



Documentación Compiladores e Intérpretes

ETAPA 3: Analizador Semántico

Cejas Ramiro, LU: 120404

| | |
|--|-----------|
| Introducción | 2 |
| Fundamentos del Analizador Semántico | 3 |
| Intuición inicial | 3 |
| Tabla de símbolos | 3 |
| Clases | 3 |
| Interfaces | 3 |
| Métodos | 4 |
| Atributos | 4 |
| Implementación | 5 |
| El módulo principal | 5 |
| Analizador Sintáctico | 5 |
| Excepciones | 5 |
| ¿Cómo compilar y utilizar el Analizador Sintáctico? | 6 |
| Logros | 12 |
| Logros de etapas anteriores (segundo round) | 13 |
| Aclaraciones Generales | 13 |

Introducción

Un analizador semántico es una herramienta en la programación que ayuda a comprender el significado de un texto o código fuente. Identifica cómo las palabras y estructuras se relacionan y qué sentido tienen en un contexto. Esto es útil para verificar que el código esté escrito correctamente y cumpla con su propósito.

En esta etapa únicamente trabajaremos con **chequeo de declaraciones**.

En el contexto del chequeo de declaraciones, un analizador semántico se encarga de examinar las declaraciones en un programa para asegurarse de que tengan sentido desde un punto de vista lógico y semántico. Esto ayuda a evitar errores lógicos en el código.

En este documento se verán los detalles del funcionamiento y la implementación de un analizador semántico 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 Semántico

Intuición inicial

A partir de la etapa anterior, que es el análisis sintáctico, podemos identificar secciones en el código donde se realizan declaraciones, como clases, interfaces, atributos, parámetros y métodos. La pregunta clave es cómo organizar esto de manera coherente y asegurarnos de que todas las declaraciones sean correctas.

Una posible solución consiste en utilizar una tabla de símbolos que almacene los datos y realice los controles necesarios (delegación de responsabilidades). De esta manera, solo necesitamos determinar cuándo agregar un método o un atributo y a qué clase pertenecen.

Tabla de símbolos

Una **tabla de símbolos** es una estructura de datos utilizada en programación para mantener un registro de los símbolos, como variables, funciones y otros identificadores, que se utilizan en un programa. Cada símbolo se asocia con información relevante, como su tipo de datos y su ubicación en el código fuente. Esto facilita el análisis del programa durante la compilación o la ejecución, ya que permite a la computadora entender y gestionar adecuadamente los símbolos utilizados en el código. Es como un diccionario que organiza información sobre los elementos del programa para su correcta interpretación.

En nuestro caso se verá poblado con Clases, Interfaces, Métodos y Atributos.

Clases

Las clases van a almacenar su nombre, sus atributos, y sus métodos, a su vez, pueden heredar otras clases, implementar interfaces y realizar chequeos sobre sus miembros.

Interfaces

Las interfaces van a almacenar su nombre y todos sus métodos declarados, ninguno de estos puede ser estático, y no puede declarar al método `debugPrint`.

