JFlex, Manual de Usuario

Versión 1.3.5, 8 Octubre, 2001

Gerwin Klein

Manuel Nieto Uclés
José María Palacios Carabias

Contenidos

1 Introducción

JFlex es un generador de analizadores léxicos para Java[1] escrito en Java. Esta basado en una herramienta muy útil, el JLex [3], el cuál fue desarrollado por Elliot Berk en la universidad De Princeton. Como los estados de Vern Paxon para su herramienta C/C++, flex [11]: ellas no comparten nada de su código.

1.1 Metas del diseño

Las principales metas del diseño de JFlex son:

- Soporte completo para unicote
- Rápidos escáneres generados
- Generación rápida de escáneres
- Especificación sintáctica conveniente
- Independencia de la plataforma
- Compatibilidad con Alex

1.2 Acerca de este manual

Este manual da una breve pero completa descripción de la herramienta JFlex. Se asume, que esta familiarizado con los intríngulis del análisis léxico [2], [1] y [13] provee una buena introducción a este tópico.
La próxima sección de este manual describe los procedimientos de instalación para JFlex. Si nunca has trabajado con JLex o simplemente quieres comparar una especificación de un escáner en JLex y JFlex debería también leer *Trabajando con JFlex – un ejemplo* (sección 3). Todas las opciones y la especificación completa de la semántica se presentan en *Especificaciones Léxicas* (sección 4), mientras *Codificaciones, Plataformas, y Unicote* (sección 5) provee información acerca del texto escaneado contra los archivos binarios. Si estas interesado en consideraciones de mejoras y comparar la velocidad de JLex con JFlex, *pocas palabras en mejoras* (sección 6) debe ser suficiente para usted. Aquellos que solían usar sus viejas especificaciones en JLex deben ver la sección 7.1 *Portando desde JLex* para resolver posibles problemas con JLex anteriores que han sido resueltos en JFlex. La sección 7.2 habla acerca de trasladar escáneres de las herramientas Unix lex o flex. Los interfaces de los escáneres JFlex con el LARL generador de "parser" CUP y BYacc/J es explicado en *trabajando juntos* (sección 8). La sección 9 *Errores* da una lista de los errores activos actualmente conocidos. El manual concluye con notas acerca de *Copias y Licencia* (sección 10) y referencias.

[1] Java es una marca registrada de Sun Microsystems, Inc., y se refiere al lenguaje de programación Java de Sun. JFlex no está esponsorada por o afiliada con Sun Microsystems, Inc.

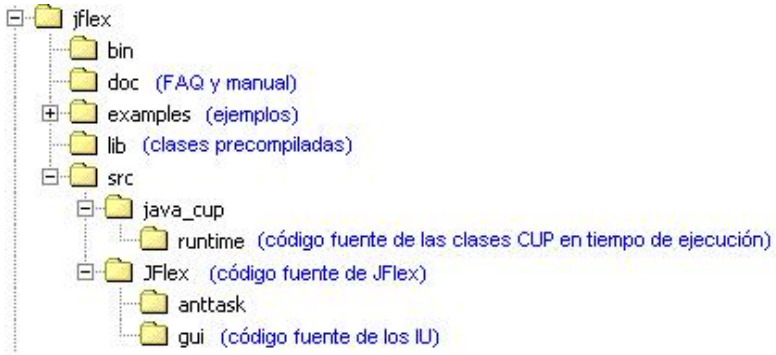
2 Instalando y Ejecutando JFlex

2.1 Instalando JFlex

2.1.1 Windows

Para instalar JFlex en Window95/98/NT, sigue los siguientes pasos:

1.- Descomprime el archivo con el programa en el directorio que tu desees (utiliza un descompresor como el **WinZip**[2]). Si descomprimiste el archivo en C:\, se habrá generado la siguiente estructura:



2.- Edita el fichero bin\JFlex.bat (en el ejemplo la ruta es **C:\JFlex\bin\JFlex.bat**) y especifica el valor de las variables siguientes:

- ***JAVA_HOME*** indica el directorio donde se encuentra el Java JDK instalado
- ***JFLEX_HOME*** indica el directorio donde se encuentra el JFlex

3.- Incluye el directorio **\bin** de JFlex en tu ruta.

