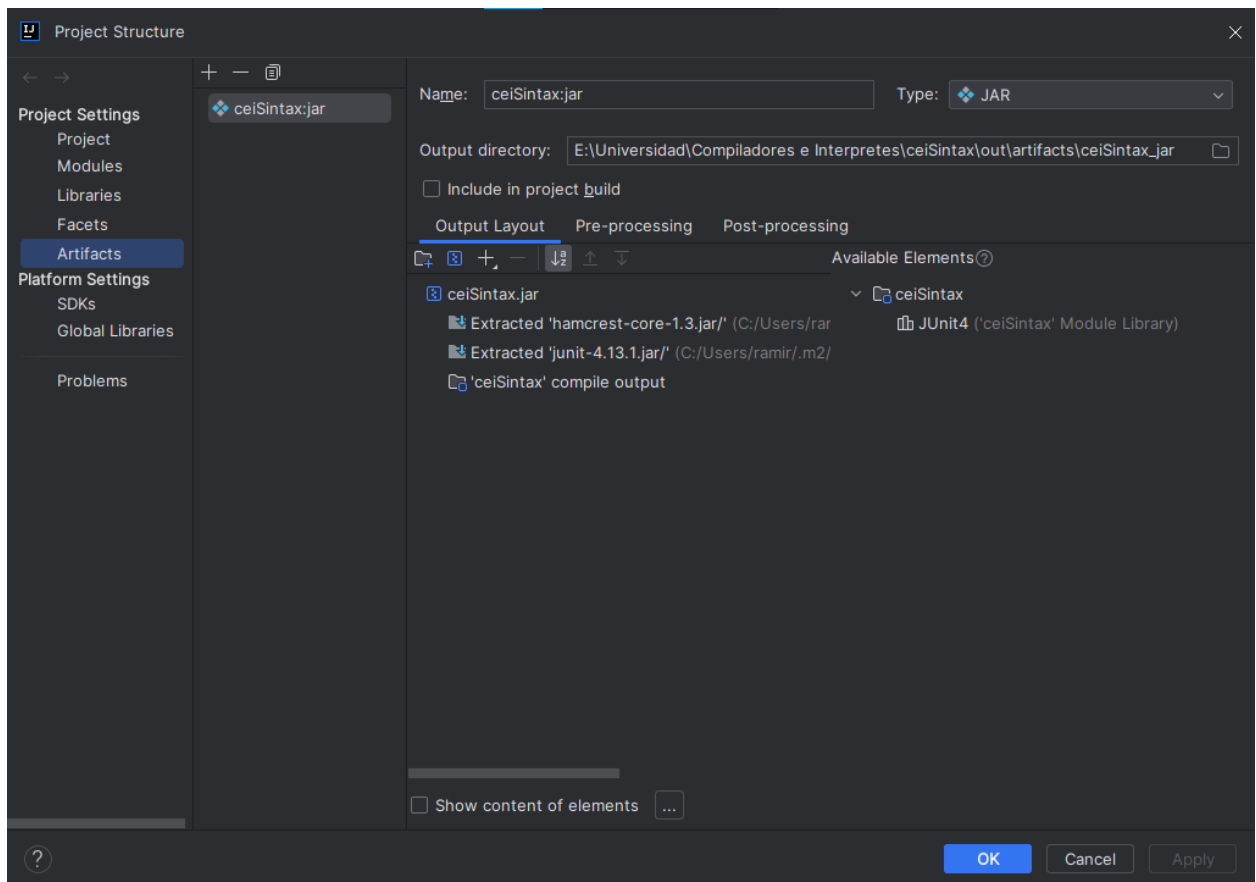


8. Luego elegimos que la clase principal sea Main y nos debería quedar así, ponemos OK.



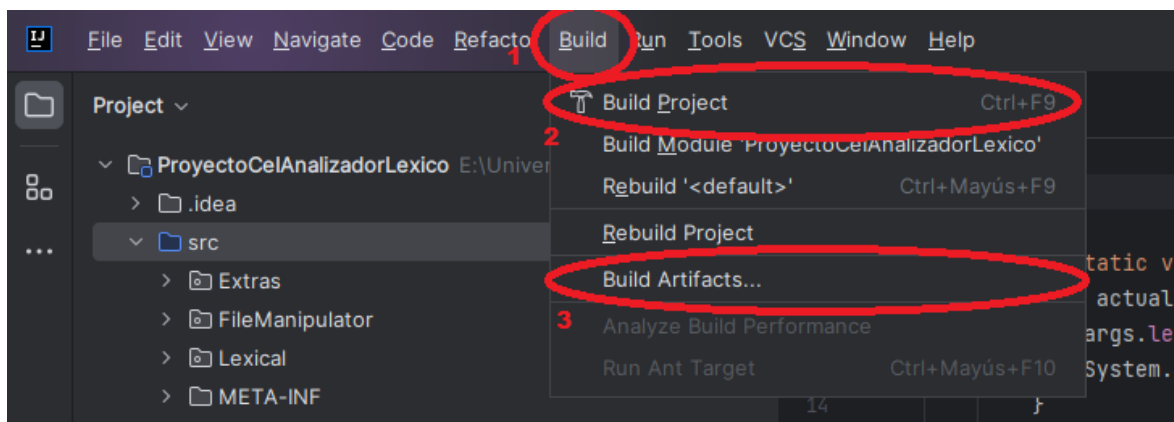
9. Nos debería quedar algo de este estilo, ponemos Apply y OK (si deseas podés cambiar el nombre del archivo que se va a generar donde dice Name:... , no le

borres los dos puntos seguidos de jar).

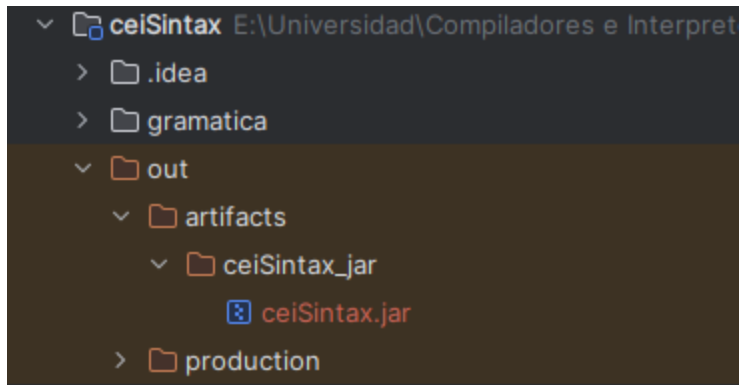


Con esto ya tenemos configurada la generación del JAR de nuestro proyecto.

10. Para generar el JAR vamos a Build y le damos primero en Build Project, esperamos que termine y luego en Build Artifacts.



11. Ahora veremos que se creó un directorio .out en nuestro proyecto, dentro del cual está el directorio del JAR, y dentro, el archivo JAR generado.




12. Para ejecutar el JAR y realizar el análisis sintáctico de un archivo fuente Mini-JAVA, vamos a utilizar la consola, ejecutamos el siguiente comando.

```
java -jar [Ruta de nuestro JAR] [Ruta del archivo a analizar]
```


Donde [Ruta de nuestro JAR] va a ser la ruta del archivo que acabamos de generar, y [Ruta del archivo a analizar] va a ser la ruta del archivo al cual queremos hacerle el análisis sintáctico.

## Logros


A continuación se mencionan los logros esperados de esta etapa del desarrollo:




**Entrega Anticipada Sintáctica**  
*(no valido para la reentrega, ni etapas subsiguiente)*  