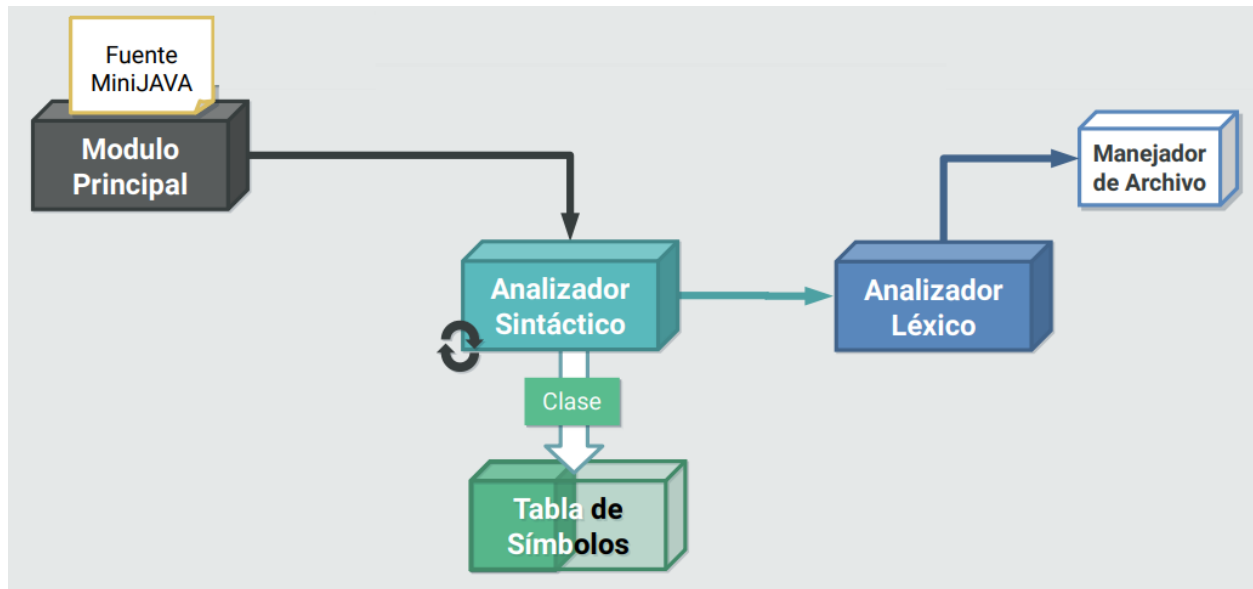
Métodos

Los métodos van a almacenar su nombre, su tipo de retorno, si son o no estáticos y su listado de parámetros (también como atributos con ciertas restricciones).

Atributos

Los atributos van a almacenar su nombre, su tipo y si son o no estáticos.

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. Durante este chequeo, también se realiza el análisis semántico. En su método principal solicita al *Analizador Semántico* que realice el chequeo, además almacena todas las excepciones que se pueden producir y las muestra al finalizar.

Analizador Sintáctico

Además de lo que ya realizaba el analizador sintáctico, fué modificado para también realizar el análisis semántico, o mejor dicho, para ir poblando la tabla de símbolos con clases, interfaces, métodos y atributos.

Excepciones

Las excepciones fueron modeladas extendiendo la clase Exception de JAVA con una nueva clase denominada SemanticException, que almacena el token que generó la excepción y el mensaje de la explicación de porqué surgió.

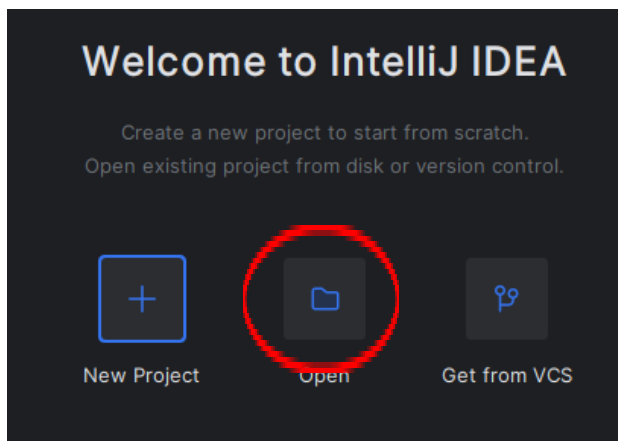
¿Cómo compilar y utilizar el Analizador Sintáctico?

Requisitos previos:

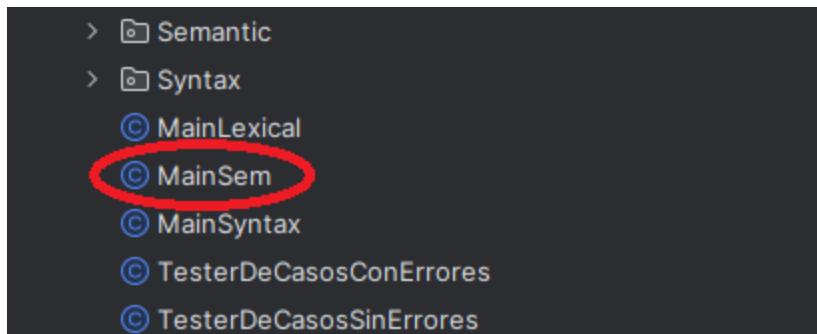
1. Instalar IntelliJ IDEA, se pueden usar cualquiera de las 2 variantes Ultimate o Community Edition, en este caso se utilizó la versión IntelliJ IDEA Community Edition 2023.2 (Podés descargarlo [acá](#)). ACLARACIÓN: El uso de este IDE no es restrictivo al funcionamiento del proyecto, pero la explicación 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:

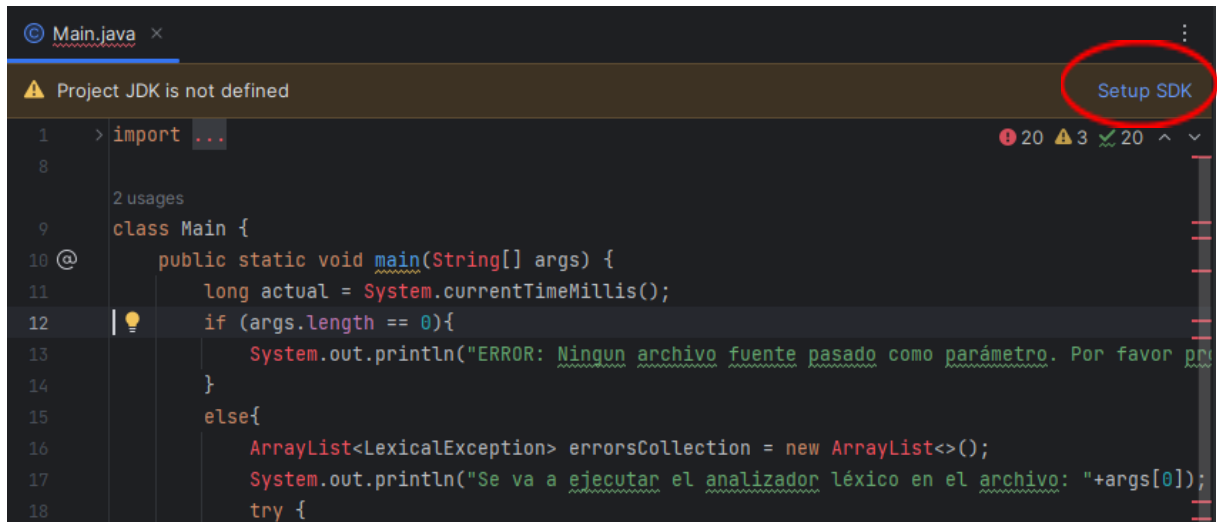
1. Abrimos el proyecto descargado del repositorio con IDEA.



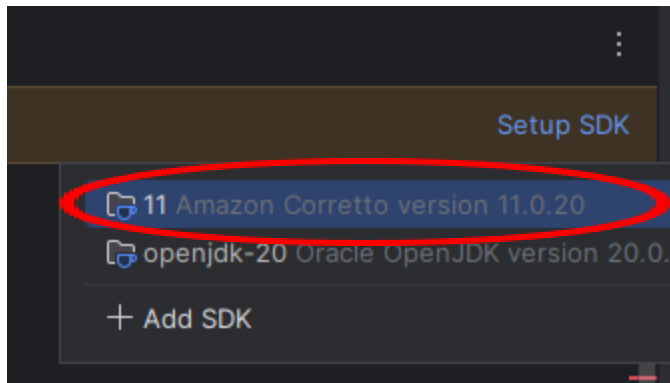
2. Veremos algo de este estilo, lo que tenemos que hacer es abrir el archivo Main.