2.1.2 Unix con archivos tar

Para instalar JFlex en un sistema Unix, siga los siguientes dos pasos:

- Descomprime el archivo en un directorio de tu elección con GNU tar, por defecto **/usr/share**:

tar –C /usr/share/ -xvzf jflex-1.3.5.tar.gz

[2] http://www.winzip.com

(El ejemplo es para una instalación total. Necesita ser root para esto. La instalación de usuario trabaja exactamente de la misma manera – simplemente elige el directorio donde tengas permiso de escritura)

- Haga un enlace simbólico desde algún lugar en tu camino binario a **bin/jflex**, por ejemplo:
  ln –s /usr/share/JFlex/bin/jflex
  Si el interprete de java no está en tu camino binario, necesitas suplir su localización en el script **bin/jflex**

Puedes verificar la integridad de los archivos descargados con el MD5 disponible en la página de descarga de JFlex. Si colocas el comprobador de archivos en el mismo directorio que el archivo, ejecutarías:
md5sun --check jflex-1.3.5.tar.gz.md5

Debería mostrar
Jflex-1.3.5.tar.gz: OK

2.1.3 Linux con RPM

- convertirte en root
- ejecutar
  rpm –U jflex-1.3.5-0.rpm

Puedes verificar la integridad del **rpm** descargado con
rpm --checksig jflex-1.3.5-0.rpm

Esto requiere mi pgp clave pública. Si no la tienes puedes usar
rpm --checksig --nopgp jflex-1.3.5-0.rpm

o puedes obtener una en http://www.jflex.de/public-key.asc

2.2 Ejecutando JFlex

Para ejecutar JFlex utiliza:
jflex <opciones> <ficheros de entrada>

Es posible también suprimir el script de entrada en el directorio **bin** e incluir el archivo **lib\JFlex.jar** en la variable de entorno **CLASSPATH**.

En tal caso ejecutaríamos de la forma:
java JFlex.Main <opciones> <ficheros de entrada>

Los ficheros de entrada y las opciones son opcionales en ambos casos. Si no los especificas JFlex abrirá una ventada para que indiques la ruta de los mismos.

JFlex reconoce las siguientes opciones:

-d<directorio>	Guarda el fichero generado en el directorio

--skel <archivo>	Usa el esqueleto externo que viene dado por el archivo. Esto es principalmente para el mantenimiento de JFlex y su personalización a bajo nivel. Úsalo solo cuando tu sepas los que estás haciendo! JFlex viene con su esqueleto en el directorio src que refleja exactamente el esqueleto interno, precompilado y puede ser usado con la opción **–skel**.
--nomin	Desactiva el paso de minimización DFA durante la generación del escaner.
--dot	Genera archivos dot para el NFA, DFA y minimizado DFA. Esta característica está todavía en estado alfa, ni totalmente implementada.
--dump	Muestra la transición entre las tablas de NFA, DFA inicial y DFA minimizado
--verbose ó –v	Muestra los mensajes de progreso de la generación (activado por defecto)
--quiet ó –q	Muestra solo los mensajes de error
--time	Muestra las estadísticas de tiempo acerca del proceso de generación de código (no muy acertadas)
--version	Imprime el número de versión
--info	Imprime la información del sistema y del jdk (útil si das parte de algún problema)
--pack	Usa el método %pack para la generación de código por defecto
--table	Usa el método %table para la generación de código por defecto
--switch	Usa el método %switch para la generación de código por defecto
--help ó –h	Imprime un mensaje de ayuda explicando la opciones y usos del JFlex