 La entrega debe realizarse 48hs antes de la fecha limite





**Imbatibilidad Sintáctica**  
*(no valido para la reentrega, ni etapas subsiguiente)*  
 El software entregado pasa correctamente toda la batería de prueba utilizada por la cátedra



**Genericidad**  
 El compilador permite declarar y utilizar clases genéricas con tipos paramétricos. Dese el punto de vista sintáctico su declaración y uso son como en Java, salvo que los tipos paramétricos en la declaración usan ids de Clase. Al igual que en Java se permite utilizar la notación diamante <> en la invocación de constructores de estas clases  
*(Este logro esta asociado a otros logros en futuras entregas!)*



**Floats! - Sintacticos** Requiere: 1   
 El compilador permite usar los literales floats en los contextos adecuados y permite declarar entidades de tipo float  
*(Este logro esta asociado a otros logros en futuras entregas!)*



**Atributos Inicializados**  
 El compilador permite realizar inicializaciones de los atributos cuando son declarados, al igual que en Java  
*(Este logro esta asociado a otros logros en futuras entregas!)*

---

## Aclaraciones Generales

- El analizador sintáctico puede recibir el argumento **-v** luego de la ruta del archivo a escanear para habilitar el modo **verbose** para ver por consola las transformaciones que fue realizando el analizador.