Capítulo 1. Analizador Léxico

1 Objetivo

El Analizador Léxico tiene como misión:

- Procesar la entrada carácter a carácter y determinar dónde acaba un lexema y dónde empieza el siguiente.
- Clasificar los lexemas anteriores en tokens (categorías de lexemas).
- Identificar las secuencias de caracteres que no forman un lexema del lenguaje y notificar el error.

Supóngase el siguiente fichero de entrada y la descripción de los tokens que le sigue:

```
DATA
        float precio;
        int ancho;
CODE
        precio = (ancho - 3.0) * 1.18;
        print precio + 2 / 5;
```

```
class Lexicon {
        public static final int INT = 257;
        public static final int REAL = 258;

        public static final int CODE = 259;
        public static final int DATA = 260;

        public static final int PRINT = 261;

        public static final int LITERALREAL = 262;
        static final int LITERALINT = 263;
        static final int IDENT = 264;
        …
}
```

Con lo anterior, el Analizador Léxico deberá agrupar los caracteres formando los siguientes lexemas y clasificarlos en el token que les corresponda:

```
        [1:1] Token: 260. Lexema: DATA
        [2:5] Token: 258. Lexema: float
        [2:11] Token: 264. Lexema: precio
        [2:17] Token: 59. Lexema: ;
        [3:5] Token: 257. Lexema: int
        [3:9] Token: 264. Lexema: ancho
        [3:14] Token: 59. Lexema: ;
        [4:1] Token: 259. Lexema: CODE
        [5:5] Token: 264. Lexema: precio
        [5:12] Token: 61. Lexema: =
        [5:14] Token: 40. Lexema: (
        [5:15] Token: 264. Lexema: ancho
        [5:21] Token: 45. Lexema: -
        [5:23] Token: 262. Lexema: 3.0
        [5:26] Token: 41. Lexema: )
        [5:28] Token: 42. Lexema: *
        [5:30] Token: 262. Lexema: 1.18
        [5:34] Token: 59. Lexema: ;
        [6:5] Token: 261. Lexema: print
        [6:11] Token: 264. Lexema: precio
        [6:18] Token: 43. Lexema: +
        [6:20] Token: 263. Lexema: 2
        [6:22] Token: 47. Lexema: /
        [6:24] Token: 263. Lexema: 5
        [6:25] Token: 59. Lexema: ;
```

2 Trabajo autónomo del alumno

Diseñar e implementar esta fase *antes de seguir leyendo* el resto del documento. Una vez hecho, comparar la solución del alumno con la planteada a continuación.

3 Solución

3.1 Diseño. Creación de la Especificación

Antes de comenzar a codificar una fase del compilador (tanto el análisis léxico como el resto de las fases) hay que crear siempre una especificación mediante una notación (metalenguaje) que indique qué tiene que hacer dicha fase.

Hay tres razones fundamentales para esto:

- **Precisión**. En la descripción mediante el lenguaje natural (que es como se ha hecho en *Descripción del Lenguaje.pdf*) normalmente habrá ambigüedades y detalles sin aclarar.
- **Concisión**. Los requisitos expresados en un metalenguaje suelen ser mucho más breves que su equivalente en lenguaje natural.
- **Facilitar la Implementación**. La especificación de cada fase tendrá asociado a un método que indicará cómo generar el código a partir de dicha especificación. Incluso en algunas fases dicho método estará ya automatizado con herramientas.

3.1.1 Extracción de Requisitos

En primer paso en cada fase es recurrir a la descripción del lenguaje y extraer, de entre todos los requisitos de la misma, aquellos que correspondan a la fase que se esté tratando. En este tutorial una parte de la descripción del lenguaje se encuentra en *Descripción del Lenguaje.pdf* y otra parte se encuentra en el ejemplo contenido en *programa.txt*, por lo que habrá que revisar ambos ficheros en cada fase.

El contenido de *programa.txt* es el siguiente. Aquí es de donde se sacan la mayor parte de los requisitos de esta fase. Se deberán contemplar todos los tokens que pueden verse en el mismo.

```
DATA
        float precio;
        int ancho;
        int alto;
        float total;

CODE
        precio = 9.95;
        total = (precio - 3.0) * 1.18;
        print total;

        ancho = 10; alto = 20;
        print 0 - ancho * alto / 2;
```

En cuanto a *Descripción del Lenguaje.pdf*, la parte que afectaría al análisis léxico sería:

- Podrán aparecer comentarios en cualquier parte del programa.
- Las expresiones podrán estar formadas por literales enteros, literales reales y variables combinados mediante operadores aritméticos (suma, resta, multiplicación y división).
- Se podrán agrupar expresiones mediante paréntesis.

