Análisis léxico con ANTLR

En esta práctica trabajaremos con ANTLR a nivel léxico, estudiaremos los elementos que esta herramienta proporciona para especificar analizadores léxicos y los aplicaremos a la resolución de distintos aspectos como la identificación de *tokens*, el tratamiento de palabras reservadas y comentarios o la sensibilidad a mayúsculas y minúsculas.

Un ejemplo simple

En ANTLR no se utilizan expresiones regulares y autómatas finitos para especificar e implementar, respectivamente, analizadores léxicos. En su lugar se aprovechan la misma notación y el mismo tipo de reconocedores utilizados en la generación de analizadores sintácticos, o sea gramáticas EBNF y reconocedores recursivos descendentes. Con esta decisión se gana en homogeneidad a nivel de notación, aunque en algunas ocasiones se echan de menos algunas facilidades propias de las expresiones regulares, fundamentalmente el hecho de no tener que preocuparse de que los patrones léxicos tengan prefijos comunes, problema que en los reconocedores descendentes está siempre presente. El siguiente analizador léxico detecta números, paréntesis, separadores y operadores aritméticos, además de ignorar blancos, tabuladores y saltos de línea:

```
////////////////////////////////
// Analizador léxico
////////////////////////////////

class Analex extends Lexer;

protected NUEVA_LINEA: "\r\n"
{newline();};
BLANCO: (' '|'\t'|NUEVA_LINEA)
{$setType(Token.SKIP);};
protected DIGITO: '0'..'9';
NUMERO: (DIGITO)+('.'(DIGITO)+)?;
OPERADOR: '+'|'-'|'/'|'*';
PARENTESIS: '('|')';
SEPARADOR: ';';
```

Como se puede observar la especificación es bastante intuitiva y se comprende a simple vista, tan solo merece la pena comentar algunos detalles:

- Las reglas se especifican en notación EBNF, por lo que se podrán utilizar los operadores de cierre (`*`), cierre positivo (`+`), opción (`|`) y opcional (`?`). Es aconsejable encerrar entre paréntesis todos aquellos elementos sujetos a los operadores unarios `*`, `+` y `?`.
- Los caracteres se especifican con comillas simples (p.e. `'0'`,`'\t'` y `'+'`).
- Las secuencias de caracteres se especifican con comillas dobles como por ejemplo `"\r\n"`, en este caso concreto cabe reseñar que la combinación se corresponde con la manera en la que en *Windows* se codifica la nueva línea (si solamente pusiésemos `'\n'` no funcionaría

en esta plataforma). Comprueba qué ocurre en Linux y cámbialo si es necesario.

- Con la acción `{$setType(Token.SKIP);}` se establece que los *tokens* reconocidos en la regla correspondiente deben ser ignorados y no serán, por tanto, visibles para el analizador sintáctico.
- Los rangos de caracteres se especifican con el operador `".."` (p.e.`'0'..'9'`).
- Con la acción `{newline();}` se establece que los *tokens* correspondientes provocan un cambio de línea, por lo que los contadores de fila y columna son actualizados convenientemente.
- Con el modificador `protected` se indica que la regla correspondiente no describe un *token*, sino que sirve de apoyo en la definición de una regla más compleja. Así, en el ejemplo, la regla `BLANCO` se apoya en la regla protegida `NUEVA_LINEA`.

Como todos los analizadores en ANTLR, el analizador léxico se implementa a través de una clase. En nuestro caso esta clase se llamará `Analex` y siempre será una subclase de `CharScanner`, que es la clase que implementa el comportamiento genérico de los analizadores léxicos. La clase `Analex` tiene varios métodos que permiten lanzar el reconocimiento de *tokens* y consultar o modificar atributos propios del analizador. De ellos los dos más destacados son:

- `nextToken`: cada vez que se ejecuta devuelve el siguiente token identificado en el flujo de entrada.
- `makeToken`: es heredado de la clase `CharScanner` y es el utilizado por el analizador para construir los objetos de la clase `Token` y actualizar sus atributos. Como veremos más adelante tendremos que redefinirlo si queremos añadir más atributos a los ya implementados en ANTLR por defecto.

