IDE: 100 warnings, no errors
Compiler: Compiled successfully

IDE: 100 warnings, 1 error
Compiler:

Documentación Compiladores e Intérpretes

ETAPA 4: Analizador Semántico 2

Cejas Ramiro, LU: 120404

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 sentencias**.

En el contexto del chequeo de sentencias, un analizador semántico se encarga de examinar las sentencias 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.

IMPORTANTE

El proyecto que se describe en este documento corresponde a la versión SecondSemantic de los archivos fuente.
La versión FirstSemantic corresponde a errores solucionados de la etapa anterior.

AST

Todos los Nodos descritos debajo implementan a una interfaz común Node, que establece que todos deben tener los métodos:

- getType() : Token
- setParentBlock(NodeBlock) : void
- check(SymbolTable) : boolean

Entonces, todos los nodos van a tener tipo (en caso de no ser necesario su tipo será null), una referencia al bloque padre que los contiene y utilizarán la tabla de símbolos a la hora de realizar el chequeo.

1. EmptyNode: Representa la sentencia vacía.
   No almacena nada, y su check siempre da válido.

2. NodeAssignment: Representa la sentencia de asignación.
   Primero realiza el chequeo de ambos lados de la asignación, luego verifica que el lado izquierdo sea asignable y que el tipo del lado derecho conforme con el tipo del lado izquierdo. Su tipo es el del lado derecho de la asignación.

3. NodeExpression: Clase abstracta que representa todas las expresiones.
   Delega los controles a las clases que lo extiendan.

4. NodeBinaryOp: Representa la expresión binaria. (Extiende a NodeExpression)
   Primero realiza el chequeo de ambos lados de la expresión y luego, dependiendo el operador asociado, verifica que los tipos sean compatibles con él, también seteando el tipo de retorno de la operación.

5. NodeUnaryOp: Representa la expresión unaria. (Extiende a NodeExpression)
   Primero realiza el chequeo de su operando y luego verifica que el tipo del operando sea compatible con el operador. Su tipo es el mismo que el del operando.

6. NodeVarDeclaration: Representa la sentencia que declara una variable local.
   Primero chequea la expresión del lado derecho de la asignación. Después

verifica que el nombre de la variable no está sobreescribiendo al nombre de otra variable visible, luego, verifica que el tipo del lado derecho de la asignación sea una clase (o interfaz) existente o bien un tipo primitivo. Por último agrega esa variable a las variables locales del bloque que lo contiene. Su tipo es el tipo del lado derecho.

7. NodeIf: Representa la sentencia ***If***.
   Primero chequea la condición, luego verifica que el tipo de la condición sea boolean, después chequea el bloque “then” y por último, si tiene, verifica el bloque “else”. No tiene tipo.

8. NodeWhile: Representa la sentencia ***While***.
   Primero chequea la condición, luego verifica que el tipo de la condición sea boolean, después chequea el bloque “body”. No tiene tipo.

9. NodeBlock: Representa un bloque.
   Almacena una lista de sentencias, una referencia al método al que pertenecen y una referencia a la clase que los contiene, para poder acceder tanto a los atributos de clase como los parámetros del método, también almacena una lista de variables locales que se va poblando a medida que se chequeen los NodeVarDeclaration (si es que tiene).
   Para chequearse agrega todos los atributos de la clase a un listado propio, los parámetros del método también a un listado propio y si tiene un bloque padre, hereda todos las variables locales allí visibles. Por último chequea todas las sentencias que contenga. No tiene tipo.

10. NodeLiteral: Representa un literal.
    Almacena el token asociado al literal. Para chequearse simplemente setea su tipo al tipo del token.

11. NodeReturn: Representa la sentencia ***return***.
    Primero chequea la expresión asociada, luego verifica que el tipo de la expresión conforme con el tipo de retorno del método que lo contiene, tiene en cuenta que el tipo del método puede ser void, por lo que no debería tener expresión asociada, también tiene en cuenta que, en caso de que el tipo de la expresión sea

null entonces puede estar en cualquier método que retorne algo de tipo referencia (no primitivo). Su tipo es el tipo de la expresión.

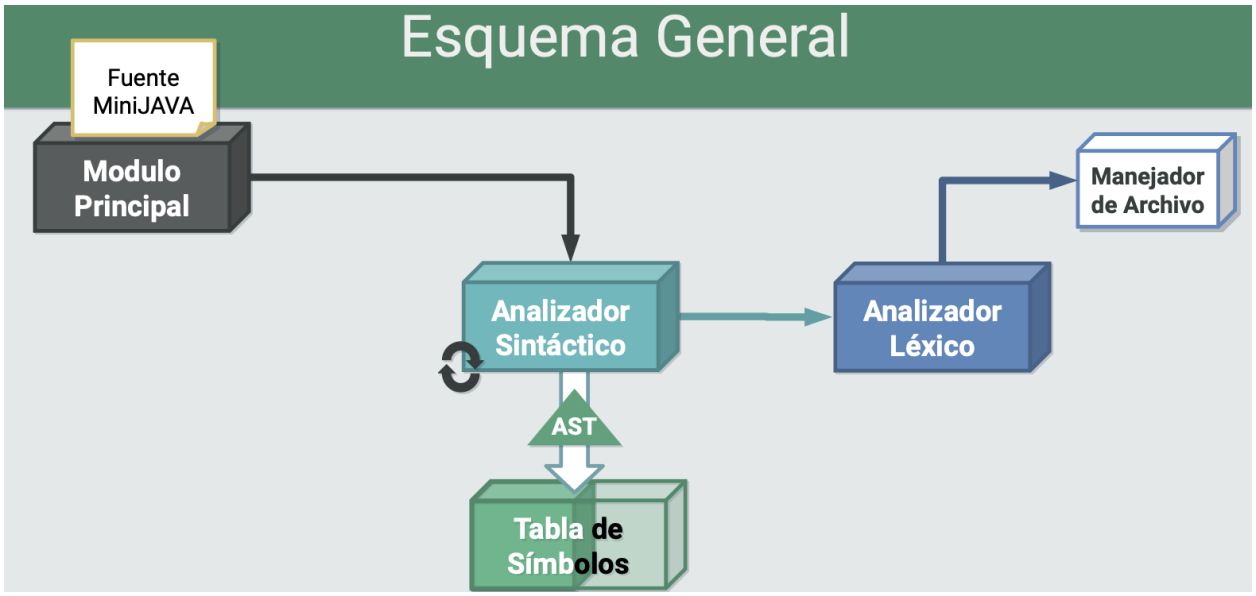
12. NodeVariable: Representa una variable o un acceso.
    Todos los NodeVariable y sus versiones más específicas mantendrán una referencia a una posible cadena hijo y una cadena padre, es decir, una cadena de llamados doblemente enlazada en donde cada uno conoce a su predecesor y su sucesor. Todos le piden a su sucesor que se chequee.
    Todos los tipos son seteados parcialmente hasta que el último hijo se chequee, que propaga su tipo a todos sus predecesores.
    Para chequearse se tiene en cuenta si es un método o un acceso a una variable.
    a. Si es un método, chequea sus argumentos y:
        i. Si no tiene padre, busca en la clase actual algún método con el mismo nombre y la misma cantidad y tipos de parámetros.
        ii. Si tiene padre, busca en el tipo de su padre, que tiene que ser un tipo clase, el algún método con el mismo nombre y la misma cantidad y tipos de parámetros.
    b. Si es un atributo:
        i. Si no tiene padre, busca en las variables visibles del bloque actual alguna que tenga el mismo nombre.
        ii. Si tiene padre, busca en la clase del tipo del padre (tiene que ser un tipo clase) algún atributo de clase con el mismo nombre.

13. NodeVariableConstructor: Representa la sentencia ***new***.
    Para chequearse verifica que la clase que está intentando crear exista (está declarada en la tabla de símbolos), chequea sus argumentos y por último verifica que el constructor de esa clase tenga la misma cantidad y tipos de parámetros. Su tipo es el de la clase que está creando.

14. NodeVariableStaticMethod: Representa la llamada a un método estático.
    Primero verifica que su padre sea el nombre de una clase, luego se asegura que esa clase esté declarada y por último, chequea los argumentos y verifica que esa clase tenga el método estático que se está intentando utilizar, con la misma cantidad y tipos de parámetros. Su tipo es el tipo de retorno del método.

15. NodeVariableThis: Representa la sentencia ***this***.
    Para chequearse simplemente sea su tipo al tipo de la clase que lo contiene y le pide a su cadena hijo, si es que tiene que se chequee.

Implementación



Analizador Sintáctico

Además de lo que yá 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. Además en esta última etapa 4 fue modificado para también agregar las sentencias a los bloques correspondientes.

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.

ACLARACIÓN: Se desactivó la posibilidad de reportar múltiples excepciones en una única pasada (tarea para la entrega 5).

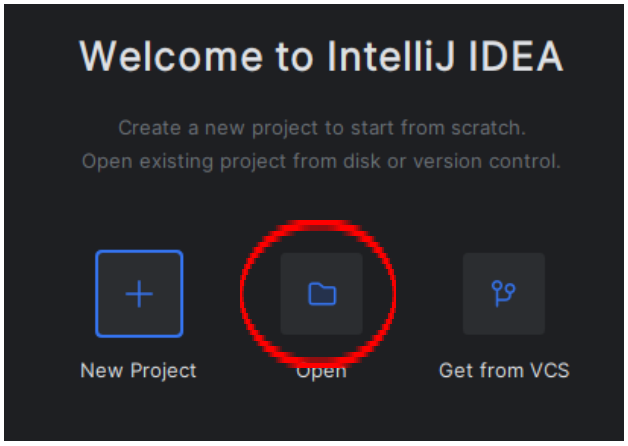
¿Cómo compilar y utilizar el Analizador Semántico?

Requisitos previos:

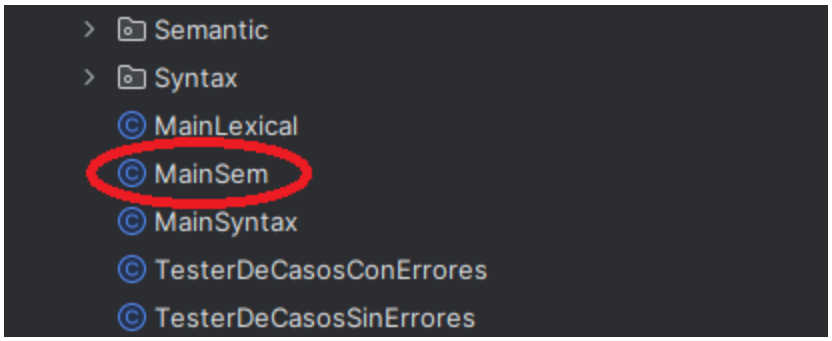
1. Instalar Intellij IDEA, se pueden usar cualquiera de las 2 variantes Ultimate o Community Edition, en este caso se utilizó la versión Intellij IDEA Community Edition 2023.2 (Podés descargarlo acá). ACLARACIÓN: El uso de este IDE no es restrictivo al funcionamiento del proyecto, pero la explicación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

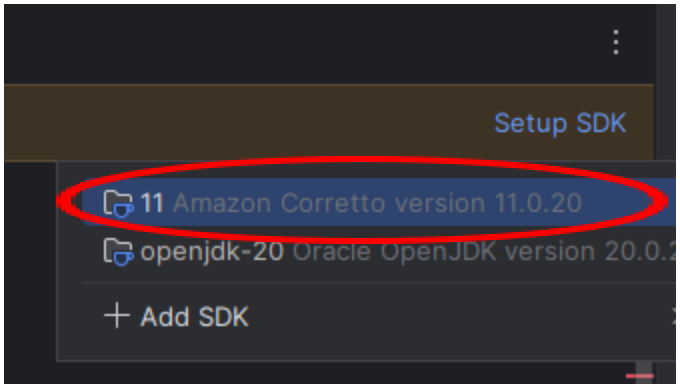


2. Veremos algo de este estilo, lo que tenemos que hacer es abrir el archivo Main.

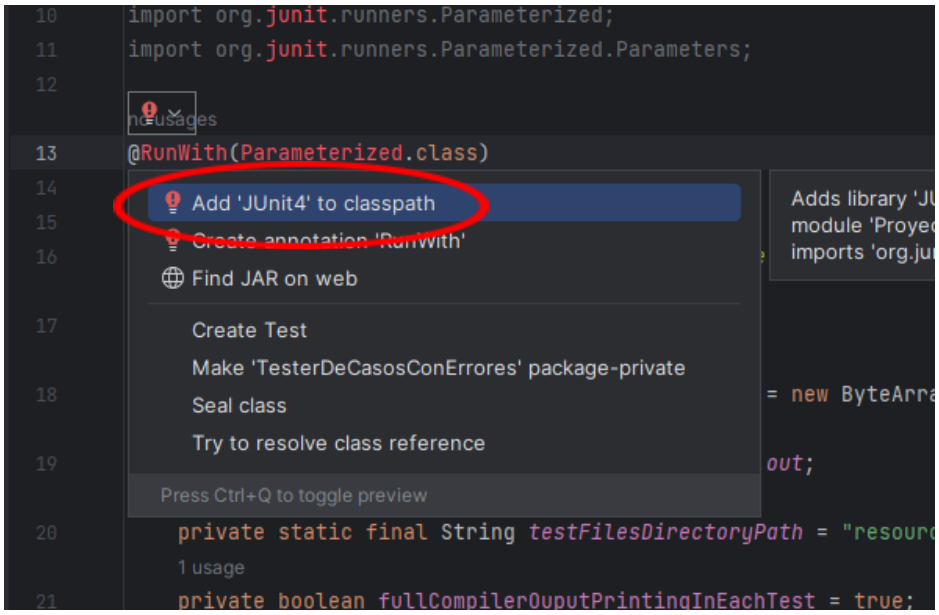


3. Nos va a decir que el proyecto no tiene ningún JDK establecido. Entonces damos click en Setup SDK.

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.

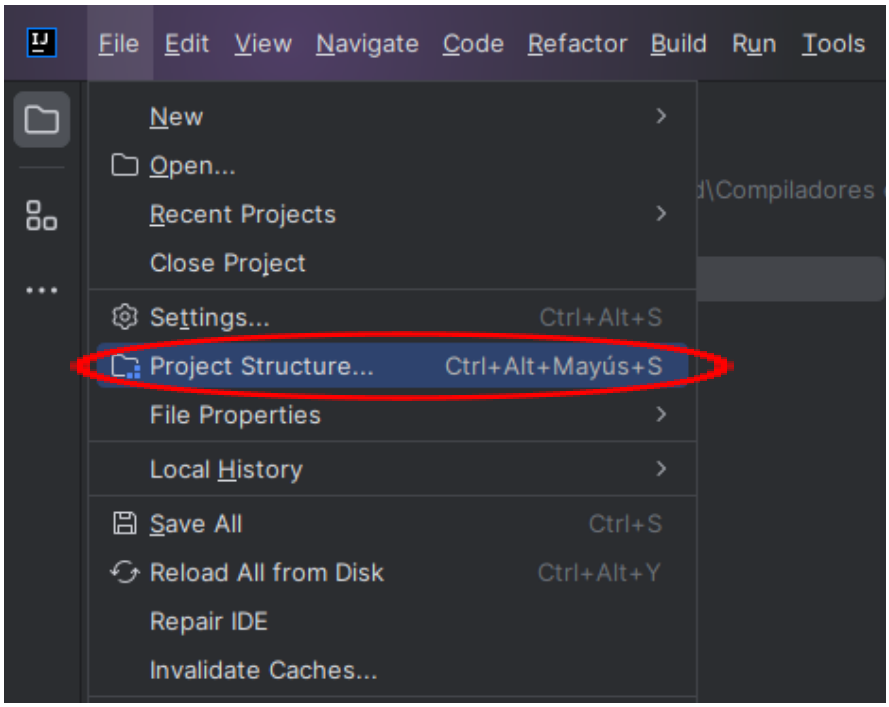


5. En este paso se realiza lo necesario para disponer de testers para probar la funcionalidad, en caso de no quererlos, eliminar los archivos TesterDeCasosSinErrores y TesterDeCasosConErrores, además eliminar el directorio resources y saltar al paso 6. En caso de querer tener los testers lo tenemos que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

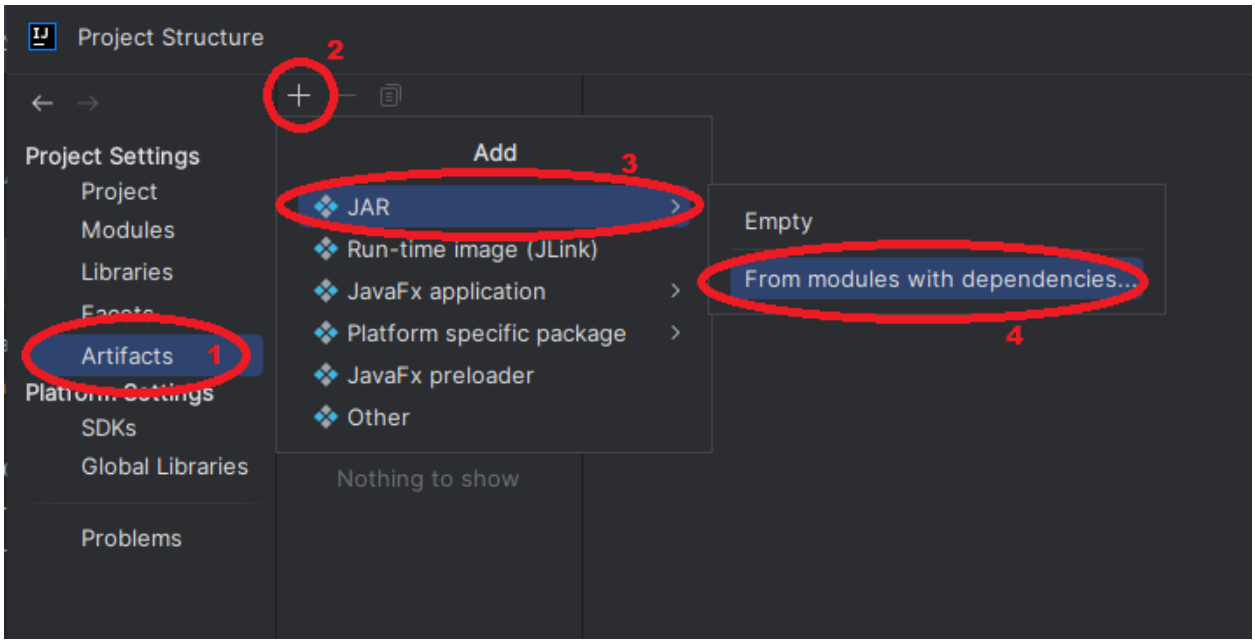


Ahora tenemos todo lo necesario para poder testear el código del analizador léxico y podemos ejecutar los test de JUnit. (Si nos sigue arrojando el error de JUnit, reiniciamos el IDE para que recargue el proyecto y se debería solucionar).

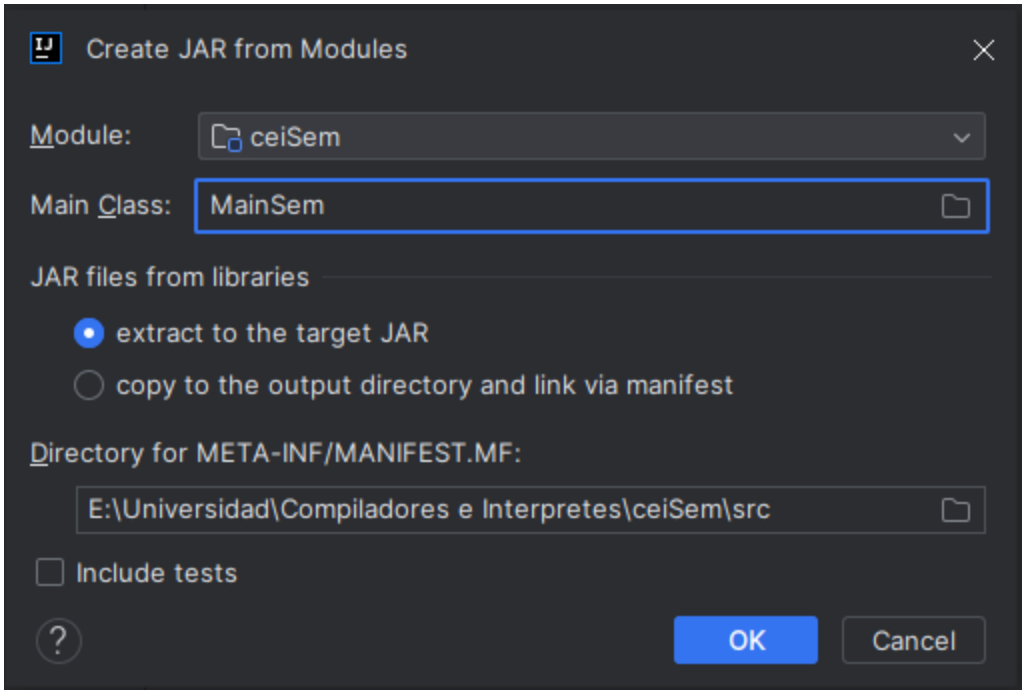
6. Para compilar el archivo principal de nuestro proyecto vamos a ir a File > Project Structure.



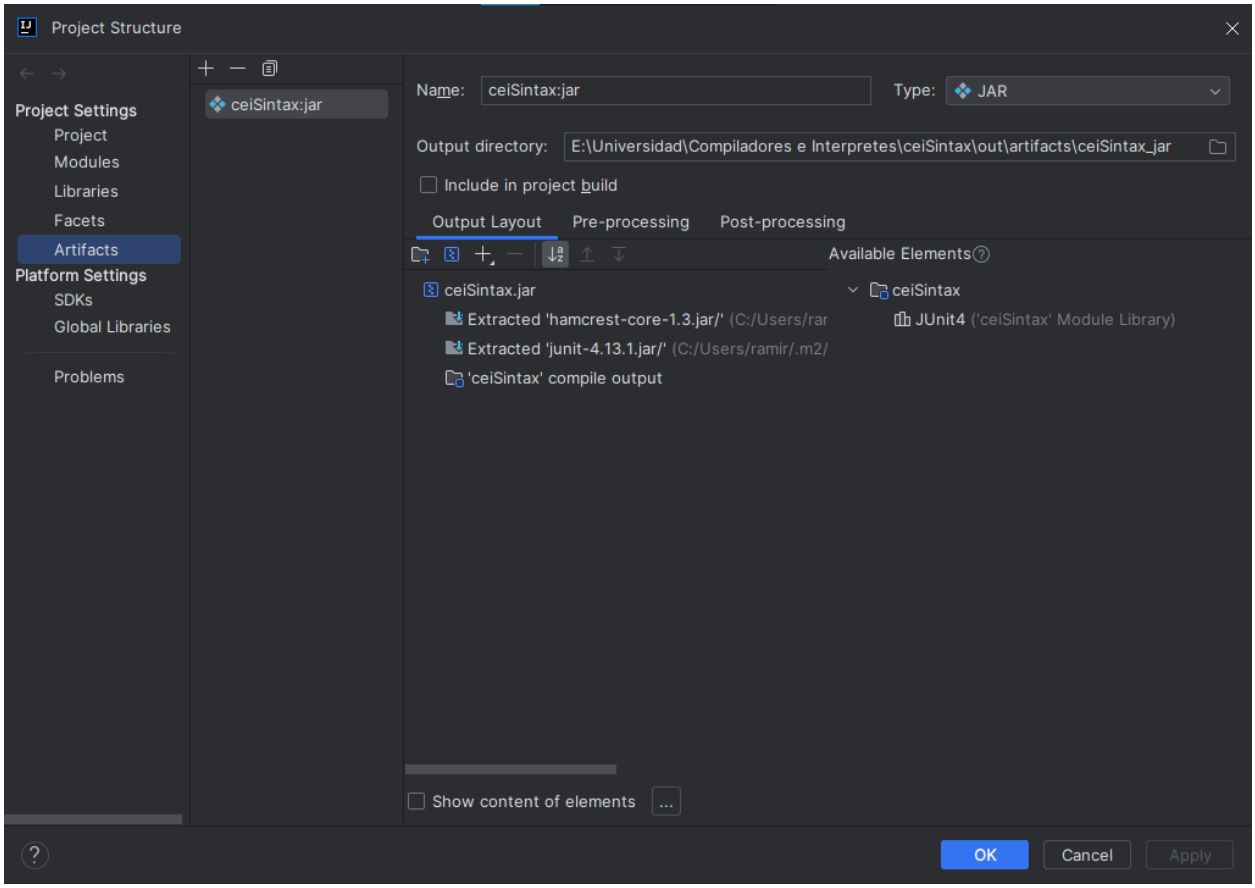
7. En la sección de Artifacts(1) vamos a presionar el botón **+**(2), luego en JAR(3) y por último en From modules with dependencies…(4).



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

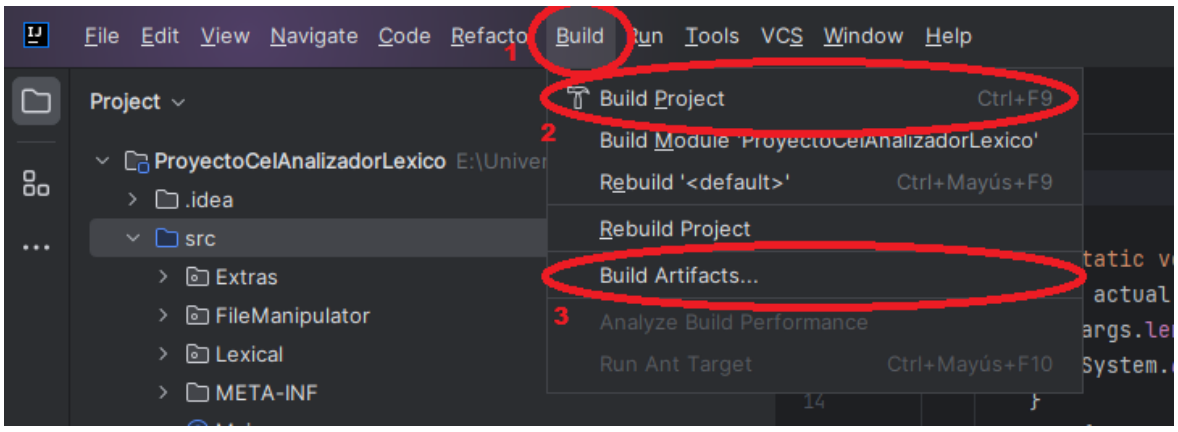


9. Nos debería quedar algo de este estilo, ponemos Apply y OK (si deseas podés cambiar el nombre del archivo que se va a generar donde dice Name:... , no le borres los dos puntos seguidos de jar).

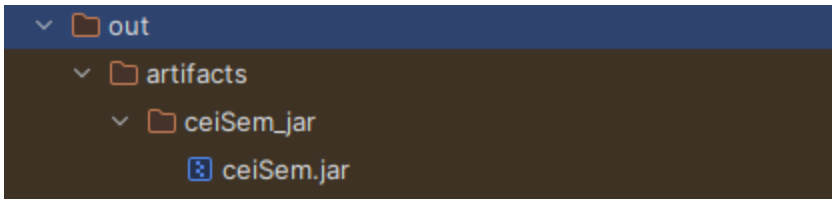


Con esto ya tenemos configurada la generación del JAR de nuestro proyecto.

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.

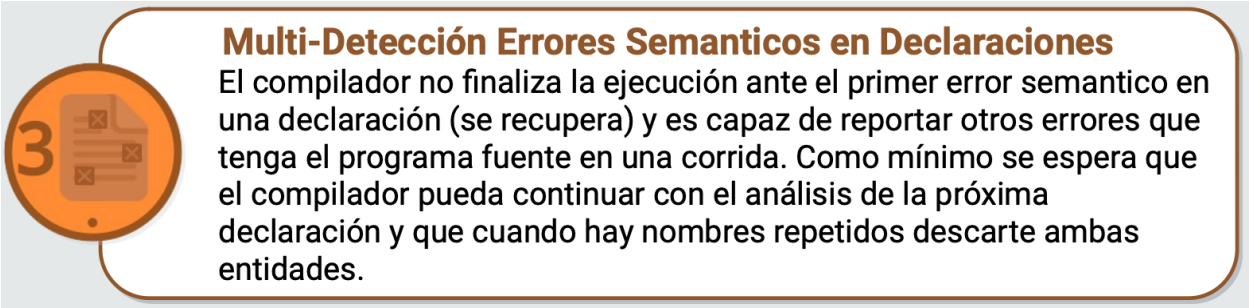
```
java -jar [Ruta de nuestro JAR] [Ruta del archivo a analizar]
```

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:

Logros de etapas anteriores (segundo round)



Aclaración: Se corrigieron las excepciones no detectadas en la etapa 3, para utilizarlo se deberá utilizar la version FirstSemantic del proyecto, y utilizar el archivo MainSem.

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.