



Universidad Nacional de Educación a Distancia
Departamento de Lenguajes y Sistemas Informáticos

Práctica de Procesadores del lenguaje I

Directrices de implementación

Dpto. de Lenguajes y Sistemas Informáticos
ETSI Informática, UNED

Alvaro Rodrigo
Andrés Duque
Anselmo Peñas (coordinador)

Curso 2021 - 2022

Contenido

1	INTRODUCCIÓN	3
2	MARCO DE TRABAJO Y DESARROLLO DE LA PRÁCTICA	3
2.1	LA ESTRUCTURA DE DIRECTORIOS	4
2.2	LA LIBRERÍA COMPILER - API	6
2.3	INSTALACIÓN	7
2.4	DESARROLLO	7
3	ENTREGA	9
3.1	ANÁLISIS LÉXICO	9
3.2	ANÁLISIS SINTÁCTICO	11
3.3	REALIZACIÓN DE PRUEBAS	12
3.3.1	<i>Pruebas del analizador léxico.....</i>	<i>12</i>
3.3.2	<i>Pruebas del analizador sintáctico</i>	<i>13</i>

1 Introducción

La construcción de un compilador es una compleja labor que requiere atender a numerosas cuestiones técnicas como la implementación de estructuras de datos orientadas a objetos, la organización de éstas en paquetes, el uso de diversos patrones de diseño, etc.

Para el desarrollo de la práctica de procesadores del lenguaje I se proporciona al alumno un marco de trabajo de referencia. Este marco de trabajo se proporciona con el siguiente fin:

- Reducir el tiempo de puesta en marcha del entorno necesario para el desarrollo de la práctica.
- Acelerar la adquisición de buenas prácticas en el desarrollo de un compilador
- Centrar el trabajo del alumno en el desarrollo de aquellos aspectos que inciden directamente con los contenidos propios de la asignatura.

En este documento se presenta el marco de trabajo software que se proporciona al alumno y se describe detalladamente el trabajo que el alumno debe realizar para superar satisfactoriamente la práctica. La corrección de la práctica se apoya en la organización software que prescribe este marco de trabajo y, por tanto, es normativo que el alumno respete completamente dicha organización. Cualquier violación a este respecto podría suponer un suspenso en el proceso de corrección.

Para la comprensión de este documento, se presupone que el alumno ha estudiado previamente la teoría de la asignatura y que por tanto está capacitado para entender de forma correcta las explicaciones y terminología ofrecida.

Las normas de entrega y división del trabajo deben consultarse en el enunciado de la práctica.

En caso de correcciones sobre el marco de trabajo, éstas se anunciarán en el Tablón de Anuncios de la asignatura. *Es necesario por tanto que el estudiante visite con asiduidad los foros de la asignatura.*

2 Marco de trabajo y desarrollo de la práctica

El desarrollo de la práctica se realiza sobre un marco de trabajo tecnológico y por tanto constituye un recurso necesario para el desarrollo de la misma. El marco de trabajo se entrega como un fichero comprimido que proporciona dos elementos fundamentales:

- **Una estructura de directorios para que el alumno organice su código.** Estos directorios contiene ficheros fuente en java con código de andamiaje que el alumno deberá completar y / o modificar para implementar las distintas funcionalidades del compilador. Estas clases están organizadas jerárquicamente en paquetes siendo **compiler** el paquete raíz. Por tanto, cualquier clase dentro del paquete compiler (o alguno de sus paquetes hijos) es una clase abierta que el alumno puede modificar. Consulte la API en el marco de trabajo y una descripción más detallada en la sección 2.1 para obtener más información.