La única parte de este documento que aporta algo nuevo es la primera (añadir comentarios) ya que en este caso todos los operadores y los paréntesis ya habían aparecido en el ejemplo.

3.1.2 Metalenguaje Elegido

El análisis léxico debe indicar qué lexemas son válidos y a qué categoría pertenecen. Se utilizará como metalenguaje las *Expresiones Regulares*.

3.1.3 Especificación en el Metalenguaje

El primer paso es determinar qué tokens (tipos de lexemas) hay en el lenguaje. Se puede observar que en el ejemplo aparecen:

DATA REAL INT IDENT ; CODE = ( ) INT_CONSTANT REAL_CONSTANT * - / + PRINT

El segundo paso es asociar un patrón a cada uno de los tokens que indique qué lexemas pertenecen a cada uno. Cada patrón se representa con una Expresión Regular:

Token	Patrón
+	\+
-	-
*****	*
/	/
;	;
(	(
)	)
=	=
DATA	DATA
CODE	CODE
PRINT	print
INT	int
REAL	float
IDENT	[a-zA-Z][a-zA-Z0-9_]*
INT_CONSTANT	[0-9]+
REAL_CONSTANT	[0-9]+"."[0-9]+

Nótese cómo el hecho de hacer la especificación ha propiciado que haya habido que plantearse cuándo una constante real es válida. La especificación en lenguaje natural no entraba, por ejemplo, en si es válido o no que no haya decimales después del punto. Si el lenguaje se está creando partiendo de cero, este es el momento de decidirlo. Si, por el contrario, el lenguaje viene impuesto, entonces es el momento de consultarlo. Algo similar ocurre en el caso de los identificadores y de su tratamiento del guion bajo (_).

Una vez concretadas estas cuestiones mediante las *expresiones regulares*, ya se puede pasar a la implementación.

3.2 Implementación de la Especificación

3.2.1 Fichero de Especificación de ANTLR

Ahora hay que implementar una función que lea caracteres de un flujo y determine, mediante la implementación de los patrones, a qué categoría pertenece cada uno.

Aunque se podría hacer fácilmente a mano, para acelerar el proceso se utilizará la herramienta *ANTLR*. El primer paso es darle la información anterior (qué tokens hay en el lenguaje y qué patrón tiene cada uno) en el formato de la herramienta. En la carpeta *parser*

se tiene el fichero *Lexicon.g4* con un esqueleto de un fichero de ANTLR con todo el código habitual de cualquier analizador léxico. Sólo habría que añadir las reglas específicas de nuestro lenguaje):

```
lexer grammar Lexicon
    ;

DATA:   'DATA';
CODE:   'CODE';

PRINT: 'print';

INT:    'int';
FLOAT:  'float';

INT_CONSTANT:   [0-9]+;
REAL_CONSTANT:  [0-9]+ '.' [0-9]+;
IDENT:          [a-zA-Z][a-zA-Z0-9_]*;

PLUS:   '+';
MULT:   '*';
SUB:    '-';
DIV:    '/';

SEMICOLON: ';';

ASSIGN: '=';

OPEN_PAREN:     '(';
CLOSE_PAREN:    ')';

LINE_COMMENT:       '//' .*? ('\n'|EOF) -> skip;
MULTILINE_COMMENT:  '/*' .*? '*/'   -> skip;

WHITESPACE: [ \t\r\n]+ -> skip;
```

Se han dejado tal cual venían en el fichero las reglas de los dos tipos de comentarios que propone el esqueleto (ya que coinciden con los que queremos en nuestro lenguaje)

3.2.2 Uso de ANTLR

Una vez completado el fichero *Lexicon.g4,* quedaría usar *ANTLR* para que genere el código en Java del Analizador Léxico. Para ello hay sólo hay que ejecutar lo siguiente en línea de comando:

```
c:\..\> antlr.bat.
```

Aparecerá así un fichero *GrammarLexer.java* que contendrá la clase del mismo nombre que implementa el analizador léxico.

Nota. Hay veces en que *eclipse* no detecta el cambio en un fichero generado por una herramienta y por tanto seguirá compilando la versión antigua del mismo. Si esto ocurre, debe seleccionarse el proyecto en la ventana *Package Explorer* y pulsar F5 para actualizar.

4 Ejecución

Para probar el analizador léxico se incluye en *source.txt* un programa que tiene todos los tokens del lenguaje:

```
// Comentario
/* Prueba del otro comentario */
DATA
        float precio;
        int ancho;
CODE
        precio = (ancho - 3.0) * 1.18;
        print precio + 2 / 5;
```

Nota. Ahora *no se debe* ejecutar la clase *main.Main*. Si se hace, aparecerá el siguiente mensaje de error:

```
line 3:0 mismatched input 'DATA' expecting INT_CONSTANT
```

Esto es debido a que el *main* invoca al analizador *sintáctico* y este todavía no se ha hecho para el lenguaje del tutorial. El *main* que se tiene actualmente (que se copió con el Esqueleto del Traductor), sólo reconoce constantes enteras. Es por ello que da error sintáctico.

Para poder comprobar la salida del léxico, hay dos formas de hacerlo. La primera es usar un comando que se obtuvo en el Esqueleto del Traductor:

```
tokens.bat source.txt
```

Esto imprime los tokens reconocidos en dicho fichero fuente:

```
[@0,49:52='DATA',<'DATA'>,3:0]
[@1,56:60='float',<'float'>,4:1]
[@2,62:67='precio',<IDENT>,4:7]
[@3,68:68=';',<';'>,4:13]
[@4,72:74='int',<'int'>,5:1]
[@5,76:80='ancho',<IDENT>,5:5]
…
line 3:0 mismatched input 'DATA' expecting INT_CONSTANT
```

Nótese que, después de la traza, sigue saliendo el error sintáctico. Esto es normal ya que, después de imprimir la traza, intenta comenzar el análisis sintáctico. Simplemente hay que ignorar dicho error.

La segunda forma, por si se tiene curiosidad, sería haciendo a mano lo que hace el comando Tokens.bat. Habría que hacer otro *main* adicional que llame directamente al léxico en vez de la llamada al sintáctico que está haciendo el *main* actual. Para ello, se añadiría en la carpeta *parser* la siguiente clase *TestLexer.java*:

```
public class TestLexer {
    public static void main(String[] args) throws Exception {

        CharStream input = CharStreams.fromFileName("source.txt");
        GrammarLexer textLexer = new GrammarLexer(input);
        Token token;
        while ((token = textLexer.nextToken()).getType() != GrammarLexer.EOF) {
            System.out.printf("Line: %d \tcolumn: %d \tlexeme: '%s' \ttype: %d, \t
token: %s\n", token.getLine(), token.getCharPositionInLine() + 1, token.getText(),
token.getType(), textLexer.getVocabulary().getDisplayName(token.getType()));
        }

        System.out.println("Traza lexer finalizada");
    }
}
```

Si se ejecuta esta clase en vez de la que se haya en la carpeta *main*, se obtendrá la traza del analizador léxico buscada.

```
Line: 3,        column: 1,      lexeme: 'DATA',     type: 1, token: 'DATA'.
Line: 4,        column: 2,      lexeme: 'float',    type: 5, token: 'float'.
Line: 4,        column: 8,      lexeme: 'precio',   type: 8, token: IDENT.
Line: 4,        column: 14,     lexeme: ';',        type: 13, token: ';'.
Line: 5,        column: 2,      lexeme: 'int',      type: 4, token: 'int'.
Line: 5,        column: 6,      lexeme: 'ancho',    type: 8, token: IDENT.
Line: 5,        column: 11,     lexeme: ';',        type: 13, token: ';'.
Line: 6,        column: 1,      lexeme: 'CODE',     type: 2, token: 'CODE'.
Line: 7,        column: 2,      lexeme: 'precio',   type: 8, token: IDENT.
…
```

Durante la implementación de fases posteriores del compilador, es posible que éste, al compilar un fichero de prueba, señale errores que creamos que no debería. En estos casos, para averiguar qué está yendo mal, es conveniente ir acotando por fases. *Lo primero que hay que descartar* es que el léxico esté clasificando mal la entrada antes de hacer cambios en el sintáctico. Para eso es precisamente el comando *tokens.bat.* Permite averiguar qué le está entregando el léxico al sintáctico y así poder comprobar si el problema está en el analizador léxico o no.

5 Resumen de Cambios

Fichero	Acción	Descripción
Lexicon.g4	Modificado	Se han añadido las reglas (tokens y patrones)
source.txt	Modificado	Se ha creado una prueba que tuviera todos los tokens del lenguaje
GrammarLexer.java	Generado	Implementación del Analizador Léxico. Creado con ANTLR a partir de Lexicon.g4