3 Un Ejemplo simple: Como trabajar con JFlex

Para demostrar como es una especificación léxica con JFlex, esta sección presenta una parte de la especificación del lenguaje Java. El ejemplo no describe toda la estructura léxica completa de los programas Java, sino una pequeña y simplificada parte de ella (algunas palabras clave, algunos operadores, comentarios y solo dos tipos de literales). También muestra como interactuar con el LARL generador parser CUP [8] y por lo tanto usar una clase sym (generada por CUP), donde las constantes enteras para los token terminales de la gramática CUP están declarados. JFlex viene con un directorio de ejemplos, donde tú puedes encontrar un pequeño escáner, que no necesita otras herramientas como CUP, para darte un ejemplo que se puede ejecutar. El directorio de "ejemplos" también contiene una especificación JFlex completa de la estructura léxica de programas Java junto con la especificación CUP parser para Java para C. Scout Ananian, obtenida del sitio web CUP [8] (ha sido modificado para interactuar con el escáner JFlex). Ambas especificaciones dependen de la especificación del lenguaje Java [7].

```
/* Los ejemplos JFlex: parte de la especificación Java de un generador léxico */

Import java_cup.runtime.*;

%%


%class Lexer
%unicode
%cup
%line
%column


%{
    StringBuffer string = new StringBuffer();

    private Symbol symbol(int type){
        return new Symbol(type, yyline, yycolum);
    }

    private Symbol symbol(int type, Object value){
        return new Symbol(type, yyline, yycolumn, value);
    }
%}


LineTerminator  = \r | \n | \r\n
InputCharacter  = [^\r\n]
WhiteSpace          = {LineTerminator} | [\t\f]


/* comentarios */
Comment = {TradicionalComment} | {EndOfLineComment} | {Documentation Comment}


TraditionalComment   = "/*" [^*] ~"*/"
EndOfLineComment     = "//" {InputCharacter}* {LineTerminator}
DocumentationComment     = "/**" {CommentContent} "*" + "/"
CommentContent                  = ( [^*] | \* + [^/*] )*


Identifier   = [:jletter:] [:jletterdigit:]*

DecIntegerLiteral       = 0 | [1-9][0-9]*

%state STRING


%%
```

```
/* palabras clave */
<YYINITIAL> “abstract”      {return symbol(sym.ABSTRACT);}
<YYINITIAL> “boolean”       {return symbol(sym.BOOLEAN);}
<YYINITIAL> “break” {return symbol(sym.BREAK);}
<YYINITIAL>{
      /* identificadores */
      {Identifier}              {return symbol(sym.IDENTIFIER);}

      /* literales */
      {DecIntegerLiteral}{return symbol(sym.INTEGER_LITERAL);}
      \”                        {string.setLength(0); yybegin(STRING);}

      /* operadores */
      “=”                       {return symbol(sym.EQ);}
      “==”                      {return symbol(sym.EQEQ);}
      “+”                       {return symbol(sym.PLUS);}

      /* comentarios */
      {Comment}                       { /*ignorarlos*/ }

      /* espacios en blanco */
      {WhiteSpace}              { /*ignorarlos*/ }
}

<STRING> {
      \"                        {yybegin(YYINITIAL);
                                 return symbol(sym.STRING_LITERAL,
                                 string.toString()); }
      [^\n\r\"\\]+              { string.append( yytext() ); }
      \\t                       { string.append('\t'); }
      \\n                       { string.append('\n'); }
      \\r                       { string.append('\r'); }
      \\\"                      { string.append('\"'); }
      \\                        { string.append('\\'); }
}

/* error fallback */
      .|\n                      { throw new Error("Illegal character
<"+
                                                  yytext()+">"); }
```

Para esta especificación JFlex genera un archivo **java** para una clase que contiene código para el escáner. La clase tendrá un constructor con parámetro, tomando un **java.io.Reader** del cual se lee la entrada. La clase tendrá también una función **yylex()** que ejecutará el escáner y podrá ser usada para conseguir el siguiente token de la entrada (en este ejemplo la función tiene el nombre **next_token()** porque la especificación usa la forma del CUP).

Como con el JLex, la especificación se divide en tres partes, dividida por %%:

- código de usuario
- opciones y declaraciones
- reglas léxicas

3.1 Código a incluir

Vamos a mirar la primera sección, "código de usuario": El texto de la primera línea empieza con %% es una copia textual de la parte superior de la clase léxica generada (antes de la declaración actual de la clase). Fuera de los bloques de **package** e **import** no suele existir mucho que hacer aquí.

3.2 Opciones y Macros

La segunda sección "opciones y declaraciones" es más interesante. Esta consiste en un conjunto de opciones, código que está incluido dentro de la clase escáner generada, estados léxicos y declaraciones de macros. Cada opción de JFlex debe comenzar con %. En nuestro ejemplo las siguientes opciones fueron usadas:

- **%class Lexer** dice a JFlex que tiene que colocar como nombre de la clase generada "Lexer" y tiene que escribir el código en un archivo "Lexer.java"
- **%unicode** define el conjunto de caracteres con los que trabajará. Para escanear ficheros de texto, **%unicode** debería ser usado siempre. Ver sección 5.
- **%line** cambia la forma de contar las líneas (el número de línea correspondiente puede ser consultado mediante la variable yyline)
- **%colum** cambia la forma de contar las columnas (el número de columna correspondiente puede ser consultado mediante la variable yycolumn)