Probando el analizador léxico

Gracias al método `nextToken` podremos probar el funcionamiento del analizador con un simple bucle que muestre el lexema reconocido y el *token* asignado (un número entero) hasta encontrar el final del fichero.

```
////////////////////////////////////
// Procesador.java (clase principal)
////////////////////////////////////

import java.io.*;
import antlr.*;

public class Procesador {

     public static void main(String args[]) {

          try {

               FileInputStream fis =
               new FileInputStream(args[0]);
               Analex analex = new Analex(fis);
```

```
                Token token = analex.nextToken();
                while(token.getType() != Token.EOF_TYPE) {
                System.out.println(token);
                token = analex.nextToken();
                }
           }catch(ANTLRException ae) {
                System.err.println(ae.getMessage());
           }catch(FileNotFoundException fnfe) {
                System.err.println("No se encontró el
           fichero");
           }
     }
}
```

ANTLR utiliza la clase `CommonToken`, que hereda de la clase abstracta `Token`, para representar los *tokens*. Algunos de los métodos y atributos más usados de esta clase son:

- `getText()`: calcula el lexema asociado al *token*.
- `getType()`: calcula el tipo de *token*, representado por un número entero.
- `EOF_TYPE`: *token* fin de entrada, representado por el valor 1.
- `getLine()`: calcula la línea en la que se encuentra el *token*.
- `getColumn`: calcula la columna en la que se encuentra el *token*.

Ya sólo falta editar un fichero con la entrada y comprobar que los tokens se reconocen correctamente. Una posible entrada para la especificación léxica anterior es:

```
1+2*4;
(1-5)/6;
```

Definiendo una nueva clase de tokens

Si con lo que nos ofrece ANTLR no tenemos suficiente para representar la información que necesitamos de los *tokens*, podemos definir una nueva clase que implemente los *tokens*. Lo usual será extender la clase `CommonToken` para añadir más características, por ejemplo la siguiente clase incluye como información adicional un atributo de tipo `String`:

```
////////////////////////////////
//MiToken.java
////////////////////////////////

import antlr.*;

public class MiToken extends CommonToken {

     String nuevoAtributo;

     public MiToken() {
     }

     public void setNuevoAtributo(String s) {
          nuevoAtributo=s;
     }
```

```
	public String getNuevoAtributo() {
		return nuevoAtributo;
	}

	public String toString() {
		return
	super.toString()+"[nuevoAtr=\""+nuevoAtributo+"\"]";
	}
}
```

Se incluye un constructor por defecto, un par de métodos para actualizar y recuperar respectivamente el valor del nuevo atributo y la redefinición del método `toString` para que el nuevo atributo también aparezca cuando visualicemos los tokens.

Gracias al método `setTokenObjectClass` de la clase `Analex`, podemos indicar al analizador léxico que en lugar de `CommonToken` utilice la clase que hemos definido. Esto lo haríamos en la clase principal, justo después de crear el objeto que realizará el análisis léxico:

```
	Analex analex = new Analex(fis);
	analex.setTokenObjectClass("MiToken");
```

Por último habrá que indicarle al analizador léxico que a la hora de construir los *tokens* calcule el valor apropiado de este nuevo atributo. Esto se puede hacer extendiendo el método `makeToken` para la clase `Analex`. El lugar apropiado para ello es la zona reservada para incluir código en la especificación del analizador:

```
class Analex extends Lexer;

// ... Zona de opciones

{
	protected Token makeToken(int type) {

		// Crea un token tal y como lo haría CommonToken
		MiToken token = (MiToken)super.makeToken(type);
		// Añade la información del nuevo atributo
		token.setNuevoAtributo(...);
		// Devuelve el token completo
		return token;
	}
}

// ... Zona de reglas
```

Prefijos comunes

Los prefijos comunes suponen un problema en el modelo de reconocimiento que utiliza ANTLR. El hecho de utilizar reconocedores descendentes obliga a tener que predecir qué regla aplicar en cada momento, esto no sería necesario con una implementación basada en autómatas y expresiones regulares ya que el no-determinismo permite llevar en paralelo todas las expresiones regulares para quedarnos

con la que más nos convenga después de haber procesado el lexema completamente. En el caso del análisis léxico este problema se agrava por el hecho de que todas las reglas están activas en todo momento, lo que aumenta las posibilidades de conflicto.