- **Una librería en java que proporciona clases de soporte para el desarrollo del compilador.** Todas las clases abiertas del paquete compiler se apoyan en clases definidas dentro de la librería compiler-api.jar. Esta librería contiene, básicamente, clases de ayuda, clases abstractas e interfaces que utilizan, extienden o implementan la mayoría de las clases del paquete compiler (y sus paquetes hijos). Se trata de una familia de clases cerradas y compiladas por lo que el alumno no dispone de su código fuente para alterarlas ya que NO debe ni puede modificarlas. Todas las clases de soporte, dentro de la librería están organizadas jerárquicamente en paquetes siendo **es.uned.lsi.compiler** el nombre del paquete padre. Por tanto toda clase dentro de éste paquete (o alguno de sus paquetes hijos) es una clase de soporte proporcionada por el equipo docente que no debe modificarse. Consulte la API en el marco de trabajo o el resto del documento para obtener una información más detallada.

Tanto la estructura de directorios como las implementaciones y clases de referencia están en inglés. Se presupone, por tanto, que el alumno está familiarizado con la terminología inglesa a la hora de programar. No es necesario que el alumno complete la práctica en inglés, pero no ha de traducir el código proporcionado.

Además, debe quedar claro que la práctica se desarrolla en Java y dentro del paradigma de orientación a objetos. Existe una amplia literatura sobre este lenguaje de programación cómo referencias podemos destacar <http://java.sun.com/> y <http://www.javahispano.org/>. En esta última se puede encontrar documentación en castellano.

2.1 La estructura de directorios

A continuación se procede a explicar la estructura de los directorios del marco de trabajo que se utilizan para la realización de esta práctica, deteniéndonos en profundidad en cada uno de los paquetes donde debe almacenarse el código. El marco de trabajo completo contiene más directorios de los que se explican en este documento y que sirven para apoyar fases posteriores de un compilador de las que se tratan en esta práctica (analizador léxico y sintáctico), por lo que se ha decidido omitir dicha información¹. Aunque no se utilice el resto de directorios, de ningún modo debe de alterarse su estructura ni eliminarse nada de ellos.

- **src.** En este directorio debe almacenarse todo el código fuente del compilador. Las clases están organizadas en diferentes paquetes. El paquete principal se llama 'compiler' y dentro existen otros paquetes: uno por cada una de las fases conceptuales que constituyen el compilador. En este directorio debe almacenarse todo el código fuente del compilador organizado en paquetes tal y como describiremos a continuación. El contenido de este directorio contiene algunas clases de soporte de las que el alumno debe partir para el desarrollo de la práctica.
 - **/compiler/lexical.** Este directorio debe almacenar las clases del paquete 'compiler.lexical' que implementan el scanner de la práctica y todas sus clases

¹ El resto de fases contempladas por el marco de trabajo se tratan en la asignatura Procesadores del Lenguaje II

asociadas (incluido scanner.java). En este paquete se incluye la clase Token.java que el alumno debe utilizar y puede completar y/o extender para realizar el analizador léxico tal y como se describirá más adelante.

- **/compiler/syntax.** Este directorio debe almacenar todas las clases del paquete 'compiler.syntax' que implementan el parser y todas sus clases asociadas (incluidas parser.java y sym.java).
- **/compiler/test.** En este directorio se encuentran las clases llamadas LexicalTestCase.java y SyntaxTestCase.java que sirven para probar el analizador léxico y sintáctico, y la clase FinalTestCase.java que sirve para probar el funcionamiento final de un compilador más allá de lo que se pide en esta práctica, por lo que esta última no se utilizará. Más adelante se detallará cómo ejecutar y probar la práctica.
- **classes.** En este directorio se almacenarán las clases (archivos de extensión '.class') que se generan mediante el proceso de compilación del código fuente del compilador. El alumno no debe modificar este directorio, ya que la herramienta Ant, que presentaremos más adelante, se encarga de automatizar las tareas de compilación. Es importante que los ficheros '.class' se sitúen en este directorio y no en otros creados automáticamente por algunos IDEs, como por ejemplo el directorio bin.
- **lib.** Este directorio contiene las librerías necesarias para la realización de la práctica (archivos con extensión '.jar'). Aquí se incluyen las librerías *jflex.jar* y *cup.jar* necesarias para llevar a cabo el proceso de generación del scanner y el parser respectivamente, y *compiler-api.jar* que contiene las clases e interfaces necesarias para el desarrollo del compilador. El contenido de esta última librería será discutido en detalle más adelante. Todas las librerías bajo el directorio /lib NO deben eliminarse bajo ningún concepto. Aunque no es necesario, si el alumno considera oportuno hacer uso de otras librerías de apoyo debe incluirlas en este directorio e indicarlo en la memoria.
- **doc.** Este directorio contiene diferentes tipos de documentos necesarios para la realización y documentación de la práctica organizados de acuerdo a una serie de subdirectorios que se explican a continuación:
 - **/api.** En este directorio se almacena una completa descripción de las clases (el API) que constituyen el marco de trabajo incluido en la librería 'compiler-api.jar'. El formato de la documentación responde al estándar 'javadoc' definido por Sun. Para visualizarlo en forma de página Web ha de accederse a `index.html`.
 - **/config.** En este directorio se proporcionan el fichero de configuración de la aplicación: 'build.xml' que incluye las descripción de las tareas Ant para realizar la práctica (véase más adelante una descripción en profundidad de este documento). Nótese no obstante, que el alumno no requiere comprender el contenido de este fichero sino solamente saber utilizarlo con el uso de la herramienta Ant.