El código incluido en **%{...%}** es copiado textualmente en el código de la clase léxica generada. Aquí puedes declarar variables miembro y funciones que son usadas dentro de las acciones del scáner. En nuestro ejemplo declaramos un **StringBuffer "string"** en el cual guardaremos pedazos de cadenas de literales y dos funciones de ayuda **"symbol"** que crean **java_cup.runtime.Symbol** objetos con posición de la información del token actual (ver sección 8.1 *JFlex y Cup*). Como las opciones de JFlex, **%{** y **%}** deben estar al comienzo de una línea.
La especificación continúa con las declaraciones de macros. Macros son abreviaciones de expresiones regulares, usadas para realizar especificaciones léxicas fáciles de leer y comprender. La declaración de una macro consiste en un identificador de macro seguido de =, seguido de una expresión regular que lo representa. Esta expresión regular debe contener por si misma los usos de la macro. Aunque esto permite una gramática al estilo de una especificación, las macros son solo abreviaturas y no son no-terminales, no pueden ser recursivas o mutuamente recursivas. Ciclos en las definiciones de macros son detectados y reportados en tiempo de generación por JFlex.
Aquí hay algunos ejemplos de macros con más detalle:

- **LineTerminator** representa a la expresión regular que unifica con un CR en ASCII, LF o CR seguido de LF.
- **InputCharacter** representa cualquier carácter que no sea CR o LF.
- **TraditionalComment** es la expresión que unifica con la cadena "/*" seguida por un carácter que no sea * seguido por algo que unifique con la macro **CommentContent** seguida por cualquier número de * seguido de /.
- **Identifier** unifica cada cadena que empieza con un carácter de la clase **jletter** seguido por cero o más caracteres de la clase **jletterdigit**. **jletter** y **jletterdigit** son clases de caracteres predefinidas. **jletter** incluye todos los caracteres para los cuales la función de Java **Caracter.isJavaIdentifierStart** devuelve true y **jletterdigit** todos los caracteres para los cuales **Caracter.isJavaIdentifierPart** devuelve **true.**

La parte final de la segunda sección en nuestra especificación léxica es un estado de declaración léxico: **%state STRING** declara un estado léxico **STRING** que puede ser usado en las "reglas léxicas" parte de la especificación. Una declaración de estado es una línea empezando con **%state** seguida por una lista de identificadores de estado separados por comas o espacios. Puede haber más de una línea empezando con **%state**.

3.3 Reglas y Acciones

La sección de las "reglas léxicas" de una especificación JFlex contiene expresiones regulares y acciones (en código Java) que son ejecutadas cuando el escáner encuentra la expresión regular asociada. Cuando el escáner lee su entrada, mantiene pistas de todas las expresiones regulares y activa la acción de la expresión más larga. Nuestra especificación sobre el ejemplo debería con la entrada **"breaker"** reconocerla con la expresión regular para **Identifier** y no con la palabra clave **"break"** seguida por el identificador **"er"**, porque la regla **{Identificador}** reconoce más de esta entrada que cualquier otra regla de la especificación. Si dos expresiones regulares tienen el máximo valor reconocido para cierta entrada, el escáner elige la acción de la expresión que aparece primero en la especificación. De esta manera, conseguiremos para la entrada **"break"** la palabra clave **"break"** y no un identificador **"break"**.
Adicionalmente a las expresiones regulares, uno puede usar estados léxicos para refinar una especificación. Un estado léxico actúa como una condición de inicio. Si el escáner está en el estado léxico **STRING**, solo las expresiones que son precedidas por la condición inicial **<STRING>** pueden ser reconocidas. Una condición de comienzo de una expresión regular puede contener más de un estado léxico. Si es reconocido cuando el generador léxico está en uno de esos estados léxicos. El estado léxico **YYINITIAL** esta predefinido y es también el estado en el cual el generador empieza a escanear. Si una expresión regular no tiene condición de comienzo esta es reconocida en todos los estados léxicos.
Como a menudo se tienen un montón de expresiones con las mismas condiciones de inicio, JFlex permite la misma abreviatura que la herramienta de Unix **flex**:

<STRING> {

```
        expr1       {acción 1}
        expr2       {acción 2}
}
```

significa que **expr1** y **expr2** tienen como condición de inicio **<STRING>**.
Las primeras tres reglas en nuestro ejemplo demuestran la sintaxis de una expresión regular precedida por la condición inicial **<YYINITIAL>**.

