Informe Etapa 3
Compiladores e Intérpretes

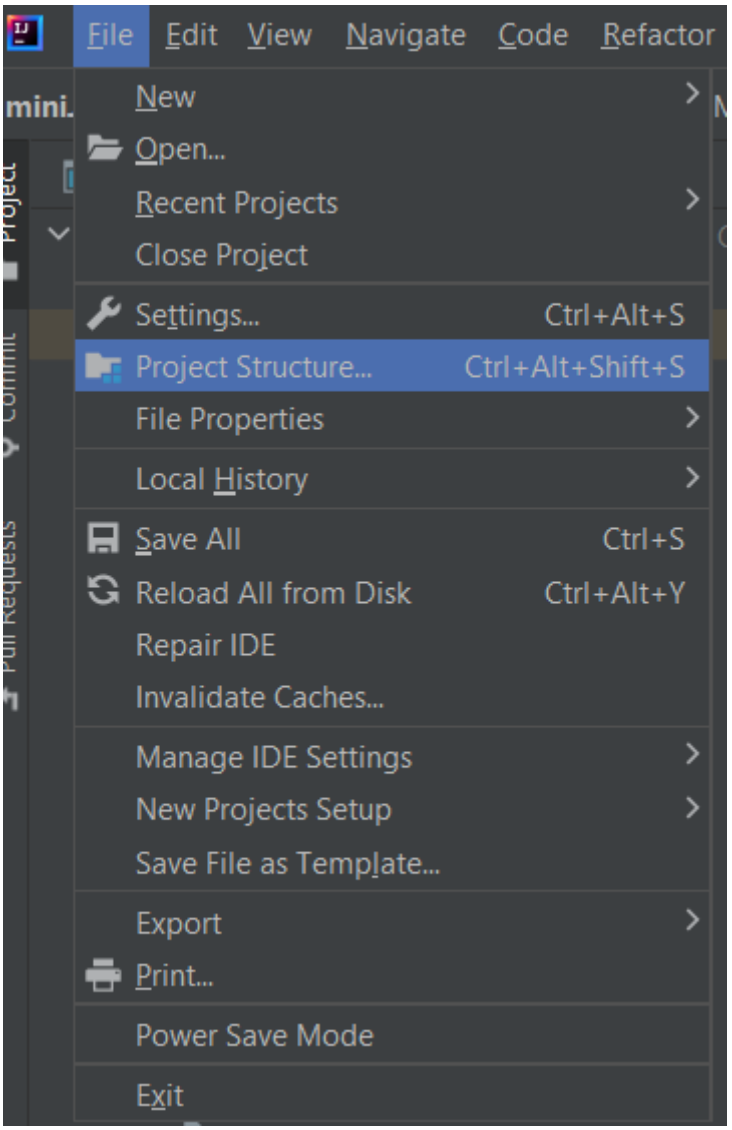
Índice

Modo de uso

Compilación

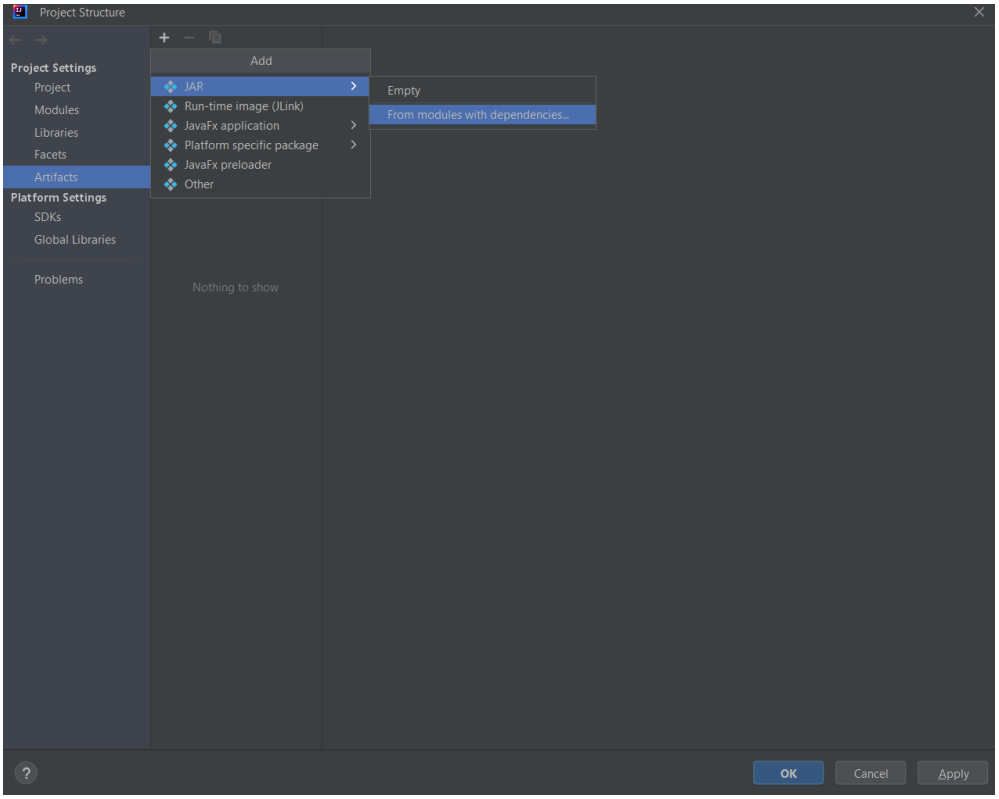
El proyecto fue desarrollado en InteliiJ y compilado con Java 11, para poder compilarlo es necesario usar la versión de Java mencionada y compilar los archivos fuentes en un ejecutable.