- **/memoria.** Aquí debe incluirse la memoria de la práctica según las normas explicadas en el documento del enunciado de la práctica.
- **/specs.** Este directorio incluye los ficheros para generar el analizador léxico o scanner (scanner.flex) y el analizador sintáctico o parser (parser.cup) que deben ser completados por el alumno. Para un correcto funcionamiento NO se debe cambiar su ubicación a otro directorio.
- **/test.** Contiene los ficheros de prueba para probar el compilador, desde la tarea anterior correspondiente (ver más adelante). Cualquier fichero fuente de prueba que desee utilizar el alumno deberá almacenarse en este directorio.

2.2 La librería compiler - api

La librería de referencia compiler-api.jar contiene una familia de clases organizadas en diferentes paquetes que deben ser utilizados por el alumno para implementar el compilador. En el directorio /doc/api puede encontrar una completa descripción técnica del contenido de este fichero jar. No obstante, a continuación describimos el uso, utilidad y organización en paquetes de cada una de las clases incluidas que afectan a esta práctica:

- Paquete **es.uned.lsi.compiler.lexical**. Este paquete contiene las clases que el alumno debe utilizar para desarrollar el analizador léxico de su compilador. En concreto podemos destacar las siguientes clases:
 - La clase **LexicalError**. Esta clase es una excepción de java (hereda de java.lang.Exception) que encapsula toda la información relativa a un error ocurrido durante el proceso de análisis léxico. Esencialmente incorpora 3 atributos miembro con sus correspondientes métodos de acceso get/set: line, el número de línea donde se encontró el error; column, el número de columna donde se encontró el error y lexema, el lexema que produjo dicho error. Esta clase debe construirse y emitirse como un mensaje de error léxico por la salida estándar mediante el uso de la clase LexicalErrorManager tal y como se ilustrará más adelante.
 - La clase **LexicalErrorManager**. Esta clase es un gestor de errores que permite generar automáticamente y emitir los mensajes de errores léxicos que encuentra el scanner durante el análisis de un fichero de código fuente mediante el uso de la función lexicalError. Por ejemplo la llegada de un carácter inválido o la ausencia de concordancia del lexema a la entrada con ningún patrón léxico definido en la especificación JFlex. Si el error es irreparable debe utilizarse el método lexicalFatalError, que provoca la terminación del proceso de análisis léxico. Además esta clase puede utilizarse para realizar trazas de información y depurado que informen del avance en el proceso de análisis léxico. En concreto se proporcionan 3 tipos de métodos: lexicalInfo, de uso opcional para emitir mensajes de información por la salida estándar; lexicalDebug de uso opcional para emitir mensajes de depuración por la salida estándar, lexicalError para emitir mensajes de error por la salida estándar y lexicalFatalError, que emite mensajes de error y paraliza la ejecución del compilador.