<YYINITIAL> "abstract" { return symbol(sym.ABSTRACT); }
Reconoce la entrada **"abstract"** solo si el escáner esta en su estado inicial **"YYINITIAL"**. Cuando la cadena **"abstract"** es reconocida, la función del escáner devuelve el símbolo CUP **sym.ABSTRACT**. Si una acción no devuelve un valor, el proceso del escáner se reanuda inmediatamente después de ejecutar la acción.

Las reglas encerradas en
```
<YYINITIAL> {
...
}
```
demuestra la sintaxis abreviada y son también reconocidas en el estado **YYINITIAL**.

De esas reglas, una debería ser de especial interés:
\" { string.setLength(0); yybegin(STRING); }
Si el escáner reconoce una doble comilla en el estado **YYINITIAL** hemos reconocido el comienzo e una cadena literal. Por lo tanto limpiamos nuestro **StringBuffer** que mantendrá el contenido de esta cadena de literales y le dirá al escáner con **yybegin(STRING)** que cambie al estado léxico **STRING**. Porque nosotros no hemos devuelto todavía un valor al parser, nuestro escáner prosigue inmediatamente.
En el estado léxico STRING otra regla demuestra como se refiere a la entrada que ha sido reconocida:
[^\n\r\"]+ { string.append(yytext()); }
La expresión **[^\n\r\"]+** reconoce todos los caracteres de la entrada con la próxima acción repentina (indicando una secuencia de escape como \n), dobles comillas (indicando el fin de la cadena), o un terminador de línea (el cual no debe ocurrir en una cadena literal). La región reconocida de la entrada está referida hacia **yytext()** o añadida al contenido de una cadena literal parseada al efecto.
La última regla léxica en la especificación del ejemplo se usa para recuperar errores. Ella reconoce a cualquier carácter en cualquier estado que no haya sido reconocido con otra regla. Esto no presenta un conflicto con otras reglas debido a que goza de menos prioridad (porque es la última regla) y porque reconoce un solo carácter (así que no puede tener un reconocimiento más largo que otra regla).

3.4 Como proceder

- Instala JFlex. (ver sección 2 *Instalando JFlex*)
- Si has escrito tu especificación léxica (por ejemplo el nombre **java-lang.flex**).
- Ejecuta JFlex con

jflex java-lang.flex

- JFlex debería devolver algunos mensajes acerca del progreso de la generación de escáner y escribir el código generado al directorio del archivo de especificación.

- Compila el archivo generado junto con tus propias clases. (Si usas CUP, genera tus clases parser primero)

- Eso es todo

4 Especificaciones léxicas

Como hemos visto antes, un fichero de especificación léxica de JFlex consiste en tres partes divididas por una línea que comienza por **%%**:

UserCode
% %
Options and declarations
% %
Lexical rules

Se puede incluir en la especificación comentarios de la forma **/* comentario de texto */** y también comentarios al estilo Java de una sola línea comenzando por //. El número de **/* y */** debe ser balanceado.

4.1 Código de usuario

La primera parte contiene el código de usuario es copiada textualmente en el principio de archivo de código del generador léxico antes de la declaración del escáner. Como vimos en el ejemplo, este es el lugar para colocar la declaración del **paquete** y las **importaciones**. Si es posible, pero no es considerado como un buen estilo de programación Java, colocar nuestra propia clase de ayuda (por ejemplo unas clases de token) en esta sección. Esta clase debería tener su propia archivo **java**.

4.2 Opciones y declaraciones

La segunda parte de la especificación léxica contiene las opciones para personalizar el reconocedor léxico generado (las directivas JFlex y el código Java a incluir en diferentes partes del reconocedor léxico), las declaraciones de los estados léxicos y las definiciones de macros para usar en la tercera sección "Reglas léxicas" del fichero de especificación.
Cada directiva JFlex debe estar situada al principio de la línea y comienza con el carácter %. Las directivas que tienen uno o más parámetros están descritas como siguen:
%class "classname"
significa que si tú comienzas una línea **%class** seguido de un espacio seguido por el nombre de la clase del escáner generado (las dobles comillas no se colocan, ver el ejemplo en la sección 3).

