

## 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;
```

```
public class Tokens {
    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 Solución

### 2.1 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 *Introducción.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.

#### 2.1.1 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 *Introducción.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 *Introducción.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 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.

### 2.1.2 Metalenguaje Elegido

El análisis léxico debe indicar qué lexemas (secuencias de caracteres) son válidos y a qué categoría pertenecen. Se utilizará como metalenguaje las *Expresiones Regulares* para indicar de manera precisa ambos aspectos.

### 2.1.3 Especificación

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 = ( ) LITERALINT LITERALREAL * - / + 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, el metalenguaje elegido:

Token	Patrón
+	\+
-	-
*	\*
/	/
;	;
(	(
)	)
=	=
DATA	DATA
CODE	CODE
PRINT	print
INT	int
REAL	float
IDENT	[a-zA-ZñÑ][a-zA-Z0-9_ñÑ]*
LITERALINT	[0-9]+
LITERALREAL	[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 contra el lenguaje viene impuesto, entonces es el momento de consultarlo. Algo similar ocurre en el caso de los identificadores y de su tratamiento de la ñ y del guion bajo (\_).

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

## 2.2 Implementación de la Especificación

### 2.2.1 Fichero de Especificación de JFlex

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 *JFlex*. 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 *sintactico* se tiene el fichero *lexico.l* con un esqueleto de un fichero de JFlex con todo el código habitual de cualquier analizador léxico. Solo habría que añadir las reglas específicas de nuestro lenguaje a la segunda sección de dicho fichero (parte con fondo gris):

```
/* -- No es necesario modificar esta parte ----- */
package sintactico;

import java.io.*;
import main.*;
import ast.Position;
%%
%byaccj
%unicode
%line
%column
%public
%{
    public Ylex(Reader in, GestorErrores gestor) {
        this(in);
        this.gestor = gestor;
    }

    public int line() { return yyline + 1; }
    public int column() { return yycolumn + 1; }
    public String lexeme() { return yytext(); }

    // Traza para probar el léxico de manera independiente al sintáctico
    public static void main(String[] args) throws Exception {
        Ylex lex = new Ylex(new FileReader(Main.programa), new GestorErrores());
        int token;
        while ((token = lex.yylex()) != 0)
            System.out.println(lex.line()+" "+lex.column()+"Token:"+token+"Lexema:"+
lex.lexeme());
    }

    private GestorErrores gestor;
}%
%%
```

```
/* -- Modificar aquí. Añadir reglas en esta sección ----- */

[+\-*/;\(\)=]    { return yytext().charAt(0); }

DATA              { return Tokens.DATA; }
CODE              { return Tokens.CODE; }
print             { return Tokens.PRINT; }
int               { return Tokens.INT; }
float             { return Tokens.REAL; }

[a-zA-ZñÑ][a-zA-Z0-9_ñÑ]*    { return Tokens.IDENT; }
[0-9]+             { return Tokens.LITERALINT; }
[0-9]+\.[0-9]*      { return Tokens.LITERALREAL; }

/*"([^\]|\"+([^\]|\"))*\"/*  /* Comentario de varias líneas como este */
//\".*              { }      // Comentario de una línea como este

[ \n\r]           { }
"\t"               { yycolumn += 3; } /* Para que coincida con la info del editor de Eclipse
(opcional). En eclipse: \t = 4 caracteres. En Jflex: \t = 1 carácter.*/

.                  { gestor.error("Léxico", "Lexema:"+yytext(), new Position(line(),column())); }
```

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)