- Paquete **es.uned.lsi.compiler.syntax**. Este paquete contiene las clases que el alumno debe utilizar para desarrollar el analizador sintáctico de su compilador. A continuación se describe su clase principal:
 - La Clase **SyntaxErrorManager**. Esta clase persigue un objetivo similar al de la clase `LexicalErrorManager` con la diferencia de que su contexto de uso es, en este caso la especificación gramatical de Cup. Mas delante de describirá el uso que puede hacerse de la misma en el fichero `parser.cup`. De momento destacamos la existencia de los siguientes tipos de métodos: `syntaxInfo`, de uso opcional para emitir mensajes de información por la salida estándar; `syntaxDebug`, de uso opcional, para emitir mensajes de traza por la salida estándar que asistan en las tareas de depuración; `syntaxError` para emitir mensajes de error recuperables por la salida estándar y `syntaxFatalError` de uso obligatorio para emitir mensajes de error irrecuperables por la salida estándar

2.3 Instalación

La instalación del marco de trabajo es un proceso sencillo que puede resumirse en la secuencia de pasos que exponemos a continuación:

- Entre al entorno virtual
- Acceda al apartado correspondiente a la práctica (tareas)
- Descargue el fichero `Arquitectura.zip`
- Cree una carpeta para albergar el desarrollo del proyecto, por ejemplo `PL1`.
- Descomprima el fichero `Arquitectura.zip` bajo ese directorio

Si desea utilizar algún entorno de desarrollo integrado puede hacerlo. En ese caso debería crear un nuevo proyecto y ubicar allí la estructura de directorios del fichero `Arquitectura.zip` asegurándose de indicar que el directorio `/src` contiene los fuentes del proyecto y que la versión compilada de los mismos deberá redirigirse al directorio `/classes`.

2.4 Desarrollo

Las tareas de generación, compilación, ejecución y prueba de los analizadores léxico y sintáctico que son necesarias para realizar el compilador de esta práctica se han automatizado mediante la herramienta `Ant`. Esta es una herramienta que permite automatizar diferentes tipos de tareas descritas en un script expresado como un documento en XML (normalmente llamado `build.xml`). Su uso resulta similar a los mecanismos de procesamiento por lotes proporcionados por algunos sistemas operativos, que usan ficheros `‘.bat’` o `‘.sh’` para describir un script de actuación o a las herramientas `Makefile` utilizadas por algunos compiladores para automatizar los procesos de compilación y enlazado de acuerdo a ficheros `‘.mak’`.

El primer paso para utilizar la herramienta `ANT` es la instalación de la misma. Se puede encontrar toda la información acerca de ella en la siguiente dirección Web:

`http://ant.apache.org`

El documento `build.xml` proporcionado en el directorio `doc/config` contiene la descripción de todas las tareas necesarias para realizar el compilador. No obstante, para que éstas funcionen es necesario importar las bibliotecas `JFlex.jar` y `Cup.jar` que se incluyen en el directorio `/lib`. Por ello, se recomienda copiar ambos archivos dentro del directorio `'lib'` de Ant. Por ejemplo, si tenemos Ant instalado en `C:/ant`, debemos copiar los dos archivos en el directorio `C:/ant/lib`

Para invocar las tareas de Ant desde la línea de comandos debe abrirse una consola de sistema y situarse sobre el directorio que contenga el fichero con la especificación de las tareas (en nuestro caso `build.xml`) y escribir:

```
ant nombreTarea -Dpropiedad=valor
```

Donde `'nombreTarea'` es el nombre de una tarea definida en el fichero de tareas XML, `'propiedad'` el nombre de una propiedad definida en ese fichero y `'valor'` el valor que tendrá esa propiedad, siendo este parámetro opcional. Además los ficheros de descripción de tareas de Ant permiten especificar una tarea por omisión. En ese caso no haría falta poner el `nombreTarea` tampoco².

Para realizar el compilador en el directorio `/doc/build` se proporciona el fichero `build.xml` que se incluye y que contiene la definición de las siguientes tareas, que están a disposición del alumno:

- `clear`. Borra los archivos generados por procesos anteriores de compilación.
- `jflex`. Lee el archivo `'doc/specs/scanner.flex'` y genera el código fuente del analizador léxico asociado en el archivo `'src/compiler/lexical/Scanner.java'`.
- `cup`. Lee el archivo `'doc/specs/parser.cup'` y genera los archivos de código fuente asociados `'src/compiler/syntax/sym.java'` y `'src/compiler/syntax/parser.java'` que corresponden a la implementación del analizador sintáctico.
- `build`. Llama por orden a las tres tareas anteriores y compila todos los archivos fuente, dejando las clases generadas en el directorio `'classes'`. Queda así generado el compilador.
- `flexTest`. Llama al método `main` de la clase `'LexicalTestCase'` pasándola como parámetro el nombre de un fichero situado en el directorio `/doc/test`. Para indicar qué fichero queremos compilar se ha de modificar en el fichero `build.xml` la siguiente línea:

```
<property name="test-file-name" value="testA.ha" />
```

Cambiando `'value'` con el nombre del fichero que queramos. Los casos de prueba que desarrolle el alumno deberán situarse en el directorio `/doc/test` y llamarlos tal y como se ha indicado. Otra opción es pasar este valor como parámetro en la llamada a Ant. Por ejemplo:

```
ant flexTest -Dtest-file-name=testB.ha
```

² Existen también entornos de desarrollo integrados (IDEs) que disponen de la herramienta Ant incorporada lo que simplifica su uso. Un ejemplo de tales entornos es Eclipse (www.eclipse.org)

- `cupTest`. Llama al método `main` de la clase `SyntaxTestCase` de forma similar a lo descrito en la tarea `'flexTest'`. En este caso se probará el analizador léxico y el sintáctico.

Estas dos últimas tareas de test invocan previamente a las tareas `clear` y `build`, por lo que no es necesario llamarlas previamente.

El alumno puede utilizar estas tareas y, si lo desea, diseñar otras nuevas. Sin embargo, antes de la entrega es fundamental comprobar que las tareas `'build'`, `'flexTest'` y `'cupTest'` funcionan correctamente con nuestro compilador:

3 Entrega

En principio, el alumno solamente debería prestar atención a las clases de soporte dentro de los paquetes `es.uned.lsi.compiler.lexical` y `es.uned.lsi.compiler.syntax` de la librería `compiler-api.jar` y (en todo caso) modificar las clases dentro de los paquetes `compiler.lexical` y `compiler.syntax`.

Sin embargo, en la práctica el trabajo del alumno se centra en definir las especificaciones para los analizadores léxico y sintáctico (ficheros `JFlex` y `Cup` respectivamente) con lo que no es preciso grandes modificaciones de las clases proporcionadas.

3.1 Análisis léxico

El analizador léxico debe implementarse dentro del paquete `'compiler.lexical'`. Para ello, ha de utilizarse la clase proporcionada `'Token.java'` como medio de comunicación con el analizador sintáctico. Por tanto, el objeto que ha de devolverse al pedir un TOKEN (mediante el uso de la función `next_token` desde el parser) ha de ser de este tipo (como se ve en el código de `scanner.flex` en el listado 1). Esta clase puede ser completada con más atributos y métodos si el alumno lo considera necesario pero NO ha de eliminarse nada del código proporcionado (consulte la documentación en `doc/api`). En caso de que se desee modificar, se recomienda extenderla con una clase que incorpore la nueva funcionalidad deseada.

El alumno ha de modificar `scanner.flex` para crear el autómata apropiado que reconozca los TOKENS del lenguaje pedidos.

Dentro de `scanner.flex` se crea un nuevo TOKEN con la sentencia:

```
Token token = new Token (CONSTANTE_NUMERICA);
```

Donde `CONSTANTE_NUMERICA` es un valor numérico arbitrario que identifica de manera unívoca el tipo de Token emitido. Adicionalmente, es necesario incorporar dentro del objeto Token información acerca del lexema del TOKEN, así como el número de línea y columna donde está ubicado dentro del código fuente³. Estas tres informaciones las provee `JFlex` a través del método `yyText ()` y de las variables `yyline` e `yycolumn` respectivamente. Por tanto, tal como puede verse en el fichero proporcionado, la secuencia de instrucciones para emitir un Token de tipo SUMA sería similar a lo siguiente:

³ Esta información podrá ser posteriormente explotada por el analizador sintáctico descrito con `Cup`.

Listado 1. Acción para emitir un token SUMA desde JFlex.

```
Token token = new Token (SUMA);  
  
token.setLine (yyline + 1);  
  
token.setColumn (yycolumn + 1);  
  
token.setLexema (yytext ());  
  
return token;
```

En aquellos casos en los que el alumno identifique que el analizador léxico debe emitir un error léxico las acciones a realizar son dos:

- En primer lugar construir un objeto de la clase `LexicalError` para encapsular toda la información relativa al error. Esta clase encapsula esencialmente el lexema a la entrada que causó el error y el número de línea y columna dentro del fichero fuente de entrada donde se detectó el mismo (consulte la documentación en doc/api).
- En segundo lugar, debe utilizarse la clase `LexicalErrorManager` para emitir el mensaje de error por la salida estándar (consulte la documentación en doc/api)

Listado 2. Acción para emitir un mensaje de error léxico desde JFlex

```
LexicalError error = new LexicalError ();  
  
error.setLine (yyline + 1);  
  
error.setColumn (yycolumn + 1);  
  
error.setLexema (yytext ());  
  
lexicalErrorManager.lexicalFatalError (error);
```

La clase '`LexicalErrorManager`' aparte de ser usada para mostrar los errores léxicos puede ser usada como *mecanismo de traza* dentro de la especificación de JFlex. Para ello se ofrecen los métodos '`lexicalDebug`' y '`lexicalInfo`', que permiten emitir un mensaje, pasado como parámetro por la salida estándar. Las llamadas a `lexicalDebug` se utilizan para emitir informaciones que ayudan a depurar el código del analizador léxico. Las llamadas a `lexicalInfo` se utilizan para emitir mensajes informativos acerca del estado de progreso del analizador léxico. En todo caso se deja al alumno la forma de utilizar dichos métodos. Aunque no debe olvidar que al producirse un error es **obligatoria** la llamada a '`lexicalErrorManager.lexicalFatalError`' tal y como se ha explicado.

3.2 Análisis sintáctico

El analizador sintáctico se implementa dentro del paquete `compiler.syntax` y las clases generadas por la especificación de Cup se incluyen automáticamente dentro de dicho paquete. El trabajo de esta segunda fase consiste en completar el fichero de especificación `parser.cup` para obtener las clases `sym.java` y `parser.java`.

Dentro de la especificación incluida en el marco de trabajo de referencia existen una serie de declaraciones que NO deben modificarse bajo ningún concepto (aunque sí pueden ampliarse). Entre ellas destaca:

- El bloque *parser code*. Dentro de este bloque se incluyen las declaraciones de las funciones manejadoras de los errores sintácticos. Estas funciones son llamadas de forma automática por el parser al producirse un error recuperable o irrecuperable en el análisis sintáctico y utilizan llamadas al método `'syntaxError'` o `'syntaxFatalError'` de la clase `'SyntaxErrorManager'`. De forma similar a como se explicó en el análisis léxico, esta clase se utiliza como gestor de errores pero también puede ser utilizada como mecanismo de traza, disponiendo de los métodos para `syntaxDebug` y `syntaxInfo` homólogos a los descritos en la sección anterior.

Por último, es necesario describir cuales son las modificaciones que hay que hacer en la especificación de JFlex para que se integre con la herramienta Cup. La estructura del documento de especificación JFlex (`scanner.flex`) ya está preparada para posibilitar la integración. No obstante, las acciones asociadas a cada regla patrón acción en JFlex tienen que cumplir las siguientes restricciones:

- Cada acción debe emitir un objeto Token tal y como se explicó en el listado 1
- El Token emitido debe corresponderse con alguno de los terminales definidos en Cup.

En efecto, cuando se describe la especificación gramatical de Cup, se debe indicar el conjunto de elementos terminales que la constituyen. Para ello existe, dentro del fichero de especificación de Cup una sección específica. El siguiente listado es un ejemplo de esta sección que se corresponde con las declaraciones proporcionadas en la estructura de referencia. El alumno deberá reescribir estas declaraciones para que se ajusten a la especificación de su compilador.

Listado 3. Bloque de declaración de terminales en Cup

```
// Declaración de terminales (Ejemplo)

terminal Token PLUS;

terminal Token MINUS;
```

En este ejemplo se ha declarado en la gramática que se utilizarán dos elementos terminales llamados PLUS y MINUS para representar al toquen '+' y '-' respectivamente e indicar que son objetos de tipo Token (ya que es la clase que manejarán JFlex y Cup). Estas declaraciones sirven para indicar a Cup que debe generar una clase que contenga una constante simbólica de tipo entero para representar cada elemento terminal. En concreto, en el ejemplo anterior la clase que se generaría contendría la declaración de dos constantes, llamadas PLUS y MINUS.

De acuerdo a esta idea, el último paso de integración que tenemos que realizar consiste en cambiar la acción para emitir TOKEN, mostrada en el listado 1, por la que se ilustra en el listado 4.

Listado 4. Acción para emitir un token SUMA desde JFlex hacia Cup

```
Token token = new Token (sym.PLUS);  
  
token.setLine (yyline + 1);  
  
token.setColumn (yycolumn + 1);  
  
token.setLexema (yytext ());  
  
return token;
```

La única diferencia con el listado 1 es que ahora el identificador unívoco de TOKEN es el valor de una de las constantes definidas dentro de la clase Cup.

3.3 Realización de pruebas

Para terminar la descripción de este marco de trabajo vamos a discutir el contenido de las clases LexicalTestCase y SyntaxTestCase que son utilizadas para probar las fases del compilador que abarca esta práctica.

3.3.1 Pruebas del analizador léxico

Para evaluar el analizador léxico generado a través de la especificación de JFlex deben seguirse los siguientes pasos:

- Escribir un fichero de código fuente del lenguaje especificado en la práctica (correcto o incorrecto) y depositarlo dentro del directorio doc/test
- Escribir la especificación completa del analizador léxico completando la estructura proporcionada dentro del fichero doc/specs/scanner.flex de acuerdo a las directrices impuestas a lo largo de este documento
- Si la especificación de JFlex referencia las constantes de la clase sym generadas por el analizador sintáctico ejecutar la tarea Ant 'Cup' para generar el código fuente de las clases que implementan el autómata del parser (sym.java y parser.java)
- Ejecutar la tarea de Ant 'jflex' para obtener en el paquete compiler.lexical la implementación del autómata scanner.java de acuerdo a la especificación en JFlex
- Compilar todo el proyecto para obtener en /classes el código binario del programa. Estos tres últimos puntos pueden realizarse de forma automática invocando a la tarea de Ant 'build'
- Ejecutar la tarea Ant 'flexTest' para ejecutar el programa LexicalTestCase sobre uno de los ficheros de código fuente contenidos dentro de doc/test. Recuerde que el nombre del fichero que se evalúa por defecto es TestA pero puede cambiarse editando el fichero build.xml. Esta tarea llama previamente a la tarea build, por lo que *englobaría los puntos anteriores*.

En esencia el código del programa de prueba `LexicalTestCase` va solicitando al scanner uno por uno los TOKENS del fichero de entrada y los va imprimiendo por la salida estándar haciendo uso de las facilidades de traza de la clase `LexicalErrorManager`. Aunque no es necesario comprender su funcionamiento para ejecutar las pruebas a continuación discutimos el funcionamiento de la clase `LexicalTestCase` que se ilustra en el listado 5.