4.2.1 Opciones de clase y código de usuario en la clase

Estas opciones respetan el nombre, del constructor y las partes relacionadas con la clase escáner generado.

- **%class "classname"**
  Le dice a JFlex que coloque **"classname"** como el nombre de la clase generada y que escriba en un archivo llamado **"classname.java"**. Si la opción **-d <directory>** no es usada, el código será escrito en el directorio donde se encuentran los archivos de especificación. Si no existe la directiva **%class**, la clase generada tendrá el nombre **"Yylex"** y será escrita en un archivo **"Yylex.java"**. Debería haber una sola directiva **%class** en la especificación.

- **%implements "interface 1"[, "interface 2", ..]**
  Realiza en la clase generada la implementación de las interfaces especificadas. Si existe más de una directiva **%implements**, todos los interfaces especificados serán implementados.

- **%extends "classname"**
  Coloca la clase generada como una subclase de la clase "classname". Solo debería haber una directiva %extends en la especificación.

- **%public**
  Atribuye a la clase generada la categoría de pública (por defecto la clase tiene visibilidad de paquete).

- **%final**
  La clase generada es final.

- **%abstract**
  La clase generada es abstracta.

- **%{**
  ...
  %}
  El código encerrado entre **%{** y **%}** es copiado textualmente en la clase generada. Aquí puedes definir tus propias variables miembro y funciones el en escáner generado. Como las opciones, ambos **%{** y **%}** deben estar al principio de una línea de la especificación. Si aparece más de una directiva **%{...%}**, el código es concatenado en orden a la aparición en la especificación.

- **%init{**
  ...
  %init}
  El código encerrado en **%init{**y **%init}** es copiado literalmente en el constructor de la clase generada. Aquí, los miembros declarados en la directiva **%{...%}** pueden ser inicializados. Si más de una opción de inicialización está presente, el código es concatenado en el orden de aparición en la especificación.

- **%initthrow{**
  "exception1"[, "exception2", ...]
  %initthrow}
  o (en una línea)
  %initthrow "exception1" [, "exception2",...]
  Causa que las excepciones sean declaradas en la cláusula **throws** del constructor. Si existe más de una directiva **%initthrow{... %initthrow}** en la especificación, todas las especificaciones serán declaradas.

- **%scanerrror "exception"**
  Causa que el escáner generado lance una instancia de la excepción especificada en caso de que exista un error interno (por defecto es **java.lang.Error**). Notar que esta excepción es solo para los errores internos del escáner. Con las especificaciones usuales nunca debería ocurrir.

- **%buffer "size"**
  Coloca el tamaño inicial del buffer del escáner al valor especificado (decimal, en bytes). El valor por defecto es 16384.

- **%include "filename"**
  Cambia textualmente el **%include** con el fichero especificado. Esta función es todavía experimental. Esto funciona, pero el informe de errores puede ser extraño si un error de sintaxis ocurre en el último token del archivo incluido.

4.2.2 Método de escaneo

Esta sección nos muestra como el método de escaneo puede ser personalizado. Puedes redefinir el nombre y el tipo de retorno del método y si es posible declarar excepciones que pueden ser lanzadas en una de las acciones de la especificación. Si no se especifica el tipo de retorno, el método de escaneo será declarado como los valores de retorno de la clase **Yytoken**.

- **%function "name**
  Causa que el método de escaneo tenga el nombre especificado. Si la directiva **%funtion** no está presente en la especificación, el método de escaneo tendrá el nombre **"yylex"**. Esta directiva sobrescribe las opciones de la opción **%cup**. Note que el nombre por defecto del método de escaneo con la opción **%cup** es **next_token**. Cambiar el nombre puede ocasionar que el escáner generado sea declarado como abstracto implícitamente, porque no provee el método **next_token** de la interfaz **java_cup.runtime.Scanner**. Por supuesto es posible proveer una implementación tonta de este método en la sección de código de la clase, si aún quieres cambiar el nombre de la función.

- **%integer**
  %int

Ambas directivas causan que el método de escaneo sea declarado como un tipo **entero** de Java. Las acciones en la especificación pueden devolver valores **enteros** como tokens. El valor de final de línea por defecto en estas opciones es **YYEOF**, el cual es un **public static final int** miembro de la clase generada.