Esto se puede hacer desde IntelliJ de la siguiente manera:

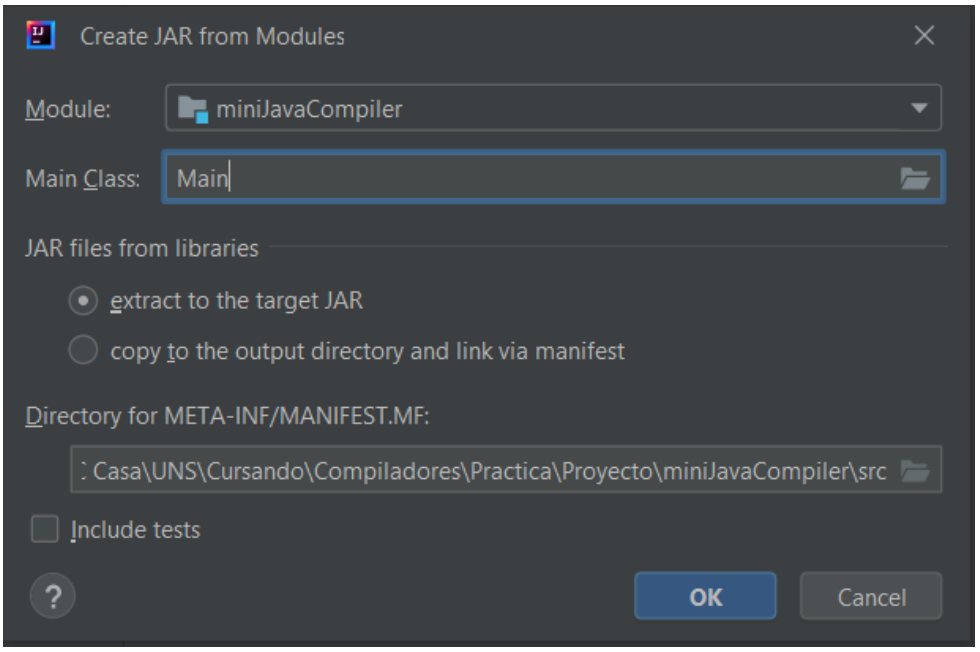


Primero creamos un nuevo proyecto con los archivos fuentes del proyecto, luego

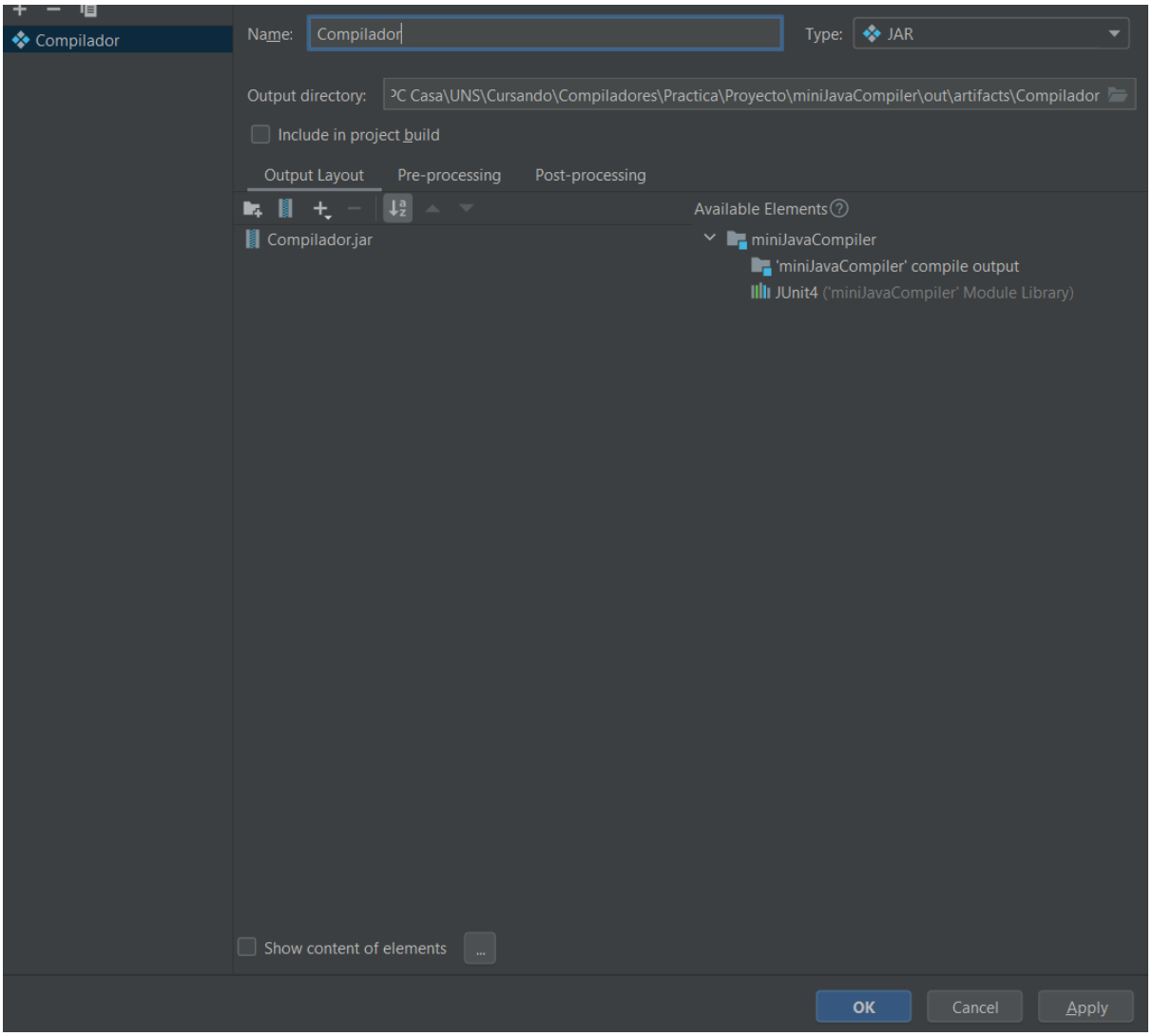
una vez construido, vamos a la estructura del proyecto



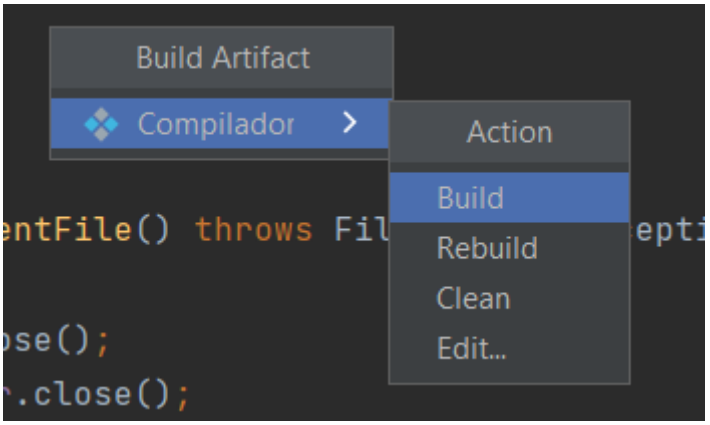
Aquí agregaremos un nuevo artifact que nos ayudará a generar el ejecutable JAR, recordar elegir la opción que crea el JAR basándose en módulos con dependencias



Nos pedira los módulos asociados a este ejecutable y allí seleccionamos el archivo Main del código fuente, luego de damos a OK, chequeamos el nombre y la dirección donde se generar el .jar y aplicamos



Ya preparamos todo para compilar, ahora solo hace falta crearlo, para ello vamos a Build Artifacts y creamos el artifact que acabamos de definir



Uso

Una vez con el ejecutable se puede utilizar el programa desde la terminal de comandos, en el lugar donde se encuentra el ejecutable del programa y los archivos a compilar se debe escribir :

```
java -jar miniJavaCompiler.jar programa.java
```

Siendo miniJavaCompiler.jar el ejecutable y programa.java el archivo a compilar.