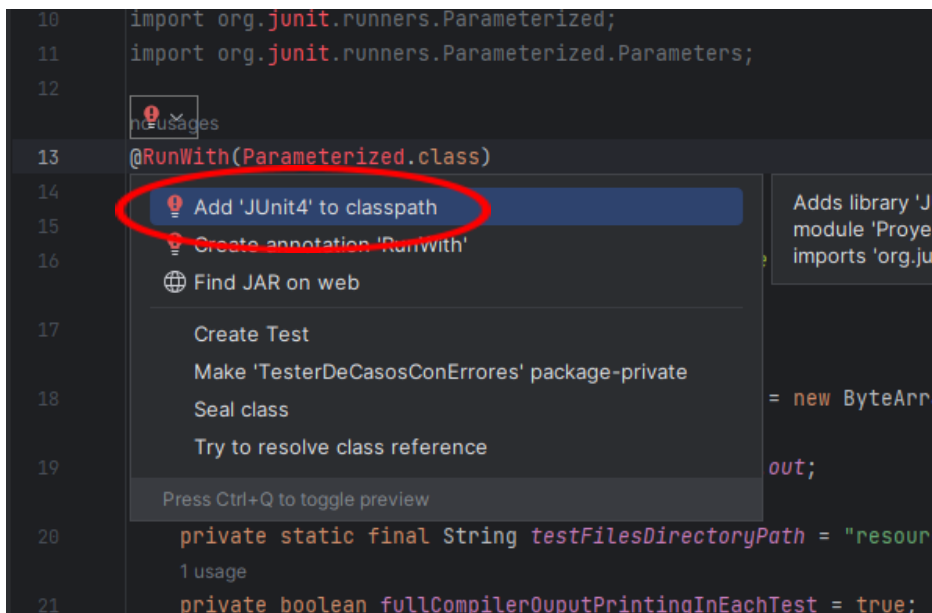
3. Nos va a decir que el proyecto no tiene ningún JDK establecido. Entonces damos click en Setup SDK.



4. 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.

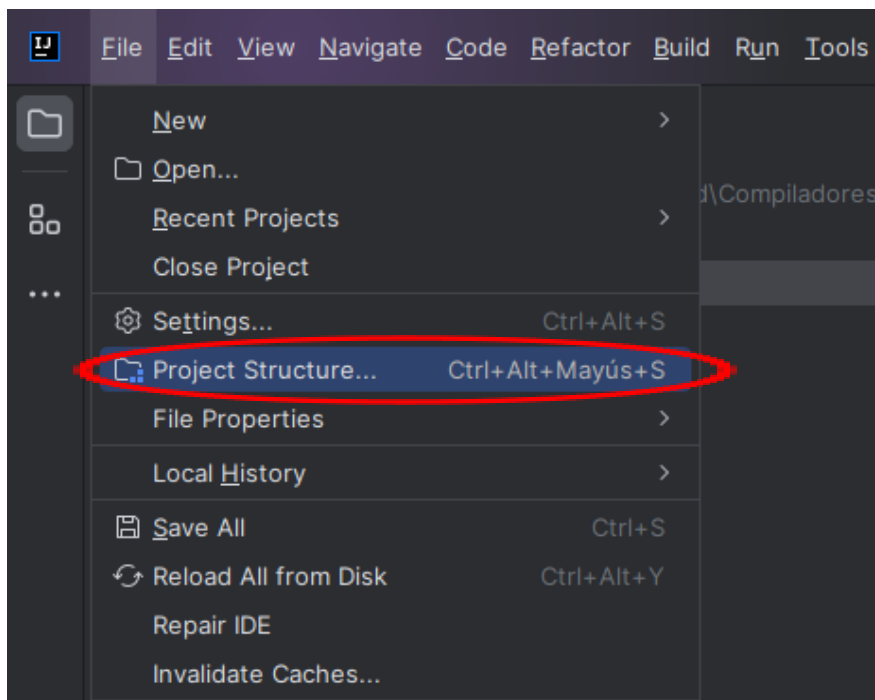


5. En este paso se realiza lo necesario para disponer de testers para probar la funcionalidad, en caso de no quererlos, eliminar los archivos TesterDeCasosSinErrores y TesterDeCasosConErrores, además eliminar el directorio resources y saltar al paso 6. En caso de querer tener los testers lo tenemos que 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.

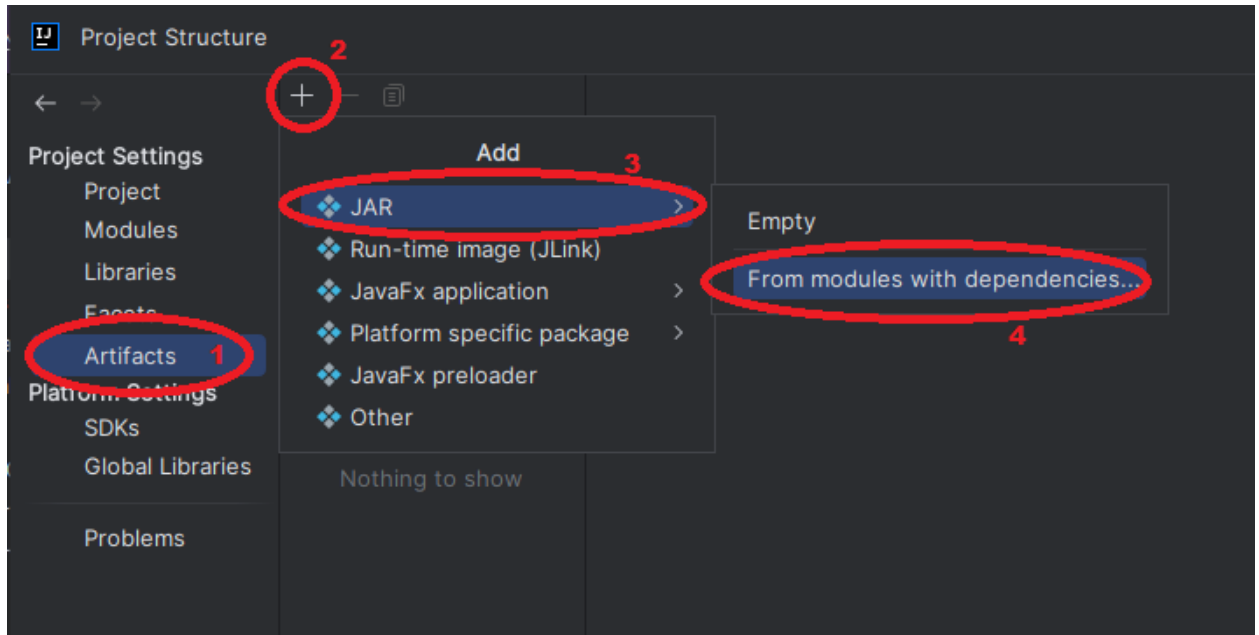


Ahora tenemos todo lo necesario para poder testear el código del analizador léxico y podemos ejecutar los test de JUnit. (Si nos sigue arrojando el error de JUnit, reiniciamos el IDE para que recargue el proyecto y se debería solucionar).

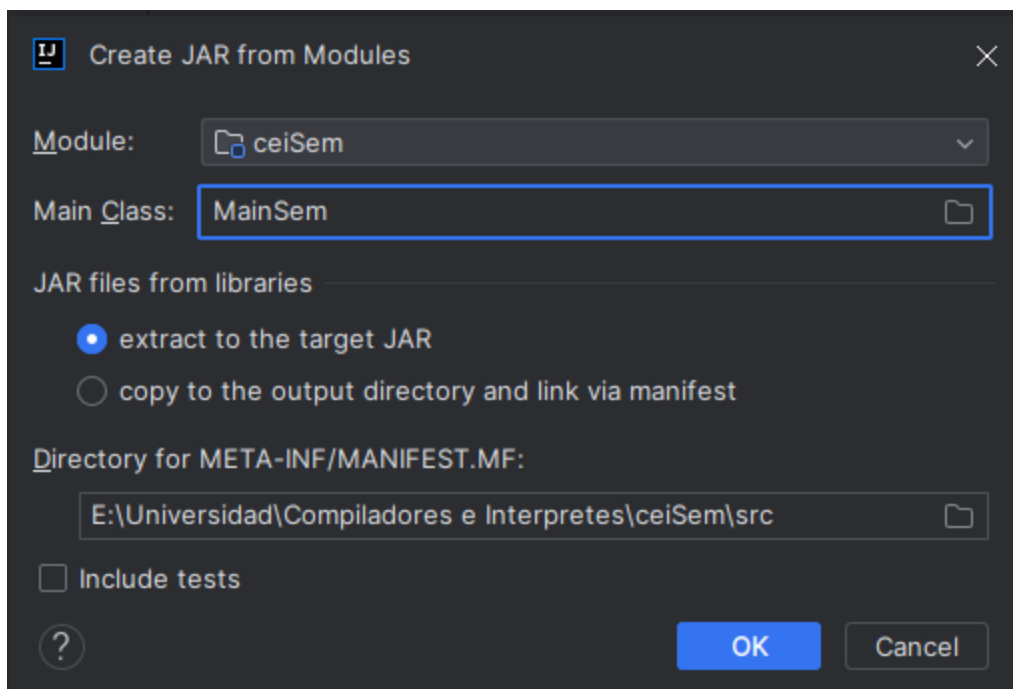
6. Para compilar el archivo principal de nuestro proyecto vamos a ir a File > Project Structure.



