Capítulo 2

Análisis lexicográfico

2.1 Visión general

Este capítulo estudia la primera fase de un compilador, es decir su análisis lexicográfico, también denominado abreviadamente análisis léxico. Las técnicas utilizadas para construir analizadores léxicos también se pueden aplicar a otras áreas como, por ejemplo, a lenguajes de consulta y sistemas de recuperación de información. En cada aplicación, el problema de fondo es la especificación y diseño de programas que ejecuten las acciones activadas por palabras que siguen ciertos patrones dentro de las cadenas a reconocer. Como la programación dirigida por patrones está ampliamente extendida y resulta de indudable utilidad, existen numerosos metalenguajes que permiten establecer pares de la forma patrón-acción, de manera que la acción se ejecuta cada vez que el sistema se encuentra una serie de caracteres cuya estructura coincide con la del patrón. En concreto, vamos a estudiar Lex con el objetivo de especificar los analizadores léxicos. En este lenguaje, los patrones se especifican por medio de expresiones regulares, y el metacompilador de Lex genera un reconocedor de las expresiones regulares mediante un autómata finito (determinista evidentemente) eficiente.

Por otro lado, una herramienta software que automatiza la construcción de analizadores léxicos permite que personas con diferentes conocimientos utilicen la concordancia de patrones en sus propias áreas de aplicación, ya que, a la hora de la verdad, no es necesario tener profundos conocimientos de informática para aplicar dichas herramientas.

2.2 Concepto de analizador léxico

Se encarga de buscar los componentes léxicos o palabras que componen el programa fuente, según unas reglas o patrones.

La entrada del analizador léxico podemos definirla como una secuencia de caracteres, que pueda hallarse codificada según cualquier estándar: ASCII (*American Standard Code for Information Interchange*), EBCDIC (*Extended Binary Coded Decimal Interchange Code*), Unicode, etc. El analizador léxico divide esta secuencia en palabras con significado propio y después las convierte a una secuencia de terminales desde el punto de vista del analizador sintáctico. Dicha secuencia es el punto de partida para que el analizador sintáctico construya el árbol sintáctico que reconoce la/s sentencia/s de entrada, tal y como puede verse en la figura 2.1.

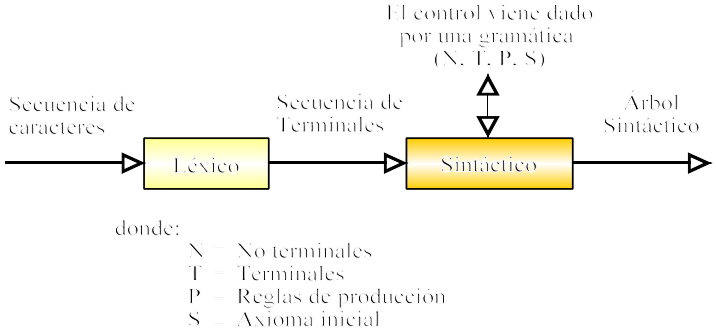


Figure 1 Entradas y salidas de las dos primeras fases de la etapa de análisis. La frase "Secuencia de Terminales" hace referencia a la gramática del sintáctico; pero también es posible considerar que dicha secuencia es de no terminales si usamos el punto de vista del lexicográfico.

El analizador léxico reconoce las palabras en función de una gramática regular de manera que el alfabeto Σ de dicha gramática son los distintos caracteres del juego de caracteres del ordenador sobre el que se trabaja (que forman el conjunto de símbolos terminales), mientras que sus no terminales son las categorías léxicas en que se integran las distintas secuencias de caracteres. Cada no terminal o categoría léxica de la gramática regular del análisis léxico es considerado como un terminal de la gramática de contexto libre con la que trabaja el analizador sintáctico, de manera que la salida de alto nivel (no terminales) de la fase léxica supone la entrada de bajo nivel (terminales) de la fase sintáctica. En el caso de Lex, por ejemplo, la gramática regular se expresa mediante expresiones regulares.

2.2.1 Funciones del analizador léxico

El analizador léxico es la primera fase de un compilador. Su principal función

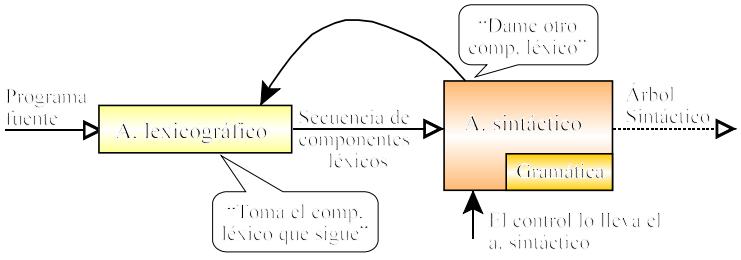


Figure 2 La fase de análisis léxico se halla bajo el control del análisis sintáctico. Normalmente se implementa como una función de éste

consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. Esta interacción suele aplicarse convirtiendo al analizador léxico en una subrutina o corrutina del analizador sintáctico. Recibida la orden "Dame el siguiente componente léxico"del analizador sintáctico, el léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico, el cual devuelve al sintáctico según el formato convenido (ver figura 2.2).

Además de esta función principal, el analizador léxico también realiza otras de gran importancia, a saber:

- Eliminar los comentarios del programa.
- Eliminar espacios en blanco, tabuladores, retorno de carro, etc, y en general, todo aquello que carezca de significado según la sintaxis del lenguaje.
- Reconocer los identificadores de usuario, números, palabras reservadas del lenguaje, etc., y tratarlos correctamente con respecto a la tabla de símbolos (solo en los casos en que este analizador deba tratar con dicha estructura).
- Llevar la cuenta del número de línea por la que va leyendo, por si se produce algún error, dar información acerca de dónde se ha producido.
- Avisar de errores léxicos. Por ejemplo, si el carácter '@' no pertenece al lenguaje, se debe emitir un error.
- También puede hacer funciones de preprocesador.

2.2.2 Necesidad del analizador léxico

Un buen profesional debe ser capaz de cuestionar y plantearse todas las decisiones de diseño que se tomen, y un asunto importante es el porqué se separa el análisis léxico del sintáctico si, al fin y al cabo, el control lo va a llevar el segundo. En otras palabras, por qué no se delega todo el procesamiento del programa fuente sólo en el análisis sintáctico, cosa perfectamente posible (aunque no plausible como veremos a continuación), ya que el sintáctico trabaja con gramáticas de contexto libre y éstas engloban a la regulares. A continuación estudiaremos algunas razones de esta separación.

2.2.2.1 Simplificación del diseño

Un diseño sencillo es quizás la ventaja más importante. Separar el análisis léxico del análisis sintáctico a menudo permite simplificar una, otra o ambas fases. Normalmente añadir un analizador léxico permite simplificar notablemente el analizador sintáctico. Aún más, la simplificación obtenida se hace especialmente patente cuando es necesario realizar modificaciones o extensiones al lenguaje inicialmente ideado; en otras palabras, se facilita el mantenimiento del compilador a medida que el lenguaje evoluciona.