Listado 5. Código de prueba de la clase `LexicalTestCase`

```
while (anObject instanceof Token)
{
    Token aToken = (Token) anObject;

    if (aToken.sym == sym.EOF) break;

    lexicalErrorManager.lexicalInfo (aToken);

    anObject = aScanner.next_token ();
}

lexicalErrorManager.lexicalInfo ("End of file.");
```

Como puede apreciarse mediante un bucle `while` se van solicitando uno por uno todos los tokens del fichero de entrada invocando al método miembro `next_token ()`. En cada iteración se imprime el Token por la salida estándar invocando al método `LexicalInfo` de la clase `LexicalErrorManager` y se solicita un nuevo Token hasta que se encuentre el final del fichero (identificada por la constante simbólica `sym.EOF` de la clase `sym`). El resultado de la ejecución de este programa sobre un código fuente debería ser el listado de todos los token encontrados por el scanner en el fichero.

3.3.2 Pruebas del analizador sintáctico

Para evaluar el analizador sintáctico generado a través de la especificación de Cup deben seguirse los siguientes pasos:

- Escribir un fichero de código fuente en el lenguaje especificado en la práctica (correcto o incorrecto) y depositarlo dentro del directorio `doc/test`
- Escribir la especificación completa del analizador léxico completando la estructura proporcionada dentro del fichero `doc/specs/scanner.flex` de acuerdo a las directrices impuestas a lo largo de este documento
- Escribir la especificación completa del analizador sintáctico completando la estructura proporcionada dentro del fichero `doc/specs/parser.cup` de acuerdo a las directrices impuestas a lo largo de este documento
- Ejecutar la tarea de Ant 'jflex' para obtener en el paquete `compiler.lexical` la implementación del autómata `scanner.java` de acuerdo a la especificación en Jflex
- Ejecutar la tarea de Ant 'cup' para obtener en el paquete `compiler.syntax` la implementación del analizador sintáctico (`sym.java` y `parser.java` de acuerdo a la especificación en Cup)

- Compilar todo el proyecto para obtener en /classes el código binario del programa. Estos tres últimos puntos pueden realizarse de forma automática invocando a la tarea de Ant 'build'
- Ejecutar la tarea Ant 'cupTest' para ejecutar el programa SyntaxTestCase sobre uno de los ficheros de código fuente contenidos dentro de doc/test. Recuerde que el nombre del fichero que se evalúa por defecto es TestA pero puede cambiarse editando el fichero build.xml (consulte la sección 3.3.1). *Esta tarea llama previamente a la tarea build, por lo que englobaría los puntos anteriores.*

Como puede verse en el listado 6, el código del programa de prueba construye un stream de entrada conectado al fichero de código fuente cuyo nombre se pasa como parámetro en la variable 'fileName'. A partir de él construye el Scanner y el parser y arranca este último. Una vez arrancado el parser, éste irá solicitando al scanner los TOKENS de acuerdo a las reglas gramaticales para intentar construir un árbol de análisis sintáctico. El resultado de esta ejecución será la emisión por la salida estándar de un mensaje 'Starting parsing...', seguido opcionalmente de una serie de mensajes de error y seguido del mensaje 'Parsing process finished'. El listado 6 ilustra el código de la clase SyntaxTestCase. Puede verse que la creación de los objetos Scanner y parser se hace mediante introspección, así como la invocación del método parser.parse(). No es necesario que el alumno entienda ni modifique esta estructura.

Listado 6. Código de prueba de la clase SyntaxTestCase

```

Constructor scannerConstructor = scannerClass.getConstructor(Reader.class);
aScanner = (ScannerIF) scannerConstructor.newInstance(aFileReader);

// reflect parser
Class parserClass = Class.forName ("compiler.syntax.parser");

Constructor parserConstructor =
parserClass.getConstructor(java_cup.runtime.Scanner.class);

// reflect call parser.parse()

Method parseMethod = parserClass.getMethod("parse");

Object aParser = parserConstructor.newInstance(aScanner);

parseMethod.invoke(aParser);

```