Es muy importante recordar que ambos deben estar al mismo nivel para poder acceder de manera correcta, sino también se puede suministrar un camino hacia el archivo a compilar, siempre asumiendo que se parte de la ubicación actual en la que se encuentra la terminal, solo se le puede suministrar un archivo a compilar a la vez, el resto de argumentos serán ignorados.

Errores reconocibles

Sumado a los errores reconocibles del analizador léxico que eran :

- Símbolos que no pertenecen al alfabeto
- Comentarios multilínea sin cerrar
- Cadena de caracteres sin cerrar
- Operadores lógicos no válidos
- Floats que exceder el valor representable
- Floats que están mal formateados

Y los del el analizador sintáctico que son :

- Bloque de ejecución sin abrir o cerrar
- Línea sin terminar correctamente
- Extensión de múltiples clases
- Implementación de múltiples clases
- Anidamiento de parámetros genéricos
- Constructor estatico
- Variables locales con modificadores de visibilidad o estáticos
- Declaración de variables en condiciones de if o while
- Errores relacionados al anidamiento de llamados

Se les agrega los errores relacionados al análisis semántico, estos son :

- Herencia Circular
- Tipo de atributo, retorno o parámetro no declarado
- Faltante de método main
- Métodos con mismo nombre pero no redefinen
- No implementar la totalidad de una interfaz
- Atributos de tipo *void*
- Mas de un constructor declarado para una misma clase

Decisiones de diseño

Durante la modificación del analizador sintáctico se encontró con que la granularidad que se busco en la etapa anterior generó efectos adversos para esta implementación, ahora algunos métodos que no necesitaban parámetros ahora si los necesitas, también algunos ahora retornar valores que necesitamos para poder agregarlos a la tabla de símbolos.

Además se les redefinió a las clases de la tabla de símbolos los métodos *toString()* y *equals()* para que reflejen más fielmente lo que queríamos lograr, la única consideración importante fue que para los *ParameterST* el método *equals()* no significa que deben tener el mismo *Type* y nombre, sino que solo el mismo *Type*, esto porque en el contexto de comparar dos métodos no importa su identificador, solo tu tipo y el orden en el que aparecen.

Si bien se especifica que una clase no puede extender otra y a la vez implementar una interfaz este caso solo ocurre implícitamente si se implementa una interfaz, esto porque todas las clases heredan por defectos de *Object*.

Por último al no estar seguros de que marcar en algunos errores se tomar algunas convenciones como :

- En el caso de nombre repetido marcar el lexema y la línea que contiene la repetición, esto es, si se declaró una clase A en línea 3 y clase A en línea 5 entonces se marca esta última, esto aplica a todas las entidades.
- Si no se implementa un método de una interfaz se marca el método que no fue implementado, si bien este no genera el error perse Java lo que hace es mostrar toda línea de la clase, nosotros no podemos mostrar toda la linea en el codigo de error entonces simplemente marcamos el nombre del método que falta y donde se declaró.
- Si hay herencia circular se marca el nombre de la clase que primero se exploró para hallar la herencia, esto significa el más cerca del principio del archivo.
- Si existe algún método predefinido redefinido de manera incorrecta se marca la línea del método que está haciendo mal la redefinición.

Clases utilizadas

Se utilizaron en total 7 clases :

- **FileManager**, establece la interfaz de un manejador de texto
- **ImpFileManager**, implementa la interfaz de un manejador de texto
- **Token**, representa un *token* de un analizador léxico, con id de operación, lexema, etc
- **LexicalAnalyzer**, encargado de realizar el análisis léxico sobre el código
- **LexicalException**, una excepción que puede lanzar LexicalAnalyzer
- **SyntaxAnalyzer**, encargado de realizar el análisis sintáctico sobre el código, le va pidiendo tokens a LexicalAnalyzer
- **SyntaxException**, una excepción que puede lanzar SyntaxAnalyzer que informa de un error sintáctico.
- **SemanticException,** una excepción que puede lanzar cualquier entidad de la tabla de símbolos o el mismo *SyntaxAnalyzer*, informa de un error semántico.
- **Main**, encargada de mostrar los errores o ejecución exitosa, tomar los argumentos de consola , crear el manejador de archivo, analizador léxico y sintáctico.
- **MainStage1,** respectiva clase Main para ejecutar la entrega de la etapa 1.

- **MainStage2,** respectiva clase Main para ejecutar la entrega de la etapa 2.
- **SyntaxAnalyzerStage2**, la clase *SyntaxAnalyzer* perteneciente a la etapa 2.
- **Type**, clase abstracta que modela y define a los tipos del compilador.
- **ReferenceType**, hereda de *Type* y modela los tipos que son por referencia, ergo, tipo clases como *Object* o *String*.
- **PrimitiveType**, hereda de *Type* y modela los tipos primitivos , estos son *int, float, char*, etc.
- **EntityST**, interfaz que modela a grandes rasgos los métodos que usa una entidad que pertenece a la tabla de símbolos.
- **SymbolTable,** modela la tabla de símbolos que se usa durante el análisis semántico, tanto crearla como chequear que su contenido sea el correcto.
- **AttributeST,** modela la entidad atributo en la tabla de símbolos, tiene el nombre y el *Type* de un atributo.
- **ClassST,** modela la entidad clase en la tabla de símbolos, tiene nombre, clase que hereda o interfaz a implementar, etc.
- **InterfaceST,** modela la entidad interfaz en la tabla de símbolos, su nombre, si extiende a otra interfaz, que métodos declara, etc.
- **RoutineST,** clase abstracta que define que compone a una rutina o unidad en la tabla de símbolos, nombre, lista de parámetros, como se comparan, etc.
- **ConstructorST,** extiende a *RoutineST* y modela la entidad constructor de la tabla de símbolos.
- **MethodST ,** extiende a *RoutineST* y modela la entidad método en la tabla de símbolos, agrega el tipo de retorno que puede tener un método.
- **ParameterST,** modela la entidad parámetro en la tabla de símbolos , con su identificador y tipo.

Logros intentados

En la etapa actual , número 3, se intentó alcanzar los logros de :

- Imbatibilidad Semantica I
- Entrega Anticipada Semantica I
- Herencia Múltiple
- Atributos Tapados

De la etapa número 2 se corrigieron errores para poder alcanzar los logros :

- Implementación avanzada
- Genericidad

De la etapa 1 también se corrigieron errores relacionados algunos logros con el objetivo de conseguirlos, estos fueron :

- Columnas
- Reporte de Error Elegante

Ejecución de previas etapas

Existen múltiples clases con el nombre **Main** para poder ejecutar las entregas de las etapas previas, por ejemplo si se ejecuta el método *main* de la clase **MainStage2** entonces se ejecutara la entrega correspondiente a la etapa 2 del proyecto, en este caso era la del analizador sintáctico.

Usando la herramienta de testing provista por la cátedra esto es solamente cambiar el tipo de la constante *init* . Si se tiene en esta herramienta :

```
...
private static final Main init = null;
...
```

Se ejecutaría la batería de *tests* sobre la entrega actual, si se lo cambia por :

```
...
private static final MainStage2 init = null;
...
```

Se ejecutaría la batería de *tests* sobre la entrega de la etapa 2, así pudiendo probar los logros previos o correcciones, **MainStage2** corresponde a la clase **Main** de la etapa 2 y **MainStage1** corresponde a la clase **Main** de la etapa 1.

Otros cambios

Existía un error relacionado al analizador semántico debido a la mala especificación del operador *opMul*, esto fue corregido al igual que la genericidad en argumentos formales y constructores.

También existía un error en la declaración de la interfaz, teniendo *extiendeOpcional()* seguido de *herenciaOpcional()* algo no adecuado que nos percatamos durante la creación de la tabla de símbolos, este último método fue quitado de allí.

Para cumplir el logro de poder extender múltiples interfaces se tuvo que cambiar la gramática, antes solo se podía extender una interfaz o implementar una interfaz según las reglas sintácticas :

```
<ExtiendeOpcional> ::= extends idClase | ε
 <ImplementaA> ::= implements idClase | ε
```

Ahora se lo modifica usando la regla de producción definida en la etapa anterior como *<DeclaracionClasesOpcional>* podemos declarar que una interfaz extienda a muchas otras o que una clase implemente muchas interfaces, quedaría entonces como:

```
<ExtiendeOpcional> ::= extends idClase
   <DeclaracionClasesOpcional> | ε

 <ImplementaA> ::= implements idClase
   <DeclaracionClasesOpcional> | ε
```

Esto llevó a su modificación pertinente en el cálculo de los *firstSets* y *secondSets*, además se separó la regla de *<HerenciaOpcional>* porque no podíamos generar un *extend idClase implements idClase*, sino que era una o la otra, lo cambiamos para que pueda extender una clase e implementar varias con las reglas :

```
<ImplementaOpcional> ::= <ImplementaA> | ε
   <HeredaOpcional> ::= <HeredaDe> | ε
```

Estas fueron colocadas en lugar de *<HerenciaOpcional>,* también con sus respectivos *followSets*.