ANTLR proporciona dos mecanismos para resolver estas situaciones. El primero consiste en permitir variar el número de símbolos de anticipación lo que hace que podamos disponer de gramáticas LL(k) para un k mayor que 1. El valor de este atributo se indica en la zona de opciones específicas del analizador léxico. Esta solución permite resolver aquellas situaciones que se pueden decidir conociendo un prefijo de longitud fija de las reglas en conflicto. Por ejemplo si tenemos que reconocer el separador ":" y el operador ":=" haríamos lo siguiente:

```
class Analex extends Lexer;

     options {

          k=2;

     }

     SEPARADOR: ':';

     ASIGNACION: ":=";
```

La solución anterior no puede aplicarse de forma general ya que hay conflictos para los que no se puede establecer de manera fija el tamaño del prefijo que nos permite decidir la regla a aplicar. Esto ocurre por ejemplo con las siguientes reglas que detectan números enteros y reales:

```
     protected DIGITO: '0'..'9';

     LIT_REAL: (DIGITO)+ '.' (DIGITO) *;

     LIT_ENTERO: (DIGITO)+;
```

En estos casos tendremos que echar mano del segundo mecanismo, los predicados sintácticos. Un predicado sintáctico no es más que una expresión EBNF que se coloca al principio de una regla (por medio del operador =>). Antes de lanzar el reconocimiento de una regla con predicado sintáctico se realiza una simulación que permitirá saber si esa regla encajará con la entrada o no. Los predicados sintácticos permiten implementar un *backtracking* selectivo, de manera que sólo se lanzará en las situaciones que especifiquemos en la gramática. Por ejemplo, para el conflicto entre números reales y enteros, anotaríamos la regla de los reales con un predicado sintáctico que abarcara hasta el punto:

```
     protected DIGITO: '0'..'9';

     NUMERO : ((DIGITO)+ '.') => (DIGITO)+ '.' (DIGITO)*
              | (DIGITO)+;
```

Estamos obligados a incluir en una misma regla ambos tokens porque ANTLR ignora los predicados sintácticos si se encuentran en reglas que sólo tienen una opción.

Para generar el *token* apropiado en cada caso podemos hacer uso de la acción `$setType` que ya hemos usado anteriormente para ignorar los blancos. En este caso la utilizaremos para indicar cuál es el tipo de *token* apropiado para cada regla. Estos *tokens* "*virtuales"* se habrán definido previamente en la sección `tokens`:

```
tokens{
     LIT_REAL;
     LIT_ENTERO;
     }

protected DIGITO: '0'..'9';

NUMERO: ((DIGITO)+ '.') => (DIGITO)+ '.' (DIGITO)*
          {$setType(LIT_REAL);}
     | (DIGITO)+
          {$setType(LIT_ENTERO);};
```

Palabras reservadas

Las palabras reservadas se especifican directamente en la sección `tokens` del analizador léxico a través de una secuencia de pares lexema-*token*. Por ejemplo:

```
tokens {
     // tipos
     INT = "int";
     CHAR = "char";
     ...
     // instrucciones
     IF = "if";
     WHILE = "while";
     ...
}
```

ANTLR comprobará antes de aplicar una regla si en la entrada hay alguna de las palabras enumeradas en la sección `tokens`, justo lo que nos hace falta para distinguir las palabras reservadas de los identificadores que habitualmente obedecerán a una regla del tipo:

```
protected LETRA: ('A'..'Z')|('a'..'z');
IDENT: (LETRA)((DIGITO)|(LETRA))*;
```