Para acabar faltaría definir las constantes usadas para identificar los tokens. Para ello se crearía la clase *Tokens.java*:

```
package sintactico;

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

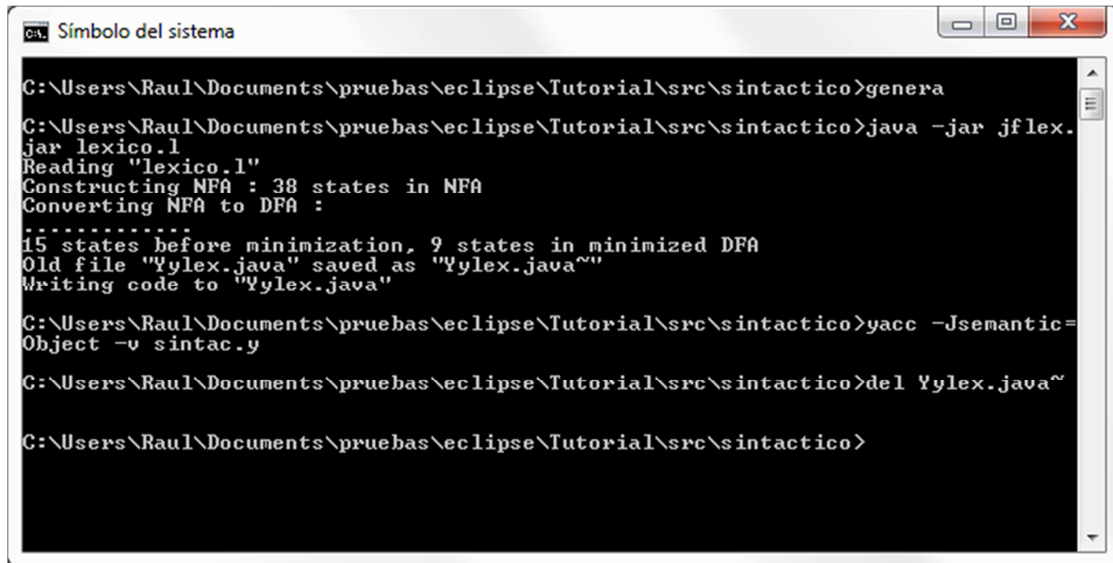
    public static final int CODE = 259;
    public static final int DATA = 260;

    public static final int PRINT = 261;

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

### 2.2.2 Uso de JFlex

Una vez completado el fichero *lexico.l* quedaría usar *JFlex* para que genere el código en Java del Analizador Léxico. Para ello hay que ir a la carpeta *sintactico* del proyecto y ejecutar el fichero *genera.bat*:



```
C:\Users\Raul\Documents\pruebas\eclipse\Tutorial\src\sintactico>genera
C:\Users\Raul\Documents\pruebas\eclipse\Tutorial\src\sintactico>java -jar jflex.jar lexico.l
Reading "lexico.l"
Constructing NFA : 38 states in NFA
Converting NFA to DFA :
-----
15 states before minimization, 9 states in minimized DFA
Old file "Ylex.java" saved as "Ylex.java~"
Writing code to "Ylex.java"
C:\Users\Raul\Documents\pruebas\eclipse\Tutorial\src\sintactico>yacc -jsemantic=Object -v sintac.y
C:\Users\Raul\Documents\pruebas\eclipse\Tutorial\src\sintactico>del Ylex.java~
C:\Users\Raul\Documents\pruebas\eclipse\Tutorial\src\sintactico>
```

Aparecerá un fichero *Ylex.java* que contendrá la clase que implementa el analizador léxico.

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

### 3 Ejecución

Para probar el analizador léxico se incluye en *entrada.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;
```

A continuación se ejecuta la clase *main.Main* para que se realice el análisis léxico:

```
Error en Sintáctico: [3:1] Token = 260, lexema = "DATA". syntax error
Error en Sintáctico: [3:1] Token = 260, lexema = "DATA". stack underflow...
```

Como era de esperar, se producen errores sintácticos ya que, hasta que no se cambie en el capítulo siguiente, el sintáctico actual solo reconoce como entrada un carácter punto y coma.

En lugar de usar la clase *Main*, otra forma de ejecutar el analizador léxico es mediante la clase *sintactico.Ylex*. En esta clase se ha incluido otro método *main* que se limita a hacer una traza de lo que está haciendo el léxico:

```
class Yylex {  
  
    // Traza para probar el léxico de manera independiente al sintáctico  
    public static void main(String[] args) throws Exception {  
  
        Yylex lex = new Yylex(new FileReader(Main.programa), new GestorErrores());  
        int token;  
        while ((token = lex.yylex()) != 0)  
            System.out.println("\t[" + lex.line() + ":" + lex.column() + "] Token: " +  
                               token + ". Lexema: " + lex.lexeme());  
    }  
  
    ... // Resto de la clase  
}
```

Si se ejecuta la clase *sintactico.Yylex* (en vez de la clase *Main*), se obtiene lo siguiente:

```
[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: ;
```

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 ello es precisamente este *main* que se encuentra en la clase *Yylex.java*. Permite averiguar que le está entregando el léxico al sintáctico y así poder comprobar si el problema está en el analizador léxico o no.

## 4 Resumen de Cambios

Fichero	Acción	Descripción
<b>lexico.l</b>	Modificado	Se han añadido las reglas (tokens y patrones)
<b>Tokens.java</b>	Creado	Se ha asignado un número único a cada Token
<b>entrada.txt</b>	Modificado	Se ha creado una prueba que tuviera todos los tokens del lenguaje
<b>Yylex.java</b>	Generado	Implementación del Analizador Léxico. Creado con JFlex a partir de <i>lexico.l</i>