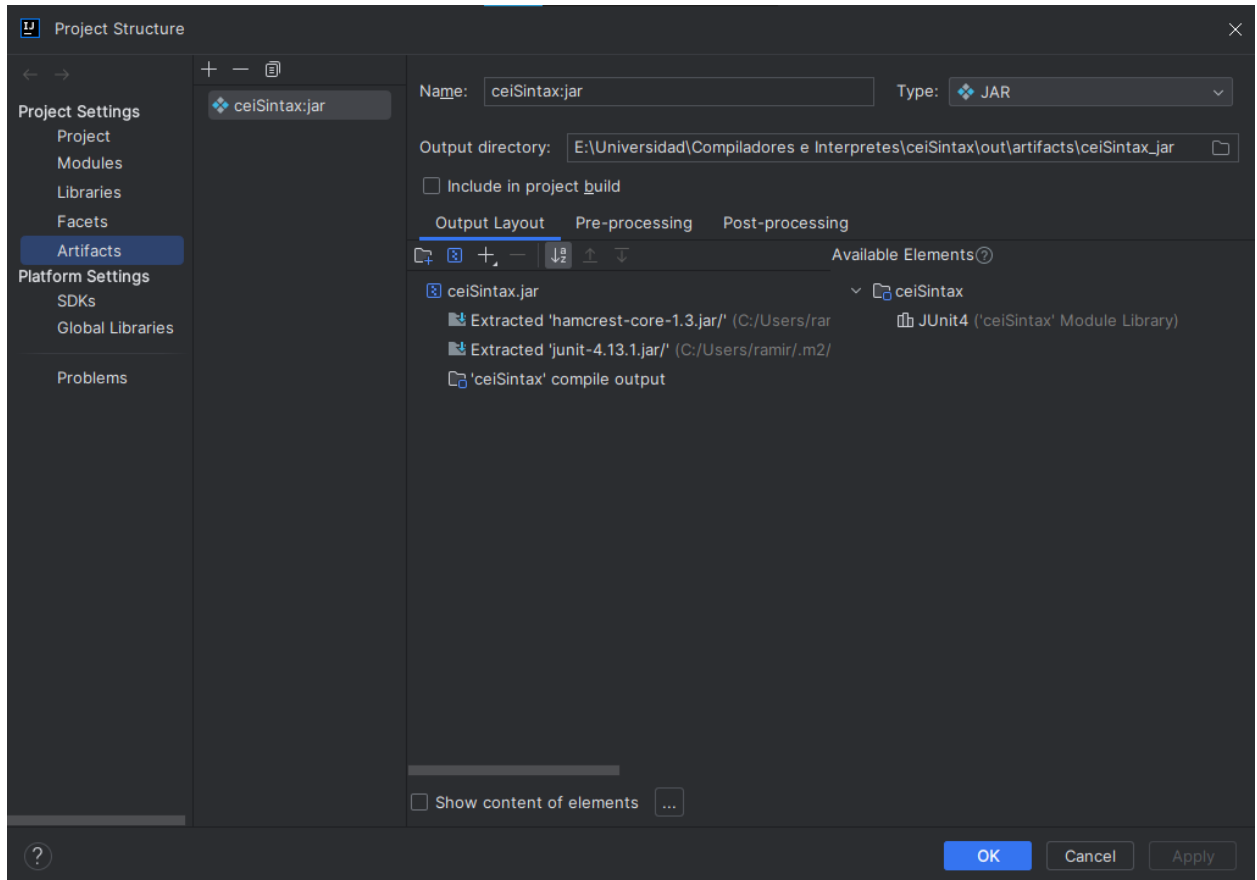
7. En la sección de Artifacts(1) vamos a presionar el botón +(2), luego en JAR(3) y por último en From modules with dependencies...(4).



