

# Documentación

## Compiladores e Intérpretes

### ETAPA 5: Generador

Cejas Ramiro, LU: 120404

---

<b>Introducción</b>	<b>2</b>
<b>AST</b>	<b>3</b>
ACLARACIONES	8
<b>¿Cómo compilar y utilizar el generador de código?</b>	<b>9</b>
<b>Logros</b>	<b>15</b>
<b>Aclaraciones Generales</b>	<b>16</b>

---

## Introducción

El generador de código se especializa en la traducción de un lenguaje intermedio a otro, en este caso, va a hacer un pasaje desde una representación del código en AST como vimos en la etapa anterior, para luego llegar a “código máquina” interpretable por la máquina virtual de Mini-JAVA.

En este documento se verán los detalles del funcionamiento y la implementación de un generador de código 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.

## 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.  
  
Genera: Nada
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.  
  
Genera:
  - Primero genera el lado derecho.
  - Duplica la última entrada para asignaciones encadenadas.
  - Genera el lado izquierdo asignando el valor obtenido del izquierdo.
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.  
  
Genera:
  - Primero genera lado izquierdo (que se apila en el stack)

- Luego genera lado derecho (también se apila)
- Se realiza la operación correspondiente según el operador

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.

Genera:

- Primero genera expresion (que se apila en el stack)
- Se realiza la operación correspondiente según el operador

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 sobrescribiendo 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.

Genera:

- Primero genera la expresión.
- Luego almacena el resultado en el offset de la variable a declarar.

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.

Genera:

- Primero genera la condición.
- Genera un salto condicional al inicio del cuerpo del else.
- Genera el cuerpo del then
- Si tiene else entonces genera su código, sino genera NOP.

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.

Genera:

- Primero genera la etiqueta while\_in.
- Genera la condición.
- Genera el salto condicional a la etiqueta while\_out.
- Genera el cuerpo.
- Genera la etiqueta while\_out : NOP.

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 todas las variables locales allí visibles. Por último chequea todas las sentencias que contenga. No tiene tipo.

Genera:

- Primero genera RMEM X (siendo X la cantidad de variables locales del bloque).
- Por cada sentencia, la genera, y en caso que sea un acceso y no se use su retorno o bien que sea una asignación se hace POP del valor no usado.
- Se genera FMEM X (siendo X la cantidad de variables locales del bloque).

10. NodeLiteral: Representa un literal.

Almacena el token asociado al literal. Para chequearse simplemente setea su tipo al tipo del token.

Genera:

- Genera el literal correspondiente. (PUSH Lit).

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.

Genera:

- En caso que esté en un método con retorno void o bien sea un return de un constructor se genera la salida del RA (se libera espacio de las vars locales, se apunta al RA del llamador, y se libera el espacio de los parametros).

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.

Genera:

- Si es un método estático:

Si es encadenado de algo, entonces hace POP para eliminar el this.

Si el retorno del método no es de tipo VOID entonces reserva lugar.

Genera los argumentos.

Hace push del método a llamar.

Hace CALL.

- Si es un método dinámico:

Si no es encadenado de algo entonces hace LOAD 3 (la referencia this).  
 SWAP (para mantener la ref THIS al tope de la pila).  
 Si el retorno del método no es de tipo VOID entonces reserva lugar.  
 SWAP (para mantener la ref THIS al tope de la pila).  
 Genera los argumentos y por cada uno SWAP.  
 DUP para no perder this.  
 LOADREF 0 se carga la VT.  
 LOADREF X (siendo X el offset del método a llamar)  
 Hace CALL.

- Si es un atributo estático:

Si es encadenado de algo, entonces hace POP para eliminar el this.  
 PUSH de la etiqueta del atributo estático.  
 Si es lectura, LOADREF 0.  
 Si es escritura, SWAP y STOREFEF 0.