La figura 2.3 ilustra una situación en la que, mediante los patrones

correspondientes, el analizador léxico reconoce números enteros y operadores aritméticos. A la hora de construir una primera versión del analizador sintáctico, podemos asumir que dos expresiones pueden ir conectadas con cualquiera de dichos operadores, por lo que se puede optar por agruparlos todos bajo la categoría léxica **OPARIT** (Operadores ARITméticos). En una fase posterior, puede resultar necesario disgregar dicha categoría en tantas otras como operadores semánticamente diferentes haya: **OPARIT** desaparece y aparecen **MAS**, **MENOS**, **MULT** y **DIV**. Una modificación tal resulta trivial si se han separado adecuadamente ambos analizadores, ya que consiste en sustituir el patrón agrupado (‘-‘|’+’|’*’|’/’) por los patrones disgregados ‘-‘, ‘+’, ‘*’ y ‘/’.

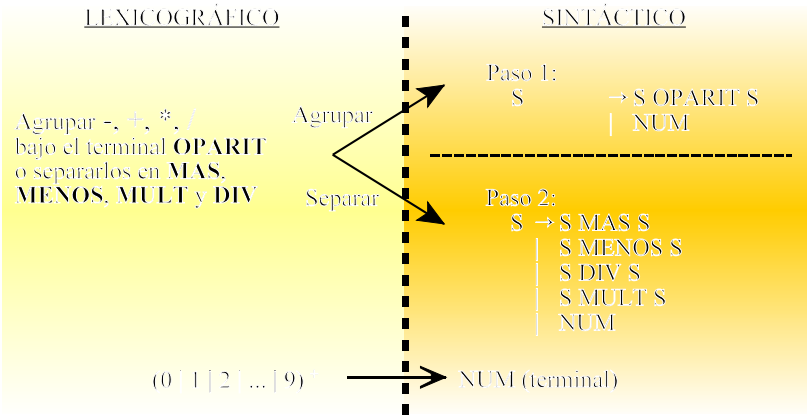


Figure 3 Pasos en la construcción progresiva de un compilador

Si el sintáctico tuviera la gramática del paso 1, el lexicográfico sería:
el patrón (0 | 1 | 2 | ... | 9)$^+$ retorna la categoría **NUM**
el patrón (‘+’ | ‘-‘ | ‘*’ | ‘/’) retorna la categoría **OPARIT**

En cambio, si el sintáctico adopta el paso 2, el lexicográfico sería:
el patrón (0 | 1 | 2 | ... | 9)$^+$ retorna la categoría **NUM**
el patrón ‘+’ retorna la categoría **MAS**
el patrón ‘-’ retorna la categoría **MENOS**
el patrón ‘*’ retorna la categoría **MULT**
el patrón ‘/’ retorna la categoría **DIV**

También se puede pensar en eliminar el lexicográfico, incorporando su gramática en la del sintáctico, de manera que éste vería incrementado su número de reglas con las siguientes:

NUM → 0
| 1
| 2

| 3
...
| NUM NUM

sin embargo, los autómatas destinados a reconocer los componentes léxicos y al árbol sintáctico son radicalmente diferentes, tanto en su concepción como en su implementación lo que, de nuevo, nos lleva a establecer una división entre estos análisis.

A modo de conclusión, se puede decir que es muy recomendable trabajar con dos gramáticas, una que se encarga del análisis léxico y otra que se encarga del análisis sintáctico. ¿Dónde se pone el límite entre lo que reconoce una y otra gramática?, ¿qué se considera un componente básico?, ¿cuántas categorías gramaticales se establecen? Si se crean muchas categorías gramaticales se estará complicando la gramática del sintáctico, como ocurre p.ej. en el paso 2. En general, deben seguirse un par de reglas básicas para mantener la complejidad en unos niveles admisibles. La primera es que la información asociada a cada categoría léxica debe ser la necesaria y suficiente, lo que quedará más claro en capítulos posteriores, cuando se conozca el concepto de **atributo**. La segunda es que, por regla general, las gramáticas que se planteen (regular y de contexto libre) no deben verse forzadas, en el sentido de que los distintos conceptos que componen el lenguaje de programación a compilar deben formar parte de una o de otra de forma natural; p.ej., el reconocimiento que deba hacerse carácter a carácter (sin que éstos tengan un significado semántico por sí solos) debe formar parte del análisis léxico.

2.2.2.2 Eficiencia

La división entre análisis léxico y sintáctico también mejora la eficiencia del compilador. Un analizador léxico independiente permite construir un procesador especializado y potencialmente más eficiente para las funciones explicadas en el epígrafe 2.2.1. Gran parte del tiempo de compilación se invierte en leer el programa fuente y dividirlo en componentes léxicos. Con técnicas especializadas de manejo de *buffers* para la lectura de caracteres de entrada y procesamiento de patrones se puede mejorar significativamente el rendimiento de un compilador.

2.2.2.3 Portabilidad

Se mejora la portabilidad del compilador, ya que las peculiaridades del alfabeto de partida, del juego de caracteres base y otras anomalías propias de los dispositivos de entrada pueden limitarse al analizador léxico. La representación de símbolos especiales o no estándares, como '↑' en Pascal, se pueden aislar en el analizador léxico.

Por otro lado, algunos lenguajes, como APL (A Program Language) se benefician sobremanera del tratamiento de los caracteres que forman el programa de entrada mediante un analizador aislado. El Diccionario de Informática de la *Oxford University Press* define este lenguaje de la siguiente forma:

«*... Su característica principal es que proporciona un conjunto muy grande de operadores importantes para tratar las órdenes multidimensionales junto con la capacidad del usuario de definir sus propios operadores. Los operadores incorporados se encuentran representados, principalmente, por caracteres solos que utilizan un conjunto de caracteres especiales. De este modo, los programas APL son muy concisos y, con frecuencia, impenetrables*».

2.2.2.4 Patrones complejos

Otra razón por la que se separan los dos análisis es para que el analizador léxico se centre en el reconocimiento de componentes básicos complejos. Por ejemplo en Fortran, existe el siguiente par de proposiciones muy similares sintácticamente, pero de significado bien distinto:

```
DO5I = 2.5      (Asignación del valor 2.5 a la variable DO5I)
DO 5 I = 2, 5   (Bucle que se repite para I = 2, 3, 4 y 5)
```

En este lenguaje los espacios en blancos no son significativos fuera de los comentarios y de un cierto tipo de cadenas (para ahorrar espacio de almacenamiento, en una época de la Informática en la que éste era un bien escaso), de modo que supóngase que todos los espacios en blanco eliminables se suprimen antes de comenzar el análisis léxico. En tal caso, las proposiciones anteriores aparecerían ante el analizador léxico como:

```
DO5I=2.5
DO5I=2,5
```

El analizador léxico no sabe si DO es una palabra reservada o es el prefijo del nombre de una variable hasta que se lee la coma. Ha sido necesario examinar la cadena de entrada mucho más allá de la propia palabra a reconocer haciendo lo que se denomina *lookahead* (o prebúsqueda). La complejidad de este procesamiento hace recomendable aislarlo en una fase independiente del análisis sintáctico.

En cualquier caso, en lenguajes como Fortran primero se diseñó el lenguaje y luego el compilador, lo que conllevó problemas como el que se acaba de plantear. Hoy día los lenguajes se diseñan teniendo en mente las herramientas de que se dispone para la construcción de su compilador y se evitan este tipo de situaciones.

2.3 *Token*, patrón y lexema

Desde un punto de vista muy general, podemos abstraer el programa que implementa un análisis léxicográfico mediante una estructura como:

```
(Expresión regular)1    {acción a ejecutar}1

(Expresión regular)2    {acción a ejecutar}2
(Expresión regular)3    {acción a ejecutar}3
...                     ...
(Expresión regular)n    {acción a ejecutar}n
```

donde cada acción a ejecutar es un fragmento de programa que describe cuál ha de ser

la acción del analizador léxico cuando la secuencia de entrada coincida con la expresión regular. Normalmente esta acción suele finalizar con la devolución de una categoría léxica.

Todo esto nos lleva a los siguientes conceptos de fundamental importancia a lo largo de nuestro estudio:

- Patrón: es una expresión regular.
- *Token*: es la categoría léxica asociada a un patrón. Cada *token* se convierte en un número o código identificador único. En algunos casos, cada número tiene asociada información adicional necesaria para las fases posteriores de la etapa de análisis. El concepto de *token* coincide directamente con el concepto de terminal desde el punto de vista de la gramática utilizada por el analizador sintáctico.
- Lexema: Es cada secuencia de caracteres concreta que encaja con un patrón. P.ej: "8", "23" y "50" son algunos lexemas que encajan con el patrón ('0'|'1'|'2'| ... |'9')$^+$. El número de lexemas que puede encajar con un patrón puede ser finito o infinito, p.ej. en el patrón 'W''H''I''L''E' sólo encaja el lexema "WHILE".

Una vez detectado que un grupo de caracteres coincide con un patrón, se considera que se ha detectado un lexema. A continuación se le asocia el número de su categoría léxica, y dicho número o *token* se le pasa al sintáctico junto con información adicional, si fuera necesario. En la figura 1.13 puede verse cómo determinadas categorías llevan información asociada, y otras no.

Por ejemplo, si se necesita construir un analizador léxico que reconozca los números enteros, los números reales y los identificadores de usuario en minúsculas, se puede proponer una estructura como:

Expresión Regular	Terminal asociado
(0 ... 9)$^+$	NUM_ENT
(0 ... 9)*. (0 ... 9)$^+$	NUM_REAL
(a ... z) (a ... z 0 ... 9)*	ID

Asociado a la categoría gramatical de número entero se tiene el *token* **NUM_ENT** que puede equivaler, p.ej. al número 280; asociado a la categoría gramatical número real se tiene el *token* **NUM_REAL** que equivale al número 281; y la categoría gramatical identificador de usuario tiene el *token* ID que equivale al número 282. Así, la estructura expresión regular-acción sería la siguiente (supuesto que las acciones las expresamos en C o Java):

(0 ... 9)$^+$	{ return 280;}
(0 ... 9)*. (0 ... 9)$^+$	{ return 281;}
(a ... z) (a ... z 0...9)*	{ return 282;}
" "	{ }

De esta manera, un analizador léxico que obedeciera a esta estructura, si

durante su ejecución se encuentra con la cadena:

95.7 99 cont

intentará leer el lexema más grande de forma que, aunque el texto “95" encaja con el primer patrón, el punto y los dígitos que le siguen “.7" hacen que el analizador decida reconocer “95.7" como un todo, en lugar de reconocer de manera independiente “95" por un lado y “.7" por otro; así se retorna el *token* **NUM_REAL**. Resulta evidente que un comportamiento distinto al expuesto sería una fuente de problemas. A continuación el patrón “ ” y la acción asociada permiten ignorar los espacios en blanco. El “99" coincide con el patrón de **NUM_ENT**, y la palabra “cont” con **ID**.

Para facilitar la comprensión de las acciones asociadas a los patrones, en vez de trabajar con los números 280, 281 y 282 se definen mnemotécnicos.

```
# define NUM_ENT 280
# define NUM_REAL 281
# define ID 282
(“ ”\t \n)
(0 ... 9)+                    {return NUM_ENT;}
(0 ... 9)*. (0 ... 9)+        {return NUM_REAL;}
(a ... z) (a ... z 0 ... 9)*  {return ID;}
```

En esta nueva versión, los lexemas que entran por el patrón (“ ”\t \n) no tienen acción asociada, por lo que, por defecto, se ejecuta la acción nula o vacía.

El asociar un número (*token*) a cada categoría gramatical se suele emplear mucho en los metacompiladores basados en lenguajes puramente imperativos, como pueda ser C o Pascal. Los metacompiladores más modernos basados en programación orientada a objetos también asocian un código a cada categoría gramatical, pero en lugar de retornar dicho código, retornan objetos con una estructura más compleja.

2.3.1 Aproximaciones para construir un analizador lexicográfico

Hay tres mecanismos básicos para construir un analizador lexicográfico:

- Ad hoc. Consiste en la codificación de un programa reconocedor que no sigue los formalismos propios de la teoría de autómatas. Este tipo de construcciones es muy propensa a errores y difícil de mantener.
- Mediante la implementación manual de los autómatas finitos. Este mecanismo consiste en construir los patrones necesarios para cada categoría léxica, construir sus automátas finitos individuales, fusionarlos por opcionalidad y, finalmente, implementar los autómatas resultantes. Aunque la construcción de analizadores mediante este método es sistemática y no propensa a errores, cualquier actualización de los patrones reconocedores implica la modificación del código que los implementa, por lo que el mantenimiento se hace muy costoso.

- Mediante un metacompilador. En este caso, se utiliza un programa especial que tiene como entrada pares de la forma (expresión regular, acción). El metacompilador genera todos los autómatas finitos, los convierte a autómata finito determinista, y lo implementa en C. El programa C así generado se compila y se genera un ejecutable que es el análizador léxico de nuestro lenguaje. Por supuesto, existen metacompiladores que generan código Java, Pascal, etc. en lugar de C.

Dado que hoy día existen numerosas herramientas para construir analizadores léxicos a partir de notaciones de propósito especial basadas en expresiones regulares, nos basaremos en ellas para proseguir nuestro estudio dado que, además, los analizadores resultantes suelen ser bastante eficientes tanto en tiempo como en memoria. Comenzaremos con un generador de analizadores léxicos escritos en C, y continuaremos con otro que genera código Java.

2.4 El generador de analizadores lexicográficos: PCLex

En esta sección se describe la herramienta actualmente más extendida, llamada Lex, para la especificación de analizadores léxicos en general, aunque en nuestro caso nos centraremos en el analizador léxico de un compilador. La herramienta equivalente para entorno DOS se denomina PCLex, y la especificación de su entrada, lenguaje Lex. El estudio de una herramienta existente permitirá mostrar cómo, utilizando expresiones regulares, se puede combinar la especificación de patrones con acciones, para p.ej., realizar inserciones en una tabla de símbolos.

2.4.1 Visión general

Como ya sabemos, las reglas de reconocimiento son de la forma:

p_1 {acción$_1$}
p_2 {acción$_2$}
... ...
p_n {acción$_n$}

donde p_i es una expresión regular y acción$_i$ es un fragmento de programa que describe cuál ha de ser la acción del analizador léxico cuando se encuentra con un lexema que encaja por p_i. En Lex, las acciones se escriben en C, aunque existen multitud de metacompiladores similares al PCLex que permiten codificar en otros lenguajes, como por ejemplo Jflex que utiliza el lenguaje Java.

Un analizador léxico creado por PCLex está diseñado para comportarse en sincronía con un analizador sintáctico. Para ello, entre el código generado existe una función llamada **yylex()** que, al ser invocada (normalmente por el sintáctico), comienza a leer la entrada, carácter a carácter, hasta que encuentra el mayor prefijo en la entrada que concuerde con una de las expresiones regulares p_i; dicho prefijo constituye un lexema y, una vez leído, se dice que "ha sido consumido" en el sentido de que el punto de lectura ha avanzado hasta el primer carácter que le sigue. A continuación **yylex()**

ejecuta la acción$_i$. Generalmente esta acción$_i$ devolverá el control al analizador sintáctico informándole del *token* encontrado. Sin embargo, si la acción no tiene un **return**, el analizador léxico se dispondrá a encontrar un nuevo lexema, y así sucesivamente hasta que una acción devuelva el control al analizador sintáctico, o hasta llegar al final de la cadena, representada por el carácter EOF (*End Of File*). La búsqueda repetida de lexemas hasta encontrar una instrucción **return** explícita permite al analizador léxico procesar espacios en blanco y comentarios de manera apropiada.

Antes de ejecutar la acción asociada a un patrón, **yylex()** almacena el lexema leído en la variable **yytext**. Cualquier información adicional que se quiera comunicar a la función llamante (normalmente el analizador sintáctico), además del *token* debe almacenarse en la variable global **yylval**, cuyo tipo se verá más adelante.

Los programas que se obtienen con PCLex son relativamente grandes, (aunque muy rápidos también), aunque esto no suele ser un problema ante las enormes cantidades de memoria que se manejan hoy día. La gran ventaja de PCLex es que permite hacer analizadores complejos con bastante rapidez.

2.4.2 Creación de un analizador léxico

Como ya se ha comentado, PCLex tiene su propio lenguaje, al que llamaremos Lex y que permite especificar la estructura abstracta de un analizador léxico, tanto en lo que respecta a las expresiones regulares como a la acción a tomar al encontrar un lexema que encaje en cada una de ellas.

Los pasos para crear un analizador léxico con esta herramienta son (figura 2.4):

- Construir un fichero de texto en lenguaje Lex que contiene la estructura abstracta del analizador.
- Metacompilar el fichero anterior con PCLex. Así se obtendrá un fichero fuente en C estándar. Algunas veces hay que efectuar modificaciones directas en este código, aunque las últimas versiones de PCLex han disminuido al máximo estas situaciones.
- Compilar el fuente en C generado por PCLex con un compilador C, con lo que obtendremos un ejecutable que sigue los pasos descritos en el epígrafe 2.4.1.

Figure 4 Obtención de un programa ejecutable a partir de una especificación en Lex

Si ejecutamos **prog.exe** la aplicación se quedará esperando a que se le introduzca la cadena de entrada por teclado para proceder a su partición en lexemas;

el fin de fichero se introduce pulsando las teclas Ctrl y Z (en pantalla aparece **^Z**). Así el programa parte la cadena de entrada en los lexemas más largos posible, y para cada una de ellos ejecuta la acción asociada al patrón por el que encaje. Por ejemplo, si el fuente en Lex es de la forma:

```
[0-9] + {printf("numero");}
[A-Z]+ {printf("palabra");}
```

y tecleamos :

```
HOLA 23 ^z
```

cuando un lexema casa con un patrón, **yylex()** cede el control a su acción. De esta forma se obtiene por pantalla lo siguiente:

```
palabra numero
```

La palabra "HOLA" encaja por el segundo patrón y la palabra "23" encaja por el primero. Nótese que no hemos especificado ningún patrón sobre los espacios en blanco; por defecto, la función **yylex()** generada cuando se encuentra un carácter o secuencia de caracteres que no casa con ningún patrón, sencillamente lo visualiza por la salida estándar y continúa a reconocer el siguiente lexema.

Para no tener que introducir la cadena fuente por teclado, sino que sea reconocida desde un fichero, podemos, o bien redirigir la entrada con:

```
prog < file.pas > salida.txt
```

donde:

```
< file.pas        redirige la entrada y
> salida.txt      redirige la salida,
```

o bien hacer uso de las variables **yyin** e **yyout** de tipo ***FILE** para inicializarlas a los ficheros correspondientes. Estas variables son suministradas por el código generado por PCLex, e inicialmente apuntan a **stdin** y **stdout** respectivamente.

2.4.3 El lenguaje Lex

Un programa Lex tiene la siguiente estructura:

```
Área de definiciones Lex
%%              /* es lo único obligatorio en todo el programa */
Área de reglas
%%
Área de funciones
```

siendo el mínimo programa que se puede construir en Lex:

```
%%
```

En el área de reglas se definen los patrones de los lexemas que se quieren buscar a la entrada, y al lado de tales expresiones regulares, se detallan (en C) las acciones a ejecutar tras encontrar una cadena que se adapte al patrón indicado. Los separadores "%%" deben ir obligatoriamente en la columna 0, al igual que las reglas y demás información propia del lenguaje Lex.

Como ya se indicó, en Lex, si una cadena de entrada no encaja con ningún patrón, la acción que se toma es escribir tal entrada en la salida. Por tanto, como el

programa %% no especifica ningún patron, pues el analizador léxico que se genera lo único que hace es copiar la cadena de entrada en la salida que, por defecto, es la salida estándar.

2.4.3.1 Premisas de Lex para reconocer lexemas

El **yylex()** generado por PCLex sigue dos directrices fundamentales para reconocer lexemas en caso de ambigüedad. Estas directrices son, por orden de prioridad:

1. Entrar siempre por el patrón que reconoce el lexema más largo posible.
2. En caso de conflicto usa el patrón que aparece en primera posición.

Como consecuencia de la segunda premisa: los patrones que reconocen palabras reservadas se colocan siempre antes que el patrón de identificador de usuario. P.ej. un analizador léxico para Pascal (en el que TYPE y VAR son palabras reservadas), podría tener una apariencia como:

```
%%
“TYPE”
“VAR”
[A-Z][A-Z0-9]*
...
```

Cuando **yylex()** se encuentra con la cadena “VAR” se produce un conflicto, (ya que dicho lexema puede entrar tanto por el patrón segundo, como por el tercero). Entonces toma el patrón que aparece antes, que en el ejemplo sería “VAR”, reconociéndose al lexema como palabra reservada. Cambiar el orden de ambos patrones tiene consecuencias funestas:

```
%%
“TYPE”
[A-Z][A-Z0-9]*
“VAR”
...
```

ya que esta vez el lexema “VAR” entraría por el patrón de identificador de usuario, y jamás se reconocería como una palabra reservada: el patrón “VAR” es superfluo.

2.4.3.2 Caracteres especiales de Lex

Lex se basa en el juego de caracteres ASCII para representar las expresiones regulares, por lo que los caracteres que veremos a continuación tienen un significado especial con tal propósito:

“”: sirve para encerrar cualquier cadena de literales. Por regla general no es necesario encerrar los literales entre comillas a no ser que incluyan símbolos especiales, esto es, el patrón “WHILE” y el patrón WHILE son equivalentes; pero para representar p.ej. el inicio de comentario en Modula-2 sí es necesario entrecomillar los caracteres que componen al patrón: “(*”, ya que éste contiene, a su vez, símbolos especiales.

\: hace literal al siguiente carácter. Ej.: \" reconoce unas comillas. También se utiliza para expresar aquellos caracteres que no tienen representación directa por pantalla: \n para el retorno de carro, \t para el tabulador, etc.

nºoctal: representa el carácter cuyo valor ASCII es **nº octal**. P.ej: \012 reconoce el carácter decimal 10 que se corresponde con LF (*Line Feed*).

[ ]: permiten especificar listas de caracteres, o sea uno de los caracteres que encierra, ej.: [abc] reconoce o la 'a', o la 'b', o la 'c', ([abc] ≡ (a|b|c)). Dentro de los corchetes los siguientes caracteres también tienen un sentido especial:

- -: indica rango. Ej.: [A-Z0-9] reconoce cualquier carácter de la 'A' a la 'Z' o del '0' a '9'.
- ^: indica compleción cuando aparece al comienzo, justo detrás de "[". Ej.: [^abc] reconoce cualquier carácter excepto la 'a', la 'b' o la 'c'. Ej.: [^A-Z] reconoce cualquier carácter excepto los de la 'A' a la 'Z'.

?: aquello que le precede es opcional[1]. Ej.: a? ≡ (a | ε). Ej.: [A-Z]? reconoce cualquier letra de la 'A' a la 'Z' o bien ε. Ej.: a?b ≡ ab | εb.

.: representa a cualquier carácter (pero sólo a uno) excepto el retorno de carro (\n). Es muy interesante porque nos permite recoger cualquier otro carácter que no sea reconocido por los patrones anteriores.

|: indica opcionalidad (*OR*). Ej.: a|b reconoce a la 'a' o a la 'b'. Ej.: .|\n reconoce cualquier carácter. Resulta curioso el patrón (.|\n)* ya que por aquí entra el programa entero, y como **yylex()** tiene la premisa de reconocer el lexema más largo, pues probablemente ignorará cualquier otro patrón. También resulta probable que durante el reconocimiento se produzca un error por desbordamiento del espacio de almacenamiento de la variable **yytext** que, recordemos, almacena el lexema actual.

*: indica repetición 0 o más veces de lo que le precede.

+: indica repetición 1 o más veces de lo que le precede.

( ): permiten la agrupación (igual que en las expresiones aritméticas).

{ }: indican rango de repetición. Ej.: a{1,5} ≡ aa?a?a?a? Las llaves vienen a ser algo parecido a un * restringido. También nos permite asignarle un nombre a una expresión regular para reutilizarla en múltiples patrones; esto se verá más adelante.

2.4.3.3 Caracteres de sensibilidad al contexto

[1] La expresión ε representa a la cadena vacía.

Lex suministra ciertas capacidades para reconocer patrones que no se ajustan a una expresión regular, sino más bien a una gramática de contexto libre. Esto es especialmente útil en determinadas circunstancias, como p.ej. para reconocer los comentarios de final de línea, admitidos por algunos lenguajes de programación (los que suelen comenzar por // y se extienden hasta el final de la línea actual.

$: el patrón que le precede sólo se reconoce si está al final de la línea. Ej.: (a|b|cd)$. Que el lexema se encuentre al final de la línea quiere decir que viene seguido por un retorno de carro, o bien por EOF. El carácter que identifica el final de la línea no forma parte del lexema.

^: fuera de los corchetes indica que el patrón que le sucede sólo se reconoce si está al comienzo de la línea. Que el lexema se encuentre al principio de la línea quiere decir que viene precedido de un retorno de carro, o que se encuentra al principio del fichero. El retorno de carro no pasa a formar parte del lexema. Nótese como las dos premisas de Lex hacen que los patrones de sensibilidad al contexto deban ponerse en primer lugar en caso de que existan ambigüedades:

Programa 1	Programa 2
^"casa"	"casa"
"casa"	^"casa"

En ambos programas el lexema "casa" cuando se encuentra al comienzo de línea puede entrar por ambos patrones luego, al existir ambigüedad, Lex selecciona el primero. Por ello, nótese cómo en el primer programa se entra por el patrón adecuado mientras que en el segundo se entra por el patrón general con lo que nunca se hará uso del patrón ^"casa".

/: Reconoce el patrón que le precede si y sólo si es prefijo de una secuencia simple como la que le sucede. Ej.: ab/c; en este caso si a la entrada se tiene "abcd", se reconocería el lexema "ab" porque está sucedido de 'c'. Si la entrada fuera "abdc" el patrón no se aplicaría. Por otro lado, un patrón como ab/c+ es erróneo puesto que el patrón c+ no se considera simple.

2.4.3.4 Estado léxicos

Los estados léxicos vienen a ser como variables lógicas excluyentes (una y sólo una puede estar activa cada vez) que sirven para indicar que un patrón sólo puede aplicarse si el estado léxico que lleva asociado se encuentra activado. En los ejemplos que hemos visto hasta ahora hemos trabajado con el estado léxico por defecto que, también por defecto, se encuentra activo al comenzar el trabajo de **yylex()**. Por ser un estado léxico por defecto no hemos tenido que hacer ninguna referencia explícita a él.

Los distintos estados léxicos (también llamados condiciones *start*) se declaran en el área de definiciones Lex de la forma:

%START id_1, id_2, ...

Una acción asociada a un patrón puede activar un estado léxico ejecutando la

macro **BEGIN**, que es suministrada por el programa generado por PCLex. Así:

```
BEGIN idi;
```

activaría la condición *start* id_i, que ha debido ser previamente declarada. La activación de un estado léxico produce automáticamente la desactivación de todos los demás. Por otro lado, el estado léxico por defecto puede activarse con:

```
BEGIN 0;
```

Para indicar que un patrón sólo es candidato a aplicarse en caso de que se encuentre activa la condición *start* id_i, se le antepone la cadena "<id_i>". Si un patrón no tiene asociada condición *start* explícita, se asume que tiene asociado el estado léxico por defecto.

El siguiente ejemplo muestra cómo visualizar todos los nombres de los procedimientos y funciones de un programa Modula-2:

```
%START PROC
%%
"PROCEDURE" {BEGIN PROC;}
<PROC> [a-zA-Z][a-zA-Z0-9]* {     printf ("%s\n", yytext);
                                  BEGIN 0 ; }
```

2.4.3.5 Área de definiciones y área de funciones

El área de definiciones de un programa Lex tiene tres utilidades fundamentales:

a) Definir los estados léxicos.
b) Asignar un nombre a los patrones más frecuentes.
c) Poner código C que será global a todo el programa.

Ya hemos visto cómo definir estados léxicos, por lo que pasaremos a estudiar las otras dos posibilidades.

En el área de definiciones podemos crear expresiones regulares auxiliares de uso frecuente y asignarles un nombre. Posteriormente, estos patrones pueden ser referenciados en el área de reglas sin más que especificar su nombre entre llaves: {}. Por ejemplo, los siguientes dos programas son equivalentes, pero el primero es más claro y sus reglas más concisas:

Programa 1

```
D       [0-9]
L       [a-zA-Z]
%%
{D}+
{L}({L}|{D})*
```

Programa 2

```
%%
[0-9]+
[a-zA-Z][a-zA-Z0-9]*
```

Recordemos que en Lex, todo debe comenzar en la primera columna. Si algo no comienza en dicha columna, PCLex lo pasa directamente al programa C generado, sin procesarlo. De esta forma es posible crear definiciones de variables, etc. Sin embargo, esto no se recomienda ya que Lex posee un bloque específico para esto en el que se pueden poner incluso directivas del procesador, que deben comenzar

obligatoriamente en la primera columna.

Así, el área de definiciones también permite colocar código C puro que se trasladará tal cual al comienzo del programa en C generado por PCLex. Para ello se usan los delimitadores "%{" y "%}". Ej.:

```
%{
#include <stdlib.h>
typedef struct _Nodo{
        int valor;
        Nodo * siguiente;
} Nodo;
Nodo * tablaDeSimbolos;
%}
```

Es normal poner en esta zona las declaraciones de variables globales utilizadas en las acciones del área de reglas y un **#include** del fichero que implementa la tabla de símbolos.

Probablemente, el lector se estará preguntando qué contiene la función **main()** del programa C que genera PCLex. La verdad es que PCLex no genera **main()** alguno, sino que éste debe ser especificado por el programador. Por regla general, cuando se pretende ejecutar aisladamente un analizador léxico (sin un sintáctico asociado), lo normal es que las acciones asociadas a cada regla no contengan ningún **return** y que se incluya un **main()** que únicamente invoque a yylex():

```
void main(){
        yylex();
};
```

Para especificar el **main()** y cuantas funciones adicionales se estimen oportunas, se dispone del área de funciones. Dicha área es íntegramente copiada por PCLex en el fichero C generado. Es por ello que, a efectos prácticos, escribir código entre "%{" y "%}" en el área de definiciones es equivalente a escribirlo en el área de funciones.

El siguiente ejemplo informa de cuántas veces aparece el literal "resultado" en la cadena de entrada.

```
%{
        int cont=0;
%}
%%
"resultado"       {cont ++;}
. | \n     {;}
%%
void main(){
        yylex();
        printf("resultado aparece %d veces", cont);
}
```

2.4.3.6 Funciones y variables suministradas por PCLex

Como ya sabemos, el núcleo básico del programa en C generado por PCLex es la función **yylex()**, que se encarga de buscar un lexema y ejecutar su acción asociada. Este proceso lo realiza iterativamente hasta que en una de las acciones se encuentre un **return** o se acabe la entrada. Además, PCLex suministra otras funciones macros y variables de apoyo; a continuación se muestra la lista de las más interesantes por orden de importancia:

yylex(): implementa el analizador lexicográfico.

yytext: contiene el lexema actual

yyleng: número de caracteres del lexema actual.

yylval: es una variable global que permite la comunicación con el sintáctico. Realmente no la define PCLex sino la herramienta PCYacc que se estudiará en capítulos posteriores.

yyin: es de tipo *FILE, y apunta al fichero de entrada que se lee. Inicialmente apunta a **stdin**, por lo que el fichero de entrada coincide con la entrada estándar.

yyout: de tipo *FILE, apunta al fichero de salida (inicialmente **stdout**).

yyerror() : es una función que se encarga de emitir y controlar errores (saca mensajes de error por pantalla). Realmente es definida por PCYacc como se verá más adelante.

yylineno: variable de tipo entero que, curiosamente, debe ser creada, inicializada y actualizada por el programador. Sirve para mantener la cuenta del número de línea que se está procesando. Normalmente se inicializa a 1 y se incrementa cada vez que se encuentra un retorno de carro.

yywrap(): el algoritmo de **yylex()** llama automáticamente a esta macro cada vez que se encuentra con un EOF. La utilidad de esta macro reside en que puede ser redefinida (para ello hay que borrarla previamente con la directiva **#undef**). Esta macro debe devolver *false* (que en C se representa por el entero 0) en caso de que la cadena de entrada no haya finalizado realmente, lo que puede suceder en dos tipos de situaciones: a) se está procesando un fichero binario que puede contener por en medio el carácter EOF como uno más, en cuyo caso el verdadero final del fichero hay que descubrirlo contrastando la longitud del fichero con la longitud de la cadena consumida; y b) la cadena a reconocer se encuentra particionada en varios ficheros, en cuyo caso cuando se llega al final de uno de ellos, hay que cargar en **yyin** el siguiente y continuar el procesamiento. **yywrap()** debe devolver *true* (valor entero 1) en caso de que el EOF encontrado identifique realmente el final de la cadena a procesar.

yyless(int n): deja en **yytext** los **n** primeros caracteres del lexema actual. El resto los devuelve a la entrada, por lo que podemos decir que son des-consumidos. **yyleng** también se modifica convenientemente. Por ejemplo, el patrón abc*/\n es equivalente a:

```
abc*\n   { yyless(yyleng-1); }
```

input(): consume el siguiente carácter de la entrada y lo añade al lexema actual. P,ej., el programa:

```
%%
abc      { printf ("%s", yytext);
           input( );
           printf("%s", yytext);}
```

ante la entrada "abcde" entraría por este patrón: el lexema antes del **input()** (en el primer **printf**) es "abc", y después del **input** (en el segundo **printf**) es "abcd".

output(char c): emite el carácter **c** por la salida estándar. PCLex para DOS no soporta esta función, pero sí el Lex para Unix.

unput(char c): des-consume el carácter **c** y lo coloca al comienzo de la entrada. P.ej., supongamos que el programa:

```
%%
abc      { printf ("%s",yytext); unput('t'); }
tal      { printf ("%s",yytext); }
```

recibe la entrada "abcal". Los tres primeros caracteres del lexema coinciden con el primer patrón. Después queda en la entrada la cadena "al", pero como se ha hecho un **unput('t')**, lo que hay realmente a la entrada es "tal", que coincide con el segundo patrón.

ECHO: macro que copia la entrada en la salida (es la acción que se aplica por defecto a los lexemas que no encajan por ningún patrón explícito).

yymore(): permite pasar a reconocer el siguiente lexema, pero sin borrar el contenido actual de **yytext**, por lo que el nuevo lexema leído se concatena al ya existente. PCLex para DOS no soporta esta función, pero sí el Lex para Unix.

La utilidad principal de **yymore()** reside en que facilita el reconocer los literales entrecomillados. Supongamos que se desea construir un patrón para reconocer literales entrecomillados. Una primera idea podría ser el patrón:

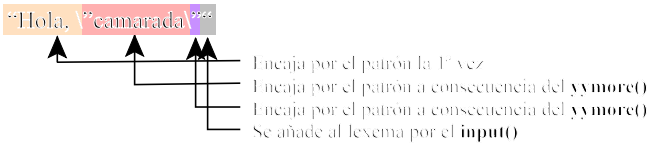
```
\"[^\"]*\"        /* Lexemas que empiecen por comillas,
                     cualquier cosa, y termine en comillas. */
```

que reconocería cadenas como "Hola", "adiós", e incluso "Ho;*". Sin embargo una cadena como "Hola, \"camarada\"" que, a su vez contiene comillas dentro, dará problemas porque la primera comilla de "camarada", es considerada como unas comillas de cierre. La solución a este problema pasa

por el siguiente bloque de código Lex:

```
\"[^\"]*/\"        { if (yytext[yyleng-1]=='\\')
                        yymore();
                    else
                        input();}
```

Este patrón reconoce cadenas entrecomilladas (excepto las últimas comillas), de forma que las cadenas que tienen dentro comillas se reconocen por partes. Siguiendo con el ejemplo de antes, "Hola, \"camarada\"" se reconocería como:



Este patrón fallaría si los caracteres anteriores a las comillas de cierre fuesen precisamente '\\'. Para solucionarlo habría que hacer uso de **unput** y de una variable global que actuase como *flag*. Cualquier sentencia que haya detrás de **yymore()** nunca se ejecutará, ya que actúa como una especie de **goto**.

REJECT: rechaza el lexema actual, lo devuelve a la entrada y busca otro patrón. **REJECT** le dice a **yylex()** que el lexema encontrado no corresponde realmente a la expresión regular en curso, y que busque la siguiente expresión regular a que corresponda. La macro **REJECT** debe ser lo último que una acción ejecute puesto que si hay algo detrás, no se ejecutará. P.ej. Para buscar cuántas veces aparecen las palabras "teclado" y "lado", se puede construir el siguiente programa Lex:

```
%{
        int t=0, l=0;
%}
%%
teclado         {t ++; REJECT;}
lado            {l ++;}
```

Ante la entrada "teclado" este programa se comporta de la siguiente forma:

1.- Entra por el patrón que lee el lexema más largo que sería "teclado" y ejecuta la acción asociada: se incrementa la variable **t** y se rechaza el lexema actual, por lo que el texto entero se devuelve a la entrada.

2.- Saca por pantalla "tec" que es la acción por defecto cuando se encuentran caracteres que no encajan con ningún patrón.

3.- El lexema "lado"' coincide con el segundo patrón y ejecuta la acción asociada: se incrementa la variable **l**.

Para finalizar, resulta evidente que el programador no debe declarar ninguna función o variable cuyo nombre coincida con las que acabamos de estudiar. De hecho PCLex también genera una serie de variables auxiliares cuyo nombre consta de un solo

carácter, por lo que tampoco es bueno declarar variables o funciones con nombres de una sola letra ni que empiecen por **yy**.

2.5 El generador de analizadores lexicográficos JFlex

JFlex es un generador de analizadores lexicográficos desarrollado por Gerwin Klein como extensión a la herramienta JLex desarrollada en la Universidad de Princeton. JFlex está desarrollado en Java y genera código Java.

Los programas escritos para JFlex tienen un formato parecido a los escritos en PCLex; de hecho todos los patrones regulares admisibles en Lex también son admitidos por JFlex, por lo que en este apartado nos centraremos tan sólo en las diferencias y extensiones, tanto de patrones como del esqueleto que debe poseer el fichero de entrada a JFlex.

La instalación y ejecución de JFlex es trivial. Una vez descomprimido el fichero **jflex-1.3.5.zip**, dispondremos del fichero JFlex.jar que tan sólo es necesario en tiempo de meta-compilación, siendo el analizador generado totalmente independiente. La clase Main del paquete JFlex es la que se encarga de metacompilar nuestro programa .jflex de entrada; de esta manera, una invocación típica es de la forma:

java JFlex.Main fichero.jflex

Figure 5 Obtención de un programa portable en Java a partir de una especificación JFlex

lo que generará un fichero Yylex.java que implementa al analizador lexicográfico. La figura 2.5 ilustra el proceso a seguir, y cómo el nombre por defecto de la clase generada es **Yylex**.

2.5.1 Ejemplo preliminar

Resulta interesante comenzar con un ejemplo preliminar sobre el que poder discutir. El ejemplo (almacenado en un fichero de texto llamado **prueba.jflex**) es el siguiente:

```
/* Ejemplo de JFlex */
import java.io.*;
%%
// %class Lexer
%int
%unicode
// %cup
```

```
%line
%column
%{
    public static void main(String[] args){
        Yylex analizadorLexico =
            new Yylex(new InputStreamReader(System.in));
        try{
            analizadorLexico.yylex();
        }catch(IOException x){
            System.out.println("Error en la línea "+
                                analizadorLexico.yyline+
                                " columna "+
                                analizadorLexico.yycolumn);
        }
    }
%}
terminadorLinea = \r|\n|\r\n
espacioBlanco = {terminadorLinea} | [ \t\f]
%state STRING
%%
<YYINITIAL>"IF"  { System.out.println("Encontrado IF"); }
<YYINITIAL>{
    "="             { System.out.println("Encontrado ="); }
    "!="        { System.out.println("Encontrado !="); }
    "THEN"      { System.out.println("Encontrado THEN"); }
    {espacioBlanco}     { /* Ignorar */; }
    \"              { yybegin(STRING); }
    [:jletter:][:jletterdigit:]*  { System.out.println("Encontrado un ID"); }
}
<STRING>{
    [^\n\"\\]   { System.out.println("Estoy en una cadena"); }
    \n          { System.out.println("Una cadena no puede acabar en \\n");}
    \\(.|\n)    { System.out.println("Encontrado "+ yytext() + " en cadena");}
    \"          { yybegin(YYINITIAL); }
}
.|\n            { System.out.println("Encontrado cualquier carácter"); }
```

Como puede observarse en este ejemplo, el texto se encuentra dividido en tres secciones separadas por **%%**, al igual que en Lex. Sin embargo, el contenido de cada sección o área difiere con respecto a aquél, siendo la estructura general de la forma:

Área de código, importaciones y paquete
%%
Área de opciones y declaraciones
%%
Área de reglas

La primera de estas áreas se encuentra destinada a la importación de los paquetes que se vayan a utilizar en las acciones regulares situadas al lado de cada

patrón en la zona de reglas. Aquí también puede indicarse una cláusula **package** para los ficheros **.java** generados por el meta-analizador.

En general, cualquier cosa que se escriba en este área se trasladará tal cual al fichero .**java** generado por JFlex., por lo que también sirve para escribir clases completas e interfaces.

A continuación se describe en profundidad las siguientes dos áreas.

2.5.2 Área de opciones y declaraciones

Este área permite indicar a JFlex una serie de opciones para adaptar el fichero .**java** resultante de la meta-compilación y que será el que implemente nuestro analizador lexicográfico en Java. También permite asociar un identificador de usuario a los patrones más utilizados en el área de reglas.

2.5.2.1 Opciones

Las opciones más interesantes se pueden clasificar en opciones de clase, de la función de análisis, de fin de fichero, de juego de caracteres y de contadores. Todas ellas empiezan por el carácter **%** y no pueden estar precedidas por nada en la línea en que aparecen.

2.5.2.1.1 Opciones de clase

Las opciones de clase más útiles son:

%class nombreClase. Por defecto, la clase que genera JFlex y que implementa el analizador lexicográfico se llama **Yylex** y se escribe, por tanto, en un fichero **Yylex.java**. Utilizando esta opción puede cambiarse el nombre de dicha clase.

%implements interface1, interface2, etc. Genera una clase (por defecto **Yylex**) que implementa las interfaces indicadas. El programador deberá introducir los métodos necesarios para hacer efectiva dicha implementación.

%extends nombreClase. Genera una clase que hereda de la clase indicada.

%public. Genera una clase pública. La clase generada por defecto no posee modificar de ámbito de visibilidad.

%final. Genera una clase final, de la que nadie podrá heredar.

%abstract. Genera una clase abstracta de la que no se pueden crear objetos.

%{ bloqueJava %}.El programador también puede incluir sus propias declaraciones y métodos en el interior de la clase **Yylex** escribiendo éstas entre los símbolos **%{** y **%}**.

%init{ códigoJava %init}. Es posible incluir código en el constructor generado automáticamente por JFlex para la clase **Yylex** mediante el uso de esta

opción.

2.5.2.1.2 Opciones de la función de análisis

Estas opciones permiten modificar el método o función encargada de realizar el análisis lexicográfico en sí. Las opciones más útiles son:

%function nombreFuncionAnalizadora. Por defecto, la función que arranca el análisis se llama **yylex()** y devuelve un valor de tipo **Yytoken**. Esta opción permite cambiar de nombre a la función **yylex()**.

%int. Hace que la función **yylex()** devuelva valores de tipo **int** en lugar de **Yytoken**.

%intwrap. Hace que la función **yylex()** devuelva valores de tipo **Integer** en lugar de **Yytoken**.

%type nombreTipo. Hace que la función **yylex()** devuelva valores del tipo especificado (ya sea primitivo o no) en lugar de **Yytoken**.

En cualquier caso, JFlex no crea la clase **Yytoken**, sino que debe ser suministrada de forma externa, o bien especificada en el área de código.

2.5.2.1.3 Opciones de fin de fichero

Por defecto, cuando **yylex()** se encuentra el carácter fin de fichero, retorna un valor **null**. En caso de que se haya especificado **%int** en las opciones anteriores, se retornará el valor **YYEOF** definido como **public static final** en la clase **Yylex**. Para cambiar este comportamiento, las opciones de fin de fichero más útiles son:

%eofclose. Hace que **yylex()** cierre el canal de lectura en cuanto se encuentre el EOF.

%eofval{ códigoJava %eofval}. Es posible ejecutar un bloque de código cuando **yylex()** se encuentre el EOF. Para ello se utilizará esta opción.

2.5.2.1.4 Opciones de juego de caracteres

Estas opciones permiten especificar el juego de caracteres en el que estará codificada la entrada al analizador lexicográfico generado por JFlex. Las posibilidades son:

%7bit. Es la opción por defecto, y asume que cada carácter de la entrada está formado por un único byte cuyo bit más significativo es 0, lo que da 128 caracteres.

%8bit. Asume que cada carácter está formado por un byte completo, lo que da 256 caracteres.

%unicode. Asume que la entrada estará formada por caracteres Unicode. Esto no quiere decir que se tomen dos *bytes* de la entrada por cada carácter sino que

la recuperación de caracteres se deja recaer en la plataforma sobre la que se ejecuta **yylex().**

%ignorecase. Hace que **yylex()** ignore entre mayúsculas y minúsculas mediante el uso de los métodos **toUpperCase** y **toLowerCase** de la clase **Character**.

Resulta evidente que las tres primeras opciones son mutuamente excluyentes.

2.5.2.1.5 Opciones de contadores

Estas opciones hacen que el analizador lexicográfico almacene en contadores el número de caracteres, líneas o columnas en la que comienza el lexema actual. La nomenclatura es:

%char. Almacena en la variable **yychar** el número de caracteres que hay entre el comienzo del canal de entrada y el comienzo del lexema actual.

%line. Almacena en la variable **yyline** el número de línea en que comienza el lexema actual.

%column. almacena en la variable **yycolumn** el número de columna en que comienza el lexema actual.

Todos los contadores se inician en 0.

2.5.2.2 Declaraciones.

Además de opciones, el programador puede indicar declaraciones de dos tipos en el área que nos ocupa, a saber, declaraciones de estados léxicos y declaraciones de reglas.

2.5.2.2.1 Declaraciones de estados léxicos.

Los estado léxicos se declaran mediante la opción:

%state estado1, estado2, etc.

y pueden usarse en el área de reglas de la misma manera que en Lex. Además, si hay muchos patrones en el área de reglas que comparten el mismo estado léxico, es posible indicar éste una sola vez y agrupar a continuación todas estas reglas entre llaves, como puede observarse en el ejemplo preliminar del punto 2.5.1. El estado inicial viene dado por la constante **YYINITIAL** y, a diferencia de Lex, puede utilizarse como cualquier otro estado léxico definido por el usuario.

Una acción léxica puede cambiar de estado léxico invocando a la función **yybegin(estadoLéxico)** generada por JFlex.

2.5.2.2.2 Declaraciones de reglas.

En caso de que un patrón se utilice repetidas veces o cuando su complejidad es elevada, es posible asignarle un nombre y utilizarlo posteriormente en cualquier otra regla encerrándolo entre llaves, de manera análoga a como se estudió en Lex. La sintaxis es:

nombre = patron

2.5.3 Área de reglas.

El área de reglas tiene la misma estructura que en Lex, con la única diferencia de que es posible agrupar las reglas a aplicar en un mismo estado léxico. Como ejemplo, las reglas:

```
<YYINITIAL>"="          { System.out.println("Encontrado ="); }
<YYINITIAL>"!="         { System.out.println("Encontrado !="); }
```

también pueden escribirse como:

```
<YYINITIAL>{
        "="             { System.out.println("Encontrado ="); }
        "!="            { System.out.println("Encontrado !="); }
}
```

Las diferencias importantes de este área con respecto a Lex se concentran en que JFlex incluye algunas características adicionales para la especificación de patrones, como son:

\u**nºhexadecimal**: representa el carácter cuyo valor Unicode es **nºhexadecimal**. Ej.: \u042D representa al carácter "Э".

!: encaja con cualquier lexema excepto con los que entran por el patrón que le sucede. Ej.: !(ab) encaja con cualquier cosa excepto el lexema "ab".

~: encaja con un lexema que empieza con cualquier cosa y acaba con la primera aparición de un texto que encaja con el patrón que le sigue. P.ej., la manera más fácil de reconocer un comentario al estilo de Java (no anidado) viene dada por el patrón: "/*"~"*/"

Por otro lado JFlex incluye algunas clases de caracteres predefinidas. Estas clases van englobadas entre los símbolo [: y :] y cada una encaja con el conjunto de caracteres que satisfacen un determinado predicado. Por ejemplo, el patrón [:digit:] encaja con aquellos caracteres que satisfacen la función lógica **isDigit()** de la clase **Character**. La lista de patrones es:

Patrón	Predicado asociado
[:jletter:]	isJavaIdentifierStart()
[:jletterdigit:]	isJavaIdentifierPart()
[:letter:]	isLetter()
[:digit:]	isDigit()
[:uppercase:]	isUpperCase()
[:lowercase:]	isLowerCase()

Finalmente, otras dos diferencias importantes con respecto a Lex son:

- Obligatoriamente toda regla debe tener una acción léxica asociada, aunque sea

vacía.

- Si un lexema no encaja por ningún patrón se produce un error léxico.
- El patrón \n engloba sólo al carácter de código ASCII 10, pero no al de código ASCII 13, que viene representado por el patrón \r.

2.5.4 Funciones y variables de la clase Yylex

Ya hemos visto dos de las funciones más importantes generadas por JFlex y que pueden ser utilizadas por el programador en el interior de las acciones léxicas. Se trata de funciones y variables miembro, por lo que, si son utilizadas fuera de las acciones léxicas deberá indicarse el objeto contextual al que se aplican (p.ej. **miAnalizador.yylex()**). Aquí las recordamos y las extendemos:

Yylex(Reader r): es el constructor del analizador léxico. Toma como parámetro el canal de entrada del cual se leerán los caracteres.

Yytoken yylex(): función principal que implementa el analizador léxico. Toma la entrada del parámetro especificado en la llamada al constructor de **Yylex**. Puede elevar una excepción **IOException**, por lo que se recomienda invocarla en el interior de una sentencia **try-catch**.

String yytext(): devuelve el lexema actual.

int yylength(): devuelve el número de caracteres del lexema actual.

void yyreset(Reader r): cierra el canal de entrada actual y redirige la entrada hacia el nuevo canal especificado como parámetro.

void yypushStream(Reader r): guarda el canal de entrada actual en una pila y continúa la lectura por el nuevo canal especificado. Cuando éste finalice continuará por el anterior, extrayéndolo de la pila. Esta función es de especial importancia para implementar la directiva **#include** de lenguajes como C.

void yypushback(int n): equivale a la función **yyless()** de Lex.

yyline, **yychar** e **yycolumn**: son las variables generadas por las opciones **%line**, **%char** y **%column** respectivamente, y comentadas en el apartado 2.5.2.1.5.