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

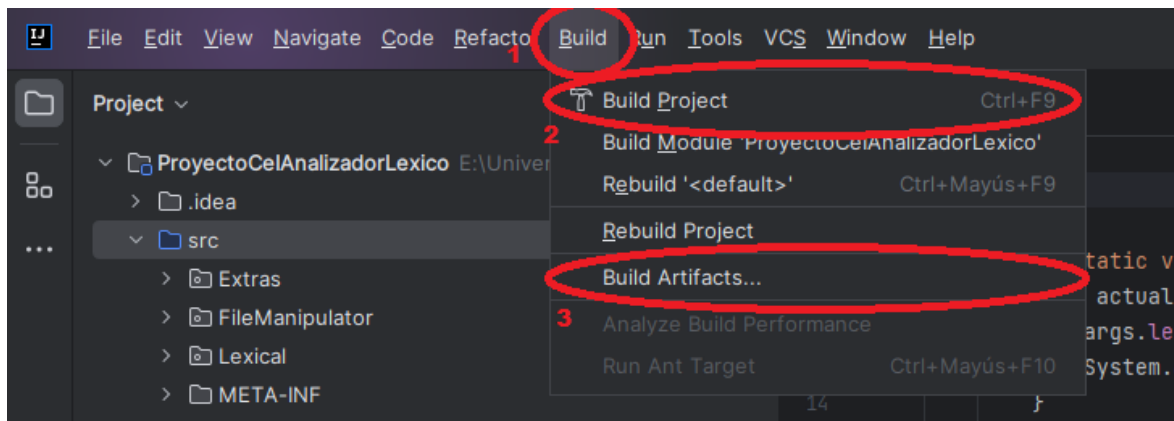


9. Nos debería quedar algo de este estilo, ponemos Apply y OK (si deseas podés cambiar el nombre del archivo que se va a generar donde dice Name:..., no le borres los dos puntos seguidos de jar).

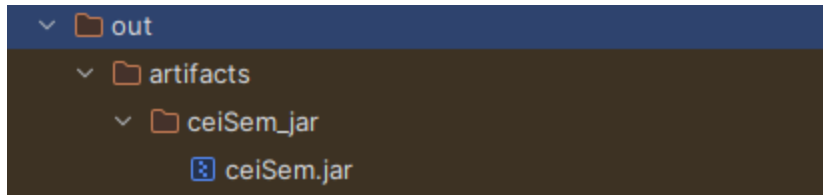


Con esto ya tenemos configurada la generación del JAR de nuestro proyecto.

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 semántico 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 semántico.

Logros


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




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



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



Multi-Detección Errores Semanticos en Declaraciones
 El compilador no finaliza la ejecución ante el primer error semantico en una declaración (se recupera) y es capaz de reportar otros errores que tenga el programa fuente en una corrida. Como mínimo se espera que el compilador pueda continuar con el análisis de la próxima declaración y que cuando hay nombres repetidos descarte ambas entidades.



Atributos Tapados
 Al igual que en java el compilador permite que una clase se declaren atributos con el mismo nombre que los definidos en sus superclases.
(Este logro esta asociado a otros logros en futuras entregas!)

Logros de etapas anteriores (segundo round)



Genericidad

El compilador permite declarar y utilizar clases genéricas con tipos paramétricos. Desde 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 está asociado a otros logros en futuras entregas!)

Aclaración: Para chequearlo se deberá utilizar el main de la etapa Sintáctica llamado MainSyn, ya que no se realizó genericidad en la etapa actual.

Aclaraciones Generales

- El analizador semántico 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 y la tabla de símbolos resultante.