Comodín y conjuntos negativos

Además de los rangos de caracteres que ya hemos utilizado previamente, por ejemplo para definir `DIGITO` (con el rango `'0'..'9'`), ANTLR proporciona otros dos elementos que hacen referencia a conjuntos de caracteres:

- El carácter comodín, representado por un punto (`.`) que encaja con cualquier carácter simple.
- El operador de complemento (`~`) que encaja con todos aquellos caracteres que no estén en un conjunto dado. Por ejemplo todos los caracteres que no son dígitos se reconocerían con la siguiente regla:

```
NODIG: ~('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9');
```

Hay que tener especial cuidado a la hora de utilizar estos elementos ya que por defecto ANTLR asume que el alfabeto de entrada contiene sólo aquellos caracteres presentes en la especificación. De esta forma, en el siguiente ejemplo el comodín (`.`) sólo se refiere a los caracteres `'a'`, `'b'` y `'c'` y la expresión `~('a'|'b')`sólo reconoce el carácter `'c'`:

```
protected LETRA : 'a'..'c';
ALFABETO : .;
TODOS_MENOS_AB : ~('a'|'b');
```

Se puede, no obstante, establecer desde un principio el conjunto de caracteres que constituyen el alfabeto de entrada. Para ello tendremos que asociar a la opción `charVocabulary` el rango deseado. Lo normal es utilizar el rango `'\3'..'\377'` que incluye los caracteres *Unicode* más comunes:

```
options {

     charVocabulary = '\3'..'\377';

}
```

Comentarios de bloque: la opción `greedy`

Los comodines y los conjuntos negativos son de gran utilidad, por ejemplo, a la hora de definir elementos como los comentarios dentro de los cuales podemos encontrar cualquier cosa. Una primera aproximación para identificar los comentarios de apertura y cierre en `JAVA` es la siguiente:

```
COMENTARIO: "/*" (.)* "*/"
          {$setType(Token.SKIP);};
```

Los bucles en ANTLR son (por defecto) voraces, lo que significa que mientras se pueda mantener la ejecución de un bucle se mantendrá. En la regla anterior eso implica que `(.)*` reconocerá cualquier carácter, por lo que consumirá también los caracteres de cierre del comentario y no parará de consumir caracteres hasta encontrar el final de la entrada.

Una posible solución a esta voracidad es detallar exactamente qué caracteres se permiten dentro del comentario. Por ejemplo:

```
COMENTARIO: "/*" ('*' ~('/')
          | ~ ('*')
          )* "*/" {$setType(Token.SKIP);};
```

En este caso el carácter comodín se sustituye por una expresión que tiene en cuenta si aparece un asterisco seguido de algo que no sea una barra, o simplemente si aparece cualquier cosa que no sea un asterisco.

Otra posible solución pasa por hacer uso de la opción `greedy`, que nos permite especificar que en ciertas ocasiones ANTLR no se comporte de forma voraz con los bucles. Podemos hacer que `greedy`, cuyo valor por defecto es `true`, tome el valor `false` localmente a una regla de la siguiente forma:

```
COMENTARIO: "/*" (options {greedy=false;}:.)* "*/"
          {$setType(Token.SKIP);};
```

Así, antes de iniciar una nueva iteración del bucle se comprobará si puede finalizar o no, si es posible finalizar se finalizará. Aún quedaría un problema más por resolver y es que se necesitan conocer dos caracteres (`"*/"`) para saber si hemos llegado al final de un comentario o si debemos quedarnos dentro del bucle. Esto se resuelve simplemente indicando que `k` es 2, lo que permite a ANTLR hacer uso de esos dos caracteres que necesita para tomar la decisión correcta:

```
options {
     k = 2;
}
```

Ejercicios

1. Compilar y probar el analizador léxico que aparece en el enunciado.

2. ¿Qué ocurre cuando se quita el modificador `protected` de la regla `NUEVA_LINEA`? ¿Por qué?

3. Ampliar el analizador del ejercicio 1 para que además de la fila y la columna muestre el nombre del fichero, para ello habrá que extender la clase `CommonToken`. La clase `Analex` proporciona un par de métodos `setFilename` y `getFilename` que permiten establecer y recuperar, respectivamente, el nombre del fichero que se está analizando en cada momento.

4. Ampliar el reconocedor anterior de manera que ignore comentarios de línea (que comienzan con `//`) y de apertura y cierre (con los símbolos `/*` y `*/` respectivamente). ¿Cómo se consigue que los contadores de fila y columna se actualicen adecuadamente?

5. La solución propuesta en el enunciado para los comentarios (que no hace uso de la opción `greedy`)reconoce correctamente el comentario `/* ***/` pero falla al procesar el comentario `/* **/`. ¿A qué se debe ese fallo? Corrígelo.

6. Escribir un analizador léxico para los elementos del lenguaje `C` presentes en el siguiente ejemplo, las palabras reservadas están resaltadas en negrita:

```
void main(void) {
     int a, b;
     scanf("%d",&b);
     a=1;
     while (a<b) {
          if(2*a>=b)
```

```
                printf("punto medio %d\n",a);
            a++;
        }
}
```

7. Ampliar el analizador del ejercicio anterior para que no sea sensible a mayúsculas y minúsculas. Para ello se utilizará la opción `caseSensitiveLiterals`, que cuando recibe el valor `false` hace que las comparaciones con las palabras de la lista de *tokens* se realicen después de convertir la palabra reconocida a minúsculas.

8. Escribir un analizador léxico para los elementos de XML presentes en el siguiente ejemplo:

```
<biblioteca>
<libro>
<titulo>La insoportable levedad del ser</titulo>
<!-- No es tan raro como parece -->
<autor> Milan Kundera </autor>
<editorial> Tusquets </editorial>
</libro>
<libro>
<titulo>La princesa durmiente va a la
escuela</titulo>
<autor> Gonzalo Torrente Ballester </autor>
<!-- El símbolo "&" no se puede usar
directamente, en su lugar usaremos "&amp;".
Ocurre lo mismo para "<", en este caso usaremos
"&gt;" -->
<editorial> Plaza &amp; Janes </editorial>
</libro>
<libro>
<titulo> La isla del tesoro </titulo>
<autor> Robert L. Stevenson </autor>
<editorial> Juventud </editorial>
</libro>
<!-- Pendiente de leer -->
<libro>
<titulo> Yo que he servido al Rey de Inglaterra
</titulo>
<autor> Bohumil Hrabal </autor>
<editorial> Destino </editorial>
</libro>
</biblioteca>
```