- Si es un atributo dinámico:

Si no es encadenado de algo entonces hace LOAD 3 (la referencia this).  
 Si es lectura, LOADREF X (siendo X el offset del atributo a llamar).  
 Si es escritura, SWAP y STOREFEF X (siendo X el offset del atributo).

- Si es parámetro o variable local:

Si es lectura, LOAD X (siendo X el offset del parámetro o var a llamar).  
 Si es escritura, STORE X (siendo X el offset del parámetro o var).

### 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.

- RMEM 1 (se guarda lugar para el retorno).
- PUSH X (siendo X la cantidad de lugares a reservar para el objeto).
- PUSH malloc y CALL.
- DUP y PUSH VT\_X (siendo X la clase a contruir).
- STOREREF 0 y DUP.
- Se generan los argumentos y por cada uno SWAP para mantener THIS.



---

- PUSH constructor y CALL.

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.

Genera:

- Si el retorno del método es distinto de VOID, RMEM 1.
- Se generan los argumentos.
- PUSH del método y CALL.

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.

Genera:

- LOAD 3.

## ACLARACIONES

Todos los accesos luego de generar su código le piden a su encadenado hijo, si es que tienen, que se genere.

---

## OFFSETS

Para la asignación de los offsets, se realizan previo a la generación.

### Atributos

Cada clase se encarga de asignar los offsets, teniendo en cuenta los atributos que hereda de su clase ancestro. Si lo sobrescribe su offset es heredado (posible problema para los atributos tapados).

### Métodos

Cada clase también asigna los offsets de los métodos teniendo en cuenta los métodos declarados en clases o interfaces ancestras, también considerando la posible redefinición, en cuyo caso hereda el offset.

### Parámetros

Cada método se encarga de asignar los offsets de los parámetros.

### Variables locales

Cada bloque se encarga de asignar los offsets de las variables locales teniendo en cuenta el offset de bloque padre.

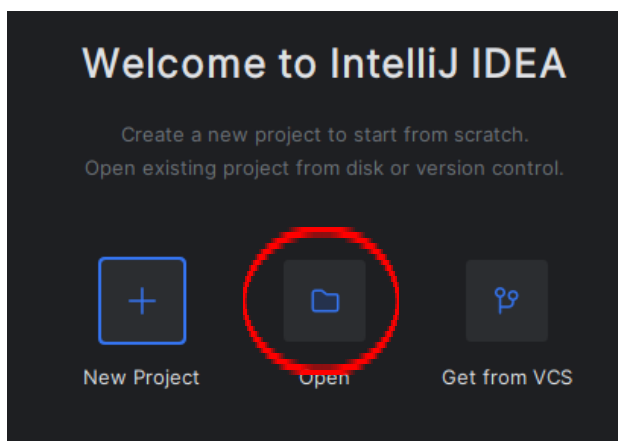
## ¿Cómo compilar y utilizar el generador de código?

Requisitos previos:

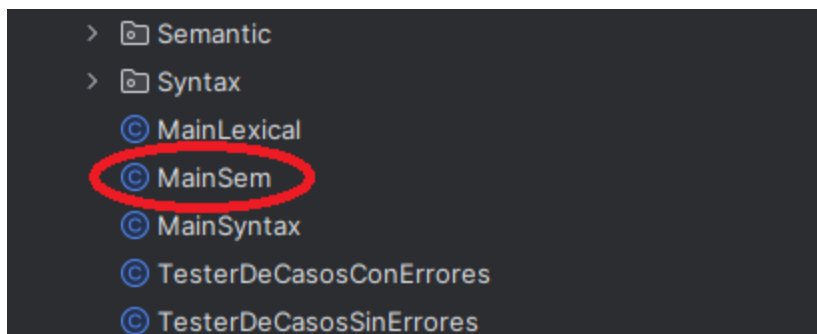
1. Instalar IntelliJ IDEA, se pueden usar cualquiera de las 2 variantes Ultimate o Community Edition, en este caso se utilizó la versión IntelliJ IDEA Community Edition 2023.2 (Podés descargarlo [acá](#)). ACLARACIÓN: El uso de este IDE no es restrictivo al funcionamiento del proyecto, pero la explicación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

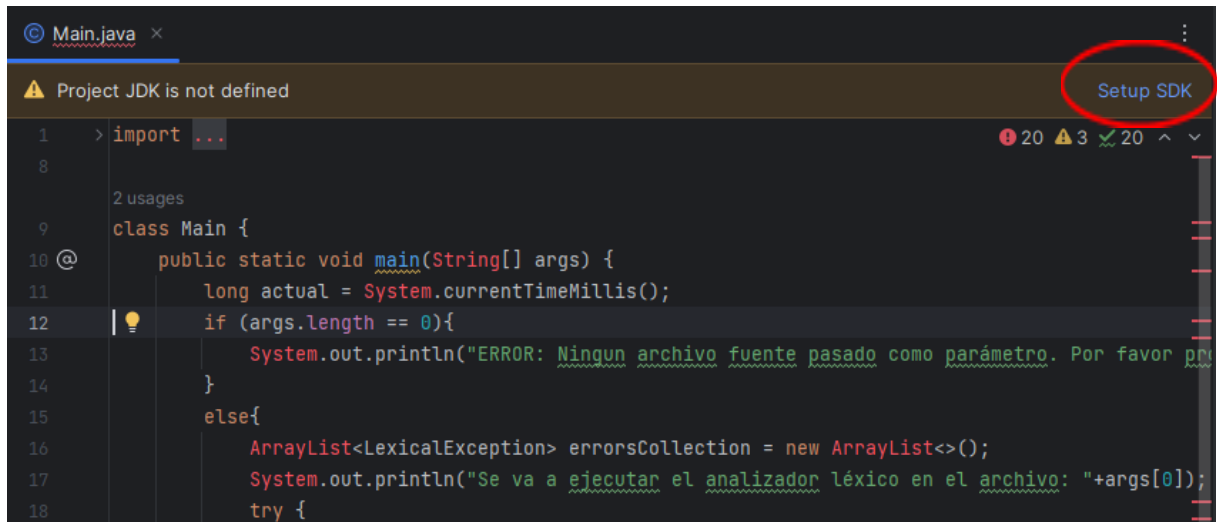
1. Abrimos el proyecto descargado del repositorio con IDEA.



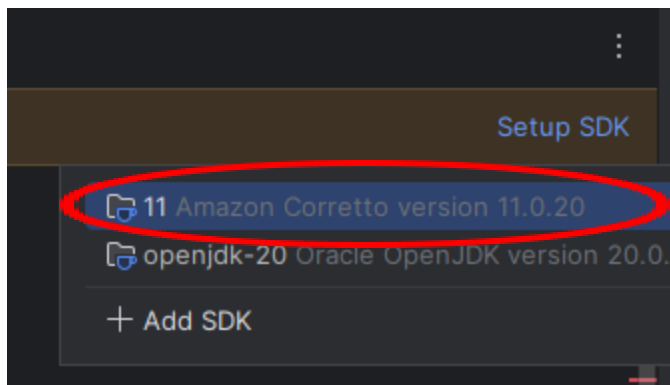
2. Veremos algo de este estilo, lo que tenemos que hacer es abrir el archivo Main.



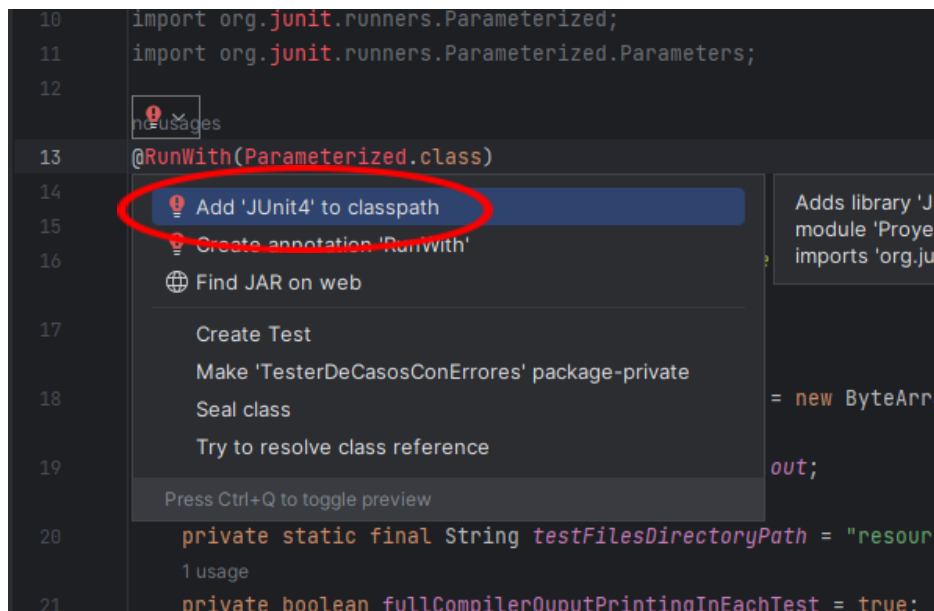
3. Nos va a decir que el proyecto no tiene ningún JDK establecido. Entonces damos click en Setup SDK.



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.

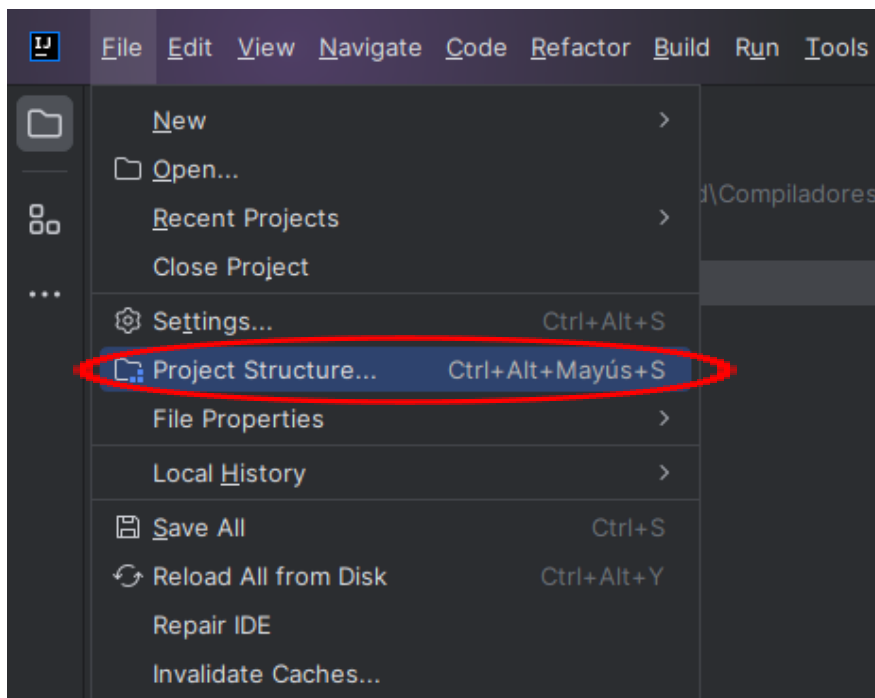


5. En este paso se realiza lo necesario para disponer de testers para probar la funcionalidad, en caso de no quererlos, eliminar los archivos TesterDeCasosSinErrores y TesterDeCasosConErrores, además eliminar el directorio resources y saltar al paso 6. En caso de querer tener los testers lo tenemos que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

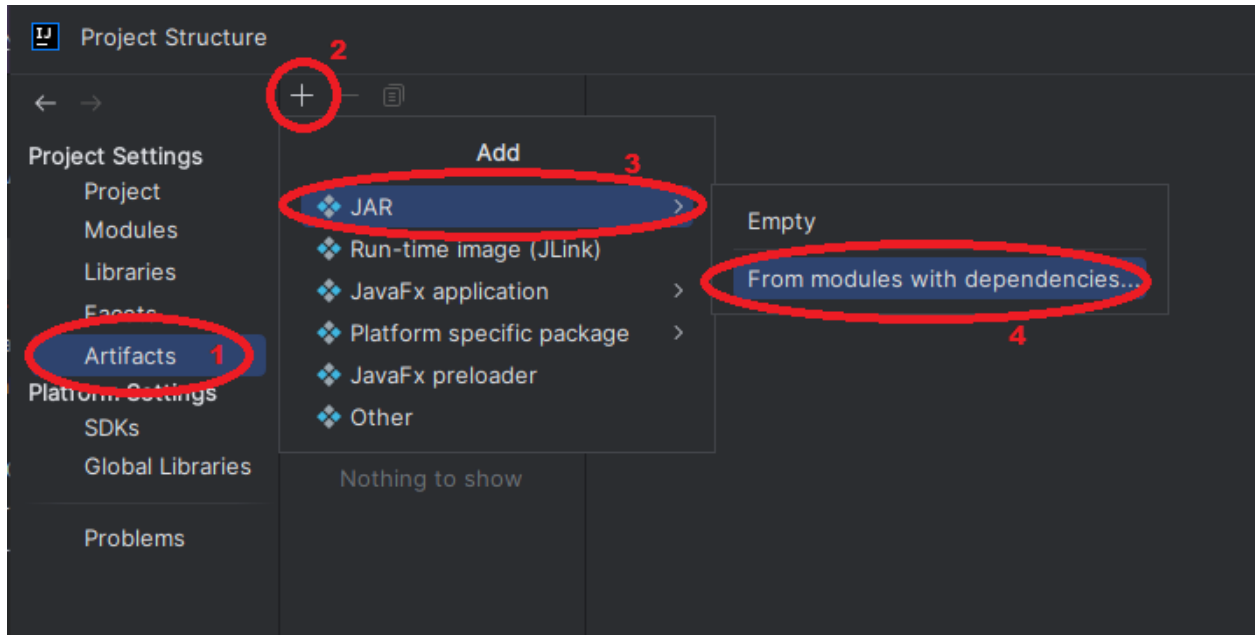


Ahora tenemos todo lo necesario para poder testear el código del analizador léxico y podemos ejecutar los test de JUnit. (Si nos sigue arrojando el error de JUnit, reiniciamos el IDE para que recargue el proyecto y se debería solucionar).

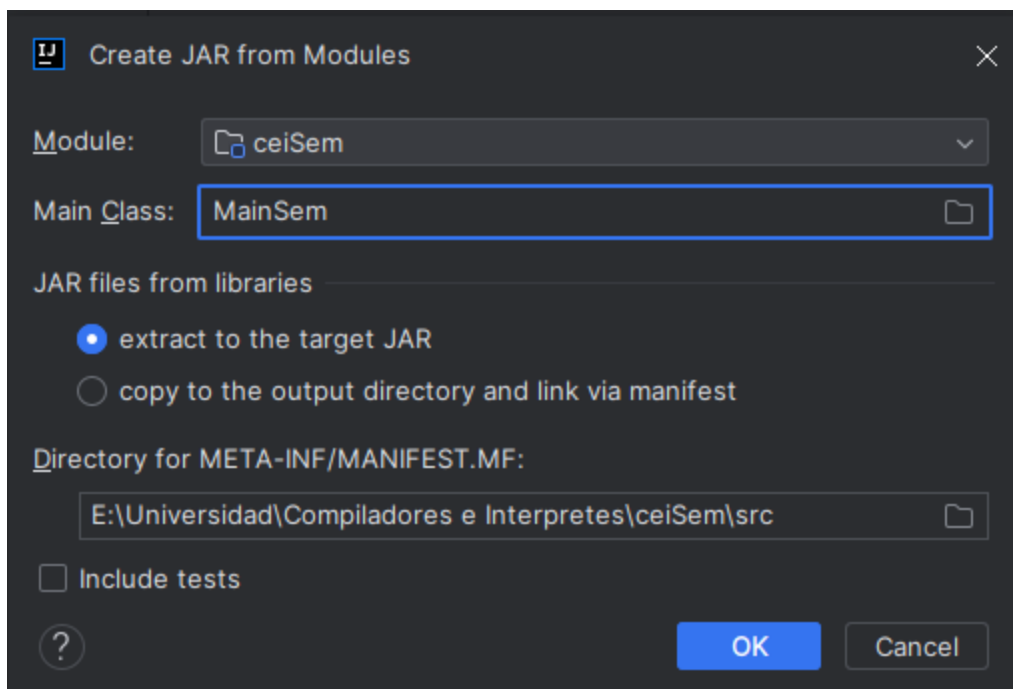
6. Para compilar el archivo principal de nuestro proyecto vamos a ir a File > Project Structure.



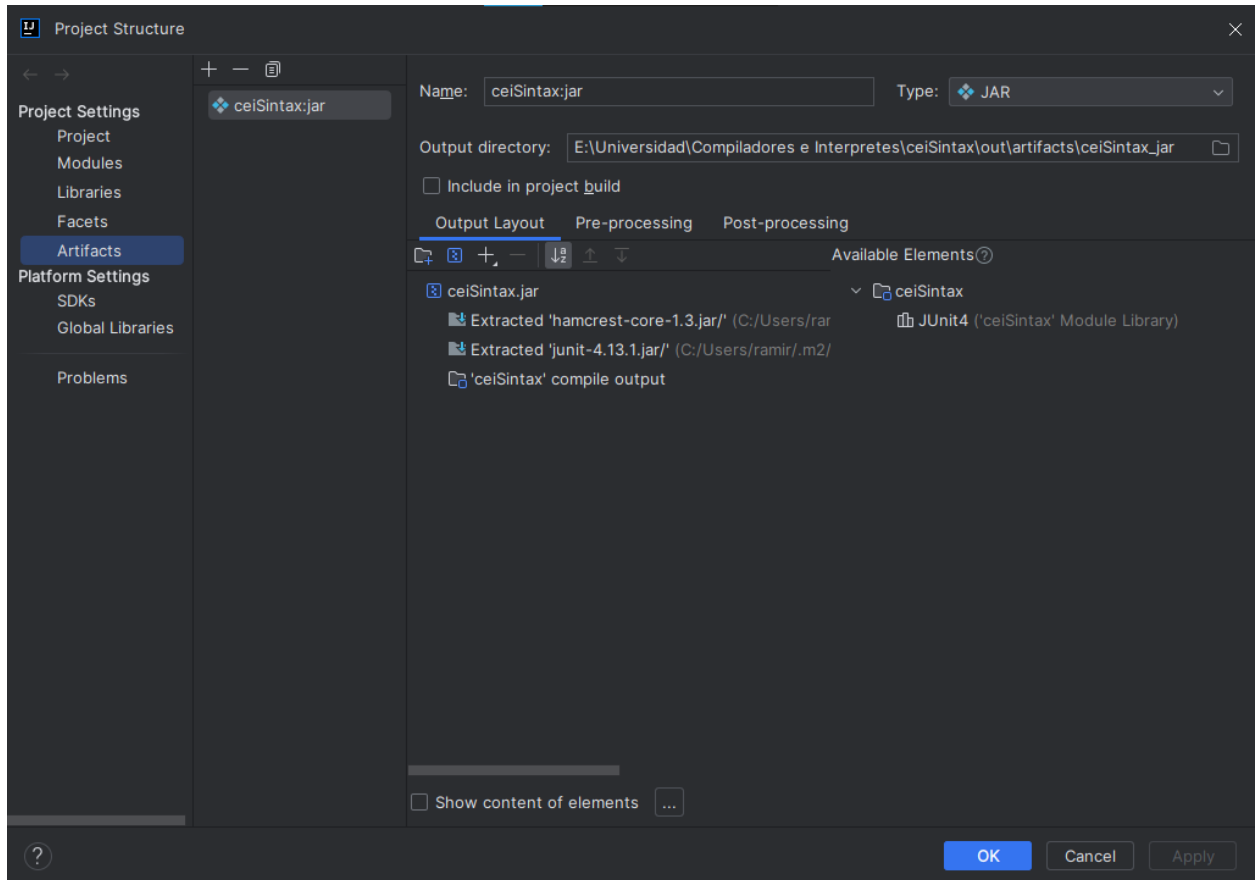
7. En la sección de Artifacts(1) vamos a presionar el botón +(2), luego en JAR(3) y por último en From modules with dependencies...(4).



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

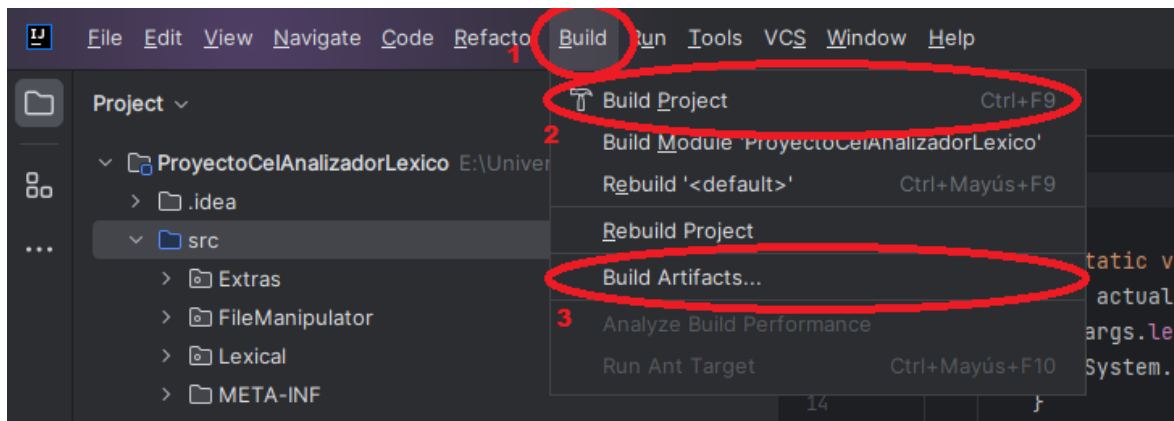


9. Nos debería quedar algo de este estilo, ponemos Apply y OK (si deseas podés cambiar el nombre del archivo que se va a generar donde dice Name:..., no le borres los dos puntos seguidos de jar).

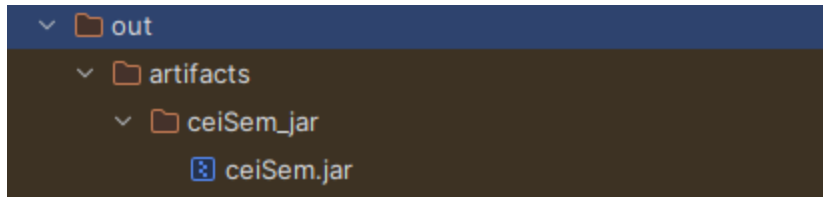


Con esto ya tenemos configurada la generación del JAR de nuestro proyecto.

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:



**Imbatibilidad Generación**  
*(no valido para la reentrega)*  
El software entregado pasa correctamente toda la batería de prueba utilizada por la cátedra



---

## 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.