# Paso 0. Creación de la Estructura Inicial del Tutorial

## Resumen

En este primer capítulo se creará el proyecto con el código inicial del proyecto. El alumno no necesitará hacer ninguna tarea adicional en esta parte; simplemente copiará el código ya hecho y comprobará que funciona.

De esta manera se estará preparado para poder ya empezar a trabajar a partir del siguiente capítulo. Los capítulos posteriores consistirán en modificar y/o añadir fuentes a este proyecto.

## Contenido del Esqueleto del Traductor

La carpeta "Esqueleto Traductor" incluye clases Java que facilitan el arranque de la implementación de un traductor. Los fuentes del esqueleto se dividen en *dos tipos*:

- Aquellos que incluyen código rutinario que es *siempre igual* en todos los compiladores. Estos fuentes no serán modificados.
- Aquellos que están *vacíos* y su única función es indicar dónde se deberían ubicar dichos ficheros cuando se creen en la fase apropiada. Se pretende ayudar así a mantener una estructura adecuada en el compilador.

Este esqueleto no ha sido hecho expresamente para este tutorial, sino que puede ser utilizado independientemente de él. De hecho, se recomienda también su uso como punto de partida para la práctica del alumno.

## Tarea: Creación del Proyecto

El alumno deberá elegir una de las formas de crear este proyecto inicial:

- *Importar* el proyecto en el editor de su elección (eclipse, Visual Studio Code, …). En este caso, el JAR de ANTLR ya estará importado en el proyecto.
- Crear un nuevo proyecto y copiar los fuentes de la carpeta "*src*". En este caso, es importante recordar que hay que añadir al proyecto el JAR de ANTLR (que se encuentra en "antlr/antlr-4.7.2-complete.jar").

Para comprobar que todo funciona correctamente, ejecute la clase en *main/Main.java*. Debería aparecer el mensaje:

| "*El AST no ha sido creado*" |
|---|

Este mensaje es correcto, ya que esa parte no se ha implementado aún.

## Descripción del Código

### Funcionalidad

El código copiado implementa un traductor que solo admite como entrada válida un fichero de texto que tengan únicamente una constante entera. En el fichero *"parser/Grammar.g4"* se encuentra la siguiente gramática:

```
start: INT_CONSTANT EOF;
```

En el fichero *source.txt* se tiene una entrada válida:

```
24
```

La única razón de esta funcionalidad es tener un código inicial que compile sin errores y se pueda ejecutar, aunque no haga nada útil aún. Es más sencillo ir modificando un programa que ya compila sin errores que empezar con un proyecto completamente vacío.

## Clase Main

La clase Main dirige todo el proceso de compilación.

```
/**
 * Clase que inicia el compilador e invoca a todas sus fases.
 *
 * Normalmente, no será necesario modificar este fichero. En su lugar, modificar
 * los ficheros de cada fase (los cuales son llamados desde aquí):
 * - Para Análisis Léxico: 'Lexico.g4'.
 * - Para Análisis Sintáctico: 'Grammar.g4'.
 * - Para Análisis Semántico: 'Identification.java' y 'TypeChecking.java'.
 * - Para Generación de Código: 'MemoryAllocation.java' y 'CodeSelection.java'.
 */
public class Main {
  public static final String program = "source.txt"; // Fichero de prueba durante el desarrollo

  public static void main(String[] args) throws Exception {
    ErrorManager errorManager = new ErrorManager();

    AST ast = compile(program, errorManager); // Poner args[0] en vez de "programa" en la versión final
    if (errorManager.errorsCount() == 0)
      System.out.println("El programa se ha compilado correctamente.");

    ASTPrinter.toHtml(program, ast, "AST"); // Utilidad generada por VGen (opcional)
  }

  /**
   * Método que coordina todas las fases del compilador
   */
  public static AST compile(String sourceName, ErrorManager errorManager) throws Exception {

    // 1. Fases de Análisis Léxico y Sintáctico
    GrammarLexer lexer = new GrammarLexer(CharStreams.fromFileName(sourceName));
    GrammarParser parser = new GrammarParser(new CommonTokenStream(lexer));

    AST ast = null;

    // IMPORTANTE: Cuando se genere el AST, INTERCAMBIAR las dos líneas siguientes:
    parser.start();
    // ast = parser.start().ast;

    if (parser.getNumberOfSyntaxErrors() > 0 || ast == null) { // Hay errores o el AST no se ha implementado
      errorManager.notify("El AST no ha sido creado.");
      return null;
    }

    // 2. Fase de Análisis Semántico
    SemanticAnalisys analyzer = new SemanticAnalisys(errorManager);
    analyzer.analize(ast);
    if (errorManager.errorsCount() > 0)
      return ast;

    // 3. Fase de Generación de Código
    File sourceFile = new File(sourceName);
    Writer out = new FileWriter(new File(sourceFile.getParent(), "output.txt"));
```

```
    CodeGeneration generator = new CodeGeneration();
    generator.generate(sourceFile.getName(), ast, out);
    out.close();

    return ast;
  }
}
```

La clase *Main* no habrá que modificarla en todo el tutorial. Lo que harán los siguientes capítulos del tutorial es implementar algunas de las clases *que son invocadas desde aquí* y que actualmente están vacías. Dichas clases, aunque no hagan nada, se incluyen para indicar dónde irá cada funcionalidad en el futuro. Por ejemplo, a la hora de implementar la Fase de Identificación del analizador semántico, habrá que completar los métodos de la clase *"semantic/Identification.java"* (que ahora se encuentra vacía).

Se recomienda ahora mirar cada una de las clases del esqueleto (son bastante pequeñas y con comentarios que explican su función). En cualquier caso, se irá viendo su función en capítulos posteriores a medida que se vayan necesitando en cada fase.