9. Analiza el fichero que se detalla a continuación: *LeLiLexer.g*. Describe qué tokens y reglas aparecen y la utilidad que presentan. Escribe una clase que integre el método *main* y que permita probarlo.

```
header{
package leli;
/*-------------------------------------------*\
| Un intérprete para un Lenguaje Limitado(LeLi) |
\*-------------------------------------------*/
}
```

```
/**
* El objeto que permite todo el analisis léxico.
* notas: comentar todo esto al final del analex.
*/

class LeLiLexer extends Lexer;

options{
        charVocabulary = '\3'..'\377'; // Unicodes usuales
        exportVocab=LeLiLexerVocab; // Comentar al hacer el parser
        testLiterals=false; // comprobar literales solamente cuando se diga exp
        k=2; // lookahead
        }

tokens
{

        // Tipos basicos
        TIPO_ENTERO = "Entero" ;
        TIPO_REAL = "Real" ;
        TIPO_BOOLEANO = "Booleano" ;
        TIPO_CADENA = "Cadena" ;

        // Literales booleanos
        LIT_CIERTO = "cierto" ; LIT_FALSO = "falso" ;

        // Literales cadena
        LIT_NL = "nl"; LIT_TAB = "tab" ; LIT_COM = "com";

        // Palabras reservadas
        RES_CLASE = "clase" ;
        RES_EXTIENDE = "extiende" ;
        RES_METODO = "método" ;
        RES_CONSTRUCTOR = "constructor" ;
        RES_ATRIBUTO = "atributo" ;
        RES_ABSTRACTO = "abstracto" ;
        RES_PARAMETRO = "parámetro" ;
        RES_CONVERTIR = "convertir" ;
        RES_MIENTRAS = "mientras" ;
        RES_HACER = "hacer" ;
        RES_DESDE = "desde" ;
        RES_SI = "si" ;
        RES_OTRAS = "otras" ;
        RES_SALIR = "volver" ;
        RES_ESUN = "esUn" ;
        RES_SUPER = "super" ;

        // Operadores que empiezan con letras;
        OP_Y = "y" ;
        OP_O = "o" ;
        OP_NO = "no" ;

        // Los literales real y entero son "devueltos" en las
        // acciones del token "privado" LIT_NUMERO
        LIT_REAL ; LIT_ENTERO;

}

{
// Comienza la zona de código

        protected Token makeToken(int type)
        {
        // Usamos la implementación de la superclase...
        Token result = super.makeToken(type);
        // ...y añadimos información del nombre de fichero
```

```
            result.setFilename(inputState.filename);
            // y devolvemos el token
            return result;
            }
}

/**
* Los tres tipos de retorno de carro.
*/
protected NL :
(
("\r\n") => "\r\n" // MS-DOS
| '\r' // MACINTOSH
| '\n' // UNIX
)
{ newline(); }
;

/**
* Esta regla permite ignorar los blancos.
*/
protected BLANCO :
( ' '
| '\t'
| NL
) { $setType(Token.SKIP); } // La accion del blanco: ignorar
;

/**
* Letras españolas.
*/
protected LETRA
: 'a'..'z'
| 'A'..'Z'
| 'ñ' | 'Ñ'
| 'á' | 'é' | 'í' | 'ó' | 'ú'
| 'Á' | 'É' | 'Í' | 'Ó' | 'Ú'
| 'ü' | 'Ü'
;

/**
* Dígitos usuales
*/
protected DIGITO : '0'..'9';

/**
* Regla que permite reconocer los literales
* (y palabras reservadas).
*/
IDENT
options {testLiterals=true;} // Comprobar palabras reservadas
:
(LETRA|'_') (LETRA|DIGITO|'_')*
;

// Separadores
PUNTO_COMA : ';' ;
COMA : ',' ;
CORCHETE_AB : '[' ;
CORCHETE_CE : ']' ;
LLAVE_AB : '{' ;
LLAVE_CE : '}' ;
PUNTO : '.' ;
PARENT_AB : '(' ;
PARENT_CE : ')' ;
BARRA_VERT : '|' ;
// operadores
```

```
OP_IGUAL : "==" ;
OP_DISTINTO : "!=" ;
OP_ASIG : '=' ;
OP_MENOR : '<' ;
OP_MAYOR : '>' ;
OP_MENOR_IGUAL : "<=" ;
OP_MAYOR_IGUAL : ">=" ;
OP_MAS : '+' ;
OP_MENOS : '-' ;
OP_MASMAS : "++" ;
OP_MENOSMENOS : "--" ;
OP_PRODUCTO : '*' ;
OP_DIVISION : '/' ;

/**
* Permite reconocer literales enteros y reales sin generar conflictos.
*/
LIT_NUMERO : (( DIGITO )+ '.' ) =>
( DIGITO )+ '.' ( DIGITO )* { $setType (LIT_REAL); }
| ( DIGITO )+ { $setType (LIT_ENTERO); }
;

/**
* Comentario de una sola línea
*/
protected COMENTARIO1
:
"//" (~('\n'|'\r'))*
{ $setType(Token.SKIP); }
;

/**
* Comentario de varias líneas
*/
protected COMENTARIO2 :
"/*"
( ('*' NL) => '*' NL
| ('*' ~('/'|'\n'|'\r')) => '*' ~('/'|'\n'|'\r')
| NL
| ~( '\n' | '\r' | '*' )
)*
"*/"
{ $setType(Token.SKIP); }
;

/**
* Literales cadena
*/
LIT_CADENA :
"" !
( ~(""|'\n'|'\r') )*
"" !
;
```