- **%intwrap**
  Causa que el método de escaneo sea declarado como el tipo envuelto en Java **Integer**. Las acciones en la especificación puede devolver valores **Integer** como tokens. El valor para el final de línea por defecto es **null**.

- **%type "typename"**
  Causa que el método de escaneo declare como valores de retorno el tipo especificado. Las acciones en la especificación pueden devolver valores de **typename** como tokens. El valor por defecto para el final de línea en esta opción es **null**. Si **typename** no es una subclase de **java.lang.Object**, deberías especificar otro valor para el final de línea usando la directiva **%eofval{... %eofval}** o la regla **<<EOF>>**. La directiva **%type** cambia las opciones de la opción **%cup**.

- **%yylexthrow{**
  "exception1"[, "exception2", ...]
  %yylexthrow}
  o (en una línea)
  %yylexthrow "exception1" [, "exception2",...]
  Las excepciones listadas dentro de **%yylexthrow{...%yylexthrow}** serán declaradas en la cláusula **throws** del método de escaneo. Si hay más de una cláusula **%yylexthrow{...%yylexthrow}** en la especificación, todas la excepciones especificadas serán declaradas.

4.2.3 El final del fichero

Siempre hay un valor por defecto que el método de escaneo devolverá cuando el final del fichero haya sido alcanzado. Puedes no obstante definir un valor específico para devolver en un trozo de código específico que debería ser ejecutado cuando se alcanza el final del fichero.
El valor del final de fichero por defecto depende del tipo de retorno del método de escaneo:

- Para **%integer**, el método de escaneo retornará el valor **YYEOF**, el cual es **public static final int** miembro de la clase generada.
- Para **%intwrap**,
- sin tipo especificado, o un
- tipo definido por el usuario, declarado usando **%type**, el valor es **null**.
- En el modo de compatibilidad CUP, usando **%cup**, el valor es **new java_cup.runtime.Symbol(sym.EOF)**

Los valores de usuario y el código para ser ejecutado al final del fichero pueden ser definidos usando estas directivas:

- **%eofval{**
  ...
  %eofval}
  El código incluido en **%eofval{...%eofval}** será copiado textualmente dentro del método de escaneo y será ejecutado cada vez que el archivo se acaba (esto es posible cuando el método de escaneo es llamado de nuevo después del haber acabado el fichero). El código debería devolver el valor que indica el fin del archivo al parser. Solo debería haber un cláusula **%eofval{...%eofval}** en la especificación. La directiva **%eofval{...%eofval}** cambia las opciones de la opción **%cup**. Como en la versión 1.2 JFlex provee una forma más legible de especificar el valor del final del fichero usando la regla **<<EOF>>** (ver más en la sección 4.3.2).

- **%eof{**
  ...
  %eof}
  El código incluido en **%{eof...%eof}** será ejecutado solamente una vez, cuando se alcance el final del fichero. El código incluido dentro del método **void yy_do_eof()** y no debería devolver ningún valor (use **%eofval{...%eofval}** o **<<EOF>>** para este propósito). Si existe más de una directiva de final de fichero, el código será concatenado en orden de aparición en la especificación.

- **%eofthrow{**
  "exception1"{,"exception2", ...]
  %eofthrow}
  o (en una sola línea)
  %eofthrow "exception1" [, "exception2", ...]
  Las excepciones listadas dentro de **%eofthrow{...%eofthrow}** serán declaradas en la cláusula **throws** del método **yy_do_eof()** (ver **%eof** para más cosas en este método). Si no hay más de un cláusula **%eofthrow{...%eofthrow}** en la especificación, todas las excepciones especificadas serán declaradas.

- **%eofclose**
  Causa que JFlex cierre el flujo de entrada al final del fichero. El código **yyclose()** se añade al método **yy_do_eof()** (junto con el código especificado en **%eof{...%eof}**) y la excepción **java.io.IOException** es declarada en la cláusula **throws** de este método (junto con los **%eofthrow{...%eofthrow}**).

4.2.4 Escáneres Solos

- **%debug**
  Crea una función main en la clase generada que espera el nombre de un fichero de entrada en la línea de comando y entonces ejecuta el escáner en este fichero de entrada imprimiendo información acerca de cada token devuelto a la consola Java hasta que acaba el fichero. La información incluye: número de línea (si está habilitada la cuenta de líneas), columnas (en caso de que se habilitara), el texto marcado, y la acción ejecutada (con el número de línea en la especificación).

- **%standalone**
  Crea una función main en la clase generada que espera el nombre de un fichero de entrada en la línea de comando y entonces ejecuta el escáner en este fichero de entrada. El valor devuelto por el escáner es ignorado, pero algún texto no seleccionado aparece en la consola en su lugar (como la herramienta de C/C++, si ejecutas un programa solo). Para evitar tener que usar una clase extra de tokens, el método de escaneo será declarado con un tipo por defecto **int**, no **YYtoken** (si no hay otro tipo explícitamente especificado). Esto es en la mayoría de los casos irrelevante, pero podría ser util para saber cuando otro escáner solo para algún propósito. Deberías también considerar usar la directiva **%debug**, si simplemente quieres ser capaz de ejecutar el escáner sin un parser añadido para testear etc.

4.2.5 Compatibilidad CUP

Deberías leer la sección 8.1 JFlex y CUP si estás interesado en el interfaz entre tu clase y CUP.

- **%cup**
  La directiva %cup habilita el modo de compatibilidad CUP y es equivalente al siguiente conjunto de directivas:

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
%eofval{
      return new java_cup.runtime.Symbol(<CUPSYM>.EOF) ;
%eofval}
%eofclose
```

  El valor de **<CUPSYM>** por defecto **sym** puede ser cambiado con la directiva **%cupsym**.

- **%cupsym "classname"**
  Personalizar el nombre de la clase/interfaz que contiene los nombres de los tokens terminales. Por defecto es **sym**. La directiva no debería ser usada después de **%cup**, sino antes.

- **%cupdebug**
  Crea una función main en la clase generada que espera el nombre de un fichero de entrada en la línea de comando y entonces ejecuta el escáner con este fichero. Imprime línea, columna, texto seleccionado, y el nombre del símbolo CUP para cada token devuelto por la salida estándar.

4.2.6 Compatibilidad BYacc/J

Deberías leer la sección 8.2 JFlex y BYacc/J si estas interesado en como comunicar tu escáner generado con Byacc/J.

- **%byacc**
  La directiva %byacc habilita el modo de compatibilidad BYacc/J y es equivalente a las siguientes directivas:

```
%integer
%eofval{
        return 0;
%eofval}
%eofclose
```

4.2.7 Generación de código

Las siguientes opciones definen que tipo de código para el analizador léxico se producirá. **%pack** es la opción por defecto y será usada, cuando no sea especificado un método de generación de código.

- **%switch**
  Con **%switch** JFlex generará un escáner que tiene codificado el DFA. Este método da una buena compresión en términos de tamaño de la clase **.class** compilada mientras todavía provee una buena mejora. Si tu escáner es muy grande (más de 200 estados) la mejora puede verse claramente desmejorada y deberías considerar usar una de las directivas **%table** o **%pack**. Si tu escáner es aún mayor (cerca de 300 estados), el compilador Java **javac** podría producir código corrupto, esto petaría cuando lo ejecutes o le des un **java.lang.VerifyError** cuando sea chequeado por la máquina virtual. Esto limita a 64K el tamaño de los métodos como especifica la máquina virtual [10]. En este caso te verás formado a usar la directiva **%pack**, **%switch** usualmente provee más compresión en la tabla DFA que la directiva **%table**.

- **%table**
  La directiva **%table** causa que JFlex produzca una tabla clásica que conducirá al escáner a codificar su tabla DFA en una tabla. En este modo, JFlex solo hace una pequeña parte de la compresión de la tabla (ver [6], [12], [1] y [13] para más detalles en el tema de la compresión de la tabla) y usa el mismo método que JLex uso en su versioón 1.2.1. Ver sección 6 *Mejoras* de este manual para complara estos métodos. La misma razón que antes (limitación a 64 KB de los métodos) causa el mismo problema, cuando el escáner es demasiado grande. Esto es, porque la maquina virtual trata las inicializaciones estáticas de matrices como los métodos normales. En este caso estarás forzado a usar la directiva **%pack** para eliminar el problema.

- **%pack**
  %pack causa que JFlex comprima la tabla DFA genera y la guarde en uno más cadenas de literales. JFlex tiene cuidado que las cadenas no sean más largas de lo establecido dado el formato del fichero de la clase. Las cadenas serán desempacadas cