



Documentación

Compiladores e Intérpretes

ETAPA 1: Analizador Léxico

Cejas Ramiro, LU: 120404

Introducción	2
Fundamentos del Analizador Léxico	3
Caracter	3
¿Lexema?	3
¿Token?	3
Características del analizador léxico implementado	4
Keywords	4
Expresiones Regulares	5
Implementación	6
El módulo principal	6
El manejador de archivo	6
El analizador léxico	6
Excepciones	7
¿Cómo compilar y utilizar el Analizador Léxico?	8
Logros	14
Aclaraciones generales	15

Introducción

Un analizador léxico es una parte fundamental en la compilación de programas. Su trabajo es tomar el código fuente escrito en un lenguaje de programación y dividirlo en unidades más pequeñas llamadas "tokens". Estos tokens son las piezas básicas del lenguaje, como palabras clave, identificadores, números y símbolos. Podríamos decir que es como el "traductor" inicial que descompone el código fuente en unidades comprensibles y prepara el camino para el resto del proceso de compilación.

En este documento se verán los detalles del funcionamiento y la implementación de un analizador léxico 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 Léxico

Caracter

Los **caracteres** son las unidades más pequeñas de un lenguaje de programación. Son las letras, números, símbolos y espacios en blanco que componen el texto del código fuente. En el análisis léxico, el analizador examina secuencialmente estos caracteres para identificar patrones que coincidan con los tokens del lenguaje. Cada caracter contribuye a la formación de **lexemas**.

¿Lexema?

Un **lexema** es una secuencia de caracteres contiguos en el código fuente que tiene un significado coherente y se identifica como una unidad reconocible por el analizador léxico. En otras palabras, un lexema es una palabra o una combinación de caracteres que representa un elemento específico del lenguaje de programación, como una palabra clave, un identificador, un número o un símbolo.

Por ejemplo, en el código `int numero = 42;` los lexemas serían:

"int" como una palabra clave

"numero" como un identificador

"=" como un símbolo de asignación

"42" como un número

El analizador léxico descompone el código fuente en lexemas y luego los clasifica en diferentes categorías de **tokens** para que el compilador pueda comprender y procesar el código de manera adecuada durante las etapas posteriores.

¿Token?

Un **token** es una unidad básica y significativa en un lenguaje de programación. Representa un componente individual del código fuente, como una palabra clave, un identificador, un número o un símbolo. Los tokens son las "palabras" fundamentales que conforman el lenguaje de programación y tienen un significado específico en la sintaxis y semántica del mismo.

El papel del token en el análisis léxico es identificar y clasificar las diferentes partes del código fuente para su posterior procesamiento. El analizador léxico examina el flujo de caracteres del código y reconoce patrones que corresponden a diferentes tipos de tokens. Cada vez que se encuentra un patrón válido, se crea un token asociado a ese patrón y se usa en etapas posteriores de la compilación.

Características del analizador léxico implementado

Hasta ahora se han mencionado elementos como caracteres, lexemas, tokens, pero no mencionamos cuales formarán parte de nuestro lenguaje Mini-JAVA. Veamos un rápido listado de los tokens que admitiremos:

Keywords

Las palabras clave, o "keywords" en inglés, son palabras reservadas en un lenguaje de programación que tienen un significado predefinido y se utilizan para propósitos específicos. Las palabras clave no pueden ser usadas como identificadores (como nombres de variables o funciones) porque ya están reservadas para construcciones específicas del lenguaje.

Las palabras clave que admitiremos y tendremos en nuestro Mini-JAVA serán:

class	interface	extends
implements	public	static
void	boolean	char
int	if	else
while	return	var
this	new	null
true	false	float

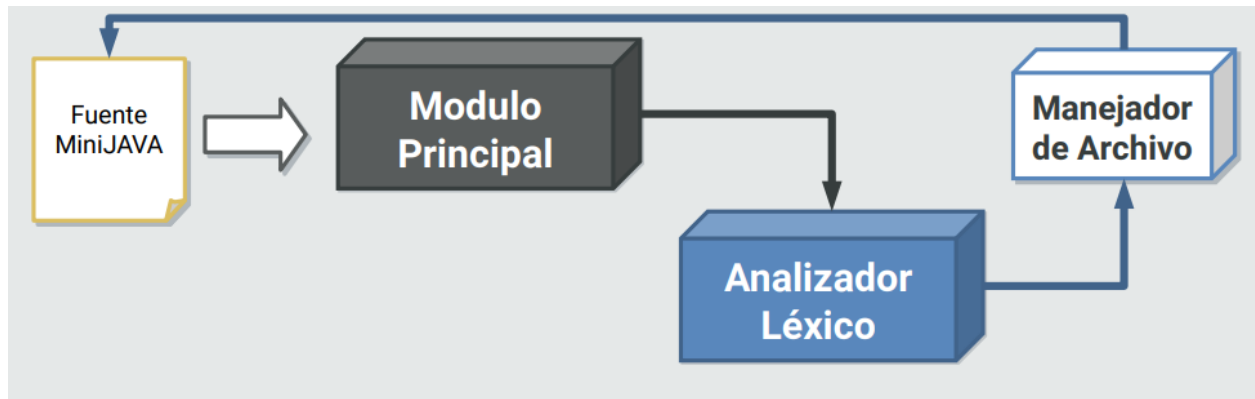
Expresiones Regulares

Las expresiones regulares se utilizan para identificar patrones específicos en el código fuente. Se utilizan para definir patrones de tokens, lo que ayuda al analizador léxico a reconocer palabras clave, identificadores, números y otros componentes.

Los patrones que modelamos en nuestro analizador léxico serán:

- Token = Literal | Identificador | Keyword | Puntuador | Operador | Asignacion
- Literal = Literal Entero | Literal Caracter | Literal String | Literal Booleano | Literal Nulo
- Literal Entero = `[0...9]*`
- Literal Caracter = `'X' | 'W'`
 - X = Cualquier caracter unicode que no sea el salto de línea `\n`, ni la comilla simple `'`
 - W = Cualquier caracter unicode que no sea el salto de línea `\n`
- Literal String = `"(Y | Z)*"`
 - Y = Cualquier caracter unicode que no sea el salto de línea `\n`, ni la comilla doble `"`
 - Z = `\`
- Literal Booleano = `true | false`
- Literal Nulo = `null`
- Keyword = Cualquiera de los keywords declarados en la tabla de arriba ([Ref](#))
- Puntuador = `(|) | { | } | ; | , | .`
- Operador = `> | < | ! | == | >= | <= | != | + | - | * | / | && | || | %`
- Asignacion = `+= | -= | =`

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.

En su método principal solicita tokens al Analizador Léxico hasta que éste devuelve el token EOF, además almacena todas las excepciones que se pueden producir y las muestra al finalizar la solicitud de los tokens.

El manejador de archivo

Fue realizado con el consejo de ["Chachi Piti"](#) el cuál orientó la utilización del Buffered Reader para leer archivos de texto.

Tiene un método getCharacter que lo que retorna el próximo caracter del archivo de texto, en caso de que el caracter a retornar sea un salto de línea el Manejador de Archivos aumenta internamente su contador de línea y resetea la columna actual a 0.

A su vez el manejador de archivo puede retornar no solo el número de línea actual sino también, el número de columna y el contenido de una determinada línea.

El analizador léxico

La implementación fue llevada a cabo mediante el mecanismo **Método x Estado**. Que utilizando llamadas encadenadas y representando los estados del autómata finito que reconoce las expresiones regulares antes declaradas ([Ref](#)) reconoce y retorna los Tokens o en su defecto una excepción léxica si se ha producido una.

Excepciones

Las excepciones fueron modeladas extendiendo la clase Exception de JAVA con una nueva clase denominada LexicalException, que almacena el lexema que generó la excepción, el número de línea donde está el lexema, el número de columna donde se detectó la excepción, una explicación de porqué se produjo y por último el contenido de la línea donde se detectó.

Los diferentes tipos de excepciones léxicas que se pueden producir son:

- No se permiten literales enteros de más de 9 dígitos.
- El literal es más pequeño que el menor float permitido.
- El literal es más grande que el mayor float permitido.
- El literal está fuera de los rangos permitidos por los floats.
- Caracter inválido.
- Se esperaba " pero se encontró EOF. Cuando no se cierra un String.
- Se esperaba " pero se encontró un salto de línea. Cuando un String no se cierra antes de un salto de línea.
- Se esperaba un caracter pero se encontró EOF. Cuando se abre la declaración de un char pero luego de la comilla simple se encuentra EOF.
- Se esperaba un caracter pero se encontró un salto de línea o un retorno de carro. Cuando se abre la declaración de un char pero luego de la comilla simple se encuentra con un salto de línea o un retorno de carro.
- Se esperaba ' pero se encontró EOF. Cuando ya se conoce el contenido del char pero no se encuentra una comilla simple de cierre, en cambio se encuentra EOF.
- Se esperaba '. Cuando ya se conoce el contenido del char pero no se encuentra una comilla simple de cierre.
- Se esperaba un dígito o un signo. En la declaración del exponente de un float se esperaba un signo (+ -) o bien un dígito.
- Se esperaba un dígito. En la declaración del exponente de un float, luego del signo, se espera un dígito.
- Se esperaba un andpersand doble.
- Se esperaba una línea vertical doble.
- Comentario multilínea no cerrado.

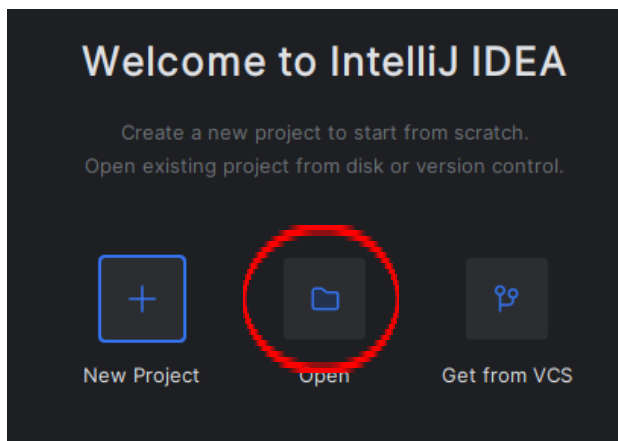
¿Cómo compilar y utilizar el Analizador Léxico?

Requisitos previos:

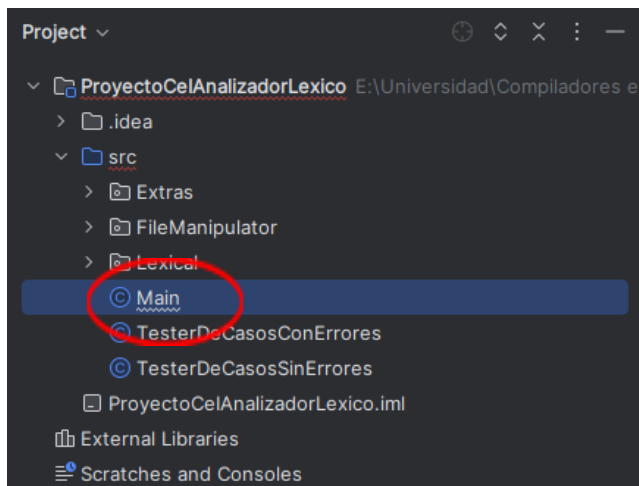
1. Instalar IntelliJ IDEA, se pueden usar cualquiera de las 2 variantes Ultimate o Community Edition, en este caso se utilizó la versión IntelliJ IDEA Community Edition 2023.2 (Podés descargarlo [acá](#)). ACLARACIÓN: El uso de este IDE no es restrictivo al funcionamiento del proyecto, pero la explicación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



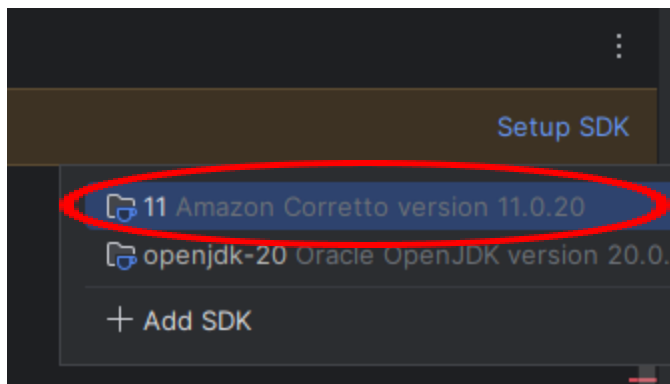
2. Veremos algo de este estilo, lo que tenemos que hacer es abrir el archivo Main.



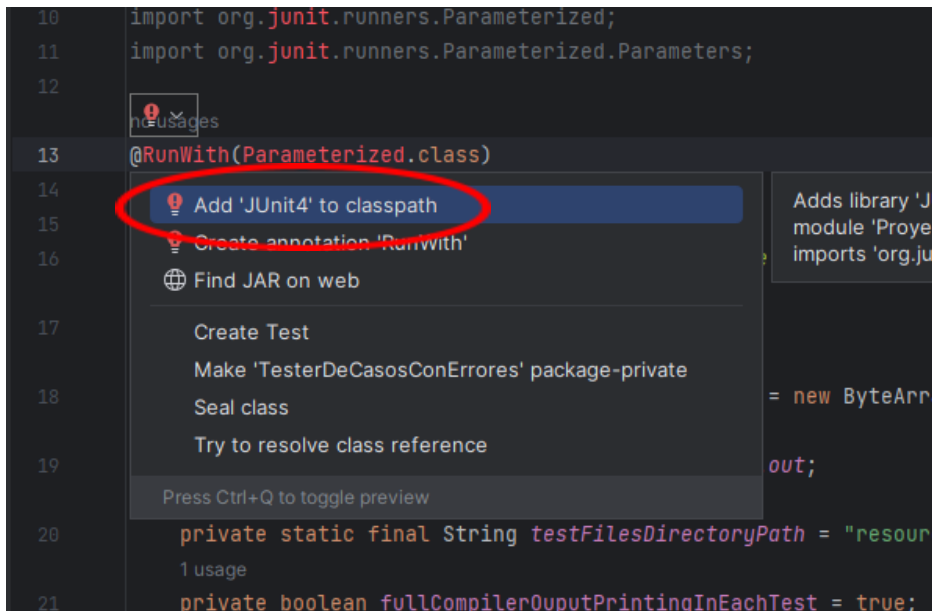
3. Nos va a decir que el proyecto no tiene ningún JDK establecido. Entonces damos click en Setup SDK.



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.

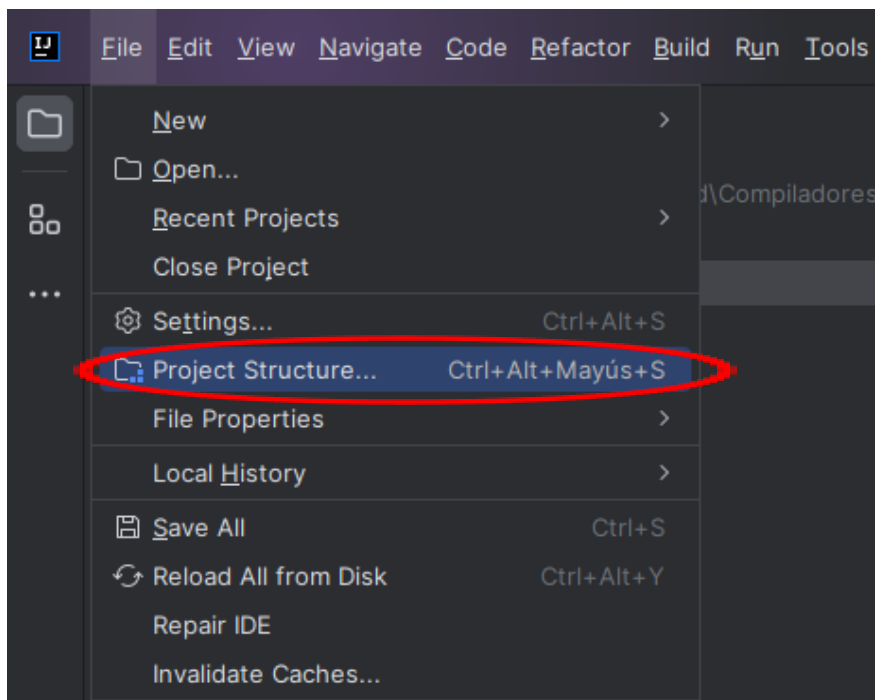


5. En este paso se realiza lo necesario para disponer de testers para probar la funcionalidad, en caso de no quererlos, eliminar los archivos TesterDeCasosSinErrores y TesterDeCasosConErrores, además eliminar el directorio resources y saltar al paso 6. En caso de querer tener los testers lo tenemos que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

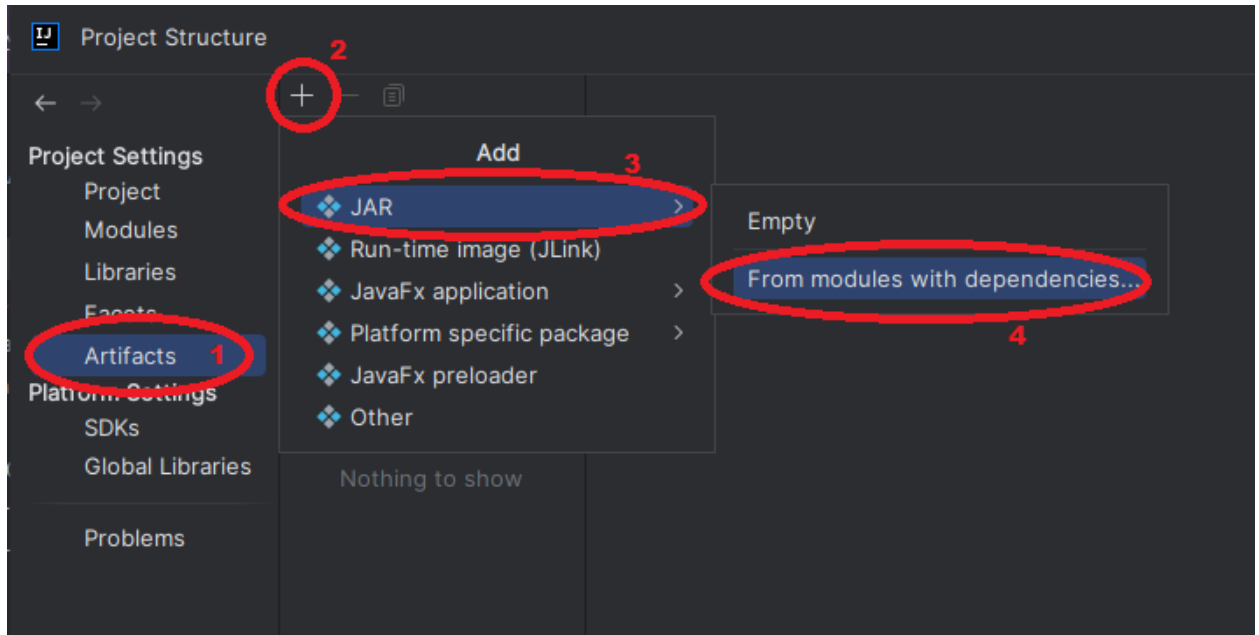


Ahora tenemos todo lo necesario para poder testear el código del analizador léxico y podemos ejecutar los test de JUnit. (Si nos sigue arrojando el error de JUnit, reiniciamos el IDE para que recargue el proyecto y se debería solucionar).

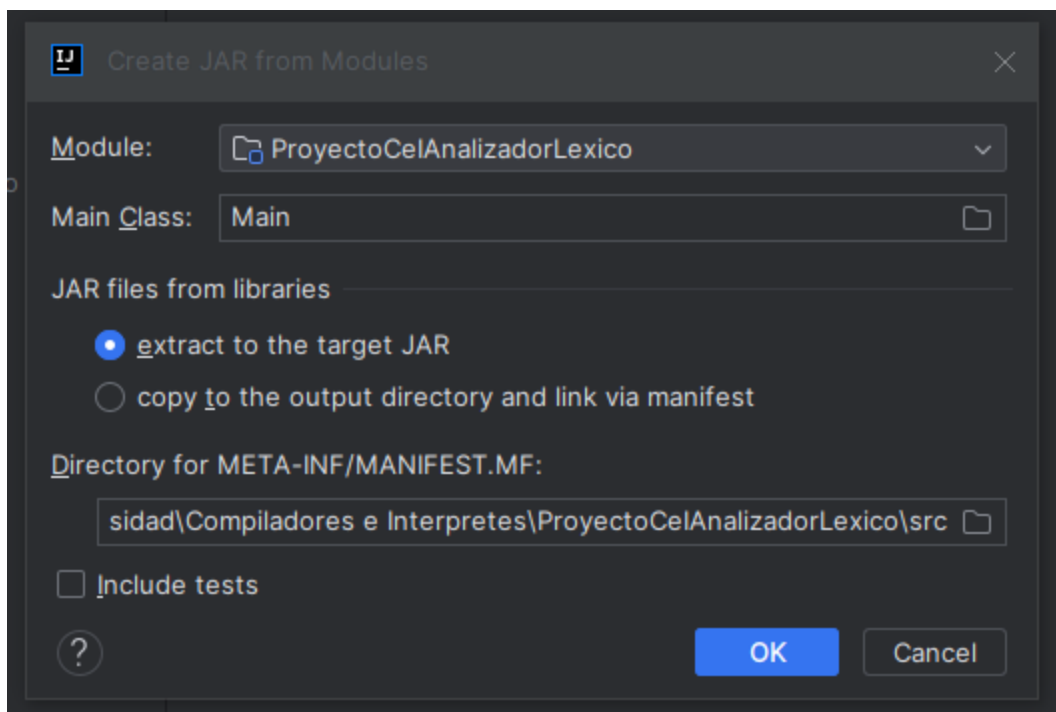
6. Para compilar el archivo principal de nuestro proyecto vamos a ir a File > Project Structure.



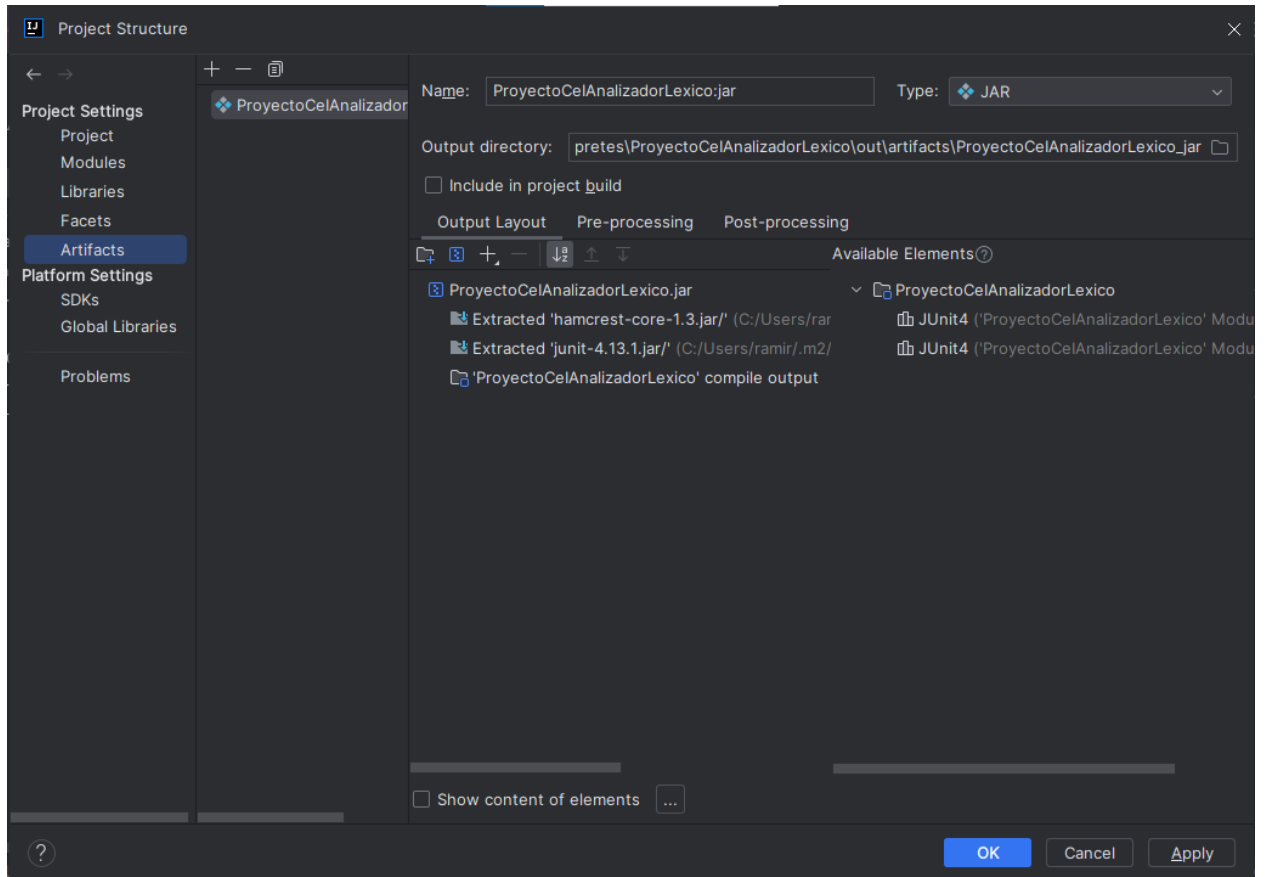
7. En la sección de Artifacts(1) vamos a presionar el botón +(2), luego en JAR(3) y por último en From modules with dependencies...(4).



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

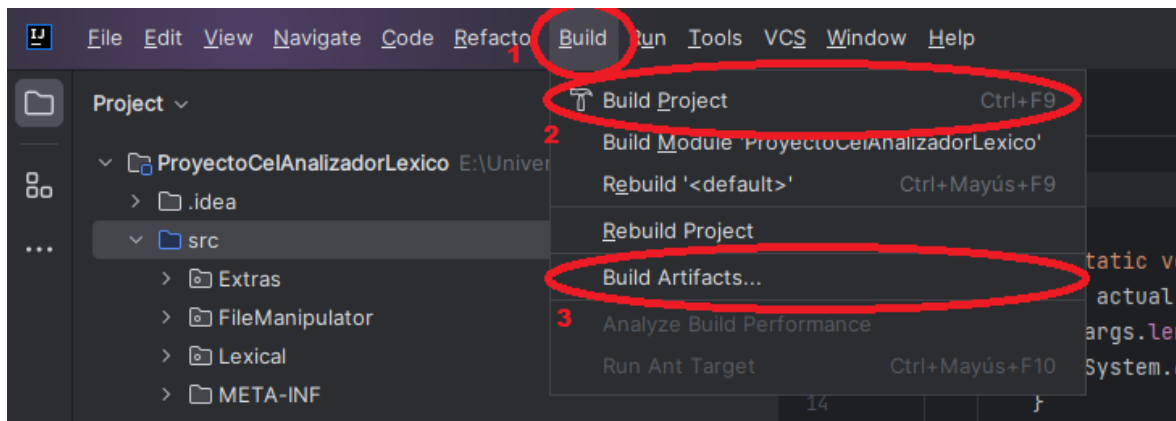


9. Nos debería quedar algo de este estilo, ponemos Apply y OK (si deseas podés cambiar el nombre del archivo que se va a generar donde dice Name:..., no le borres los dos puntos seguidos de jar).

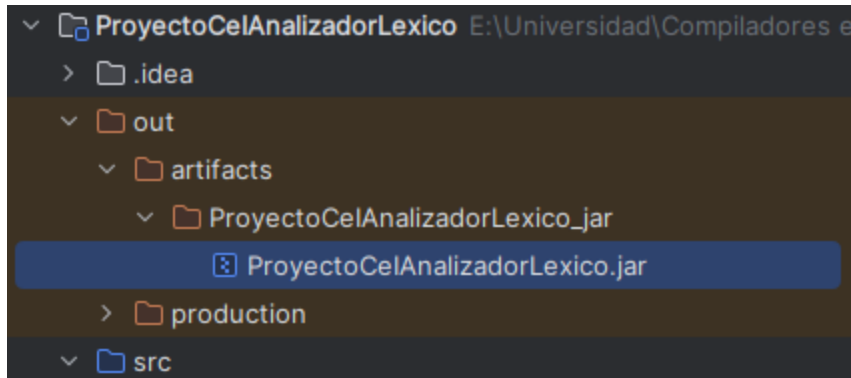


Con esto ya tenemos configurada la generación del JAR de nuestro proyecto.

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 realice el análisis léxico 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 léxico.

En este caso, estando posicionados en la raíz del proyecto y creando un archivo de prueba llamado `ArchivoPrueba.java` con el siguiente código:

```
public class Prueba {
}
```

un ejemplo de uso sería:

```
java -jar out/artifacts/ProyectoCeIAnalizadorLexico_jar/ProyectoCeIAnalizadorLexico.jar ArchivoPrueba.java
```

Y la salida esperada es:


```
Se va a ejecutar el analizador léxico en el archivo: ArchivoPrueba.java
(keyword_public,public,1)
(keyword_class,class,1)
(idClass,Prueba,1)
(punctuator_{,{,1)
(punctuator_},{,3)
(EOF,?,3)

-----


[SinErrores]
```

Logros


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



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




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



Floats!
 El Analizador Léxico permite literales float de Java. Estos literales tienen la misma estructura que en Java salvo que no permiten el sufijo de tipo (f o d) usados para distinguirlos de los doubles ya que no es necesario. El Léxico también debe reconocer la palabra reservada **float**
(Este logro esta asociado a otros logros en futuras entregas!)

ACLARACIÓN: Los floats fueron realizados según la estructura de Java, teniendo en cuenta su rango y sus diferentes sintaxis. Fuente: [docs.oracle](https://docs.oracle.com/javase/7/docs/api/java/lang/Float.html) y [JavaBNF](https://www.gnu.org/software/bnf/).



Reporte de Error Elegante
 El compilador cuando reporta un error, además del numero de linea y la razón del error, muestra la linea en cuestión y apunta al lugar donde se produjo. Por ejemplo para el programa de la derecha debería mostrarse el mensaje:
 Error Léxico en línea 2: # no es un símbolo valido
 Detalle: v1 + # chau
 ^
 [Error:#|2]

```
"hola"
v1 + # chau
if class}
```



Columnas

En los mensajes de error el compilador además del numero de linea indica el numero de columna donde se produjo el error. *Importante: esto afecta solo a los mensajes y no a los códigos de error!*



Multi-detección de Errores Lexicos

El compilador no finaliza la ejecución ante el primer error léxico (se recupera) y es capaz de reportar todos los errores léxico que tenga el programa fuente en una corrida

Aclaraciones generales

1. En ningún documento se detalla en profundidad cómo se adoptan los lexemas a la hora de reportar un error léxico, adopté la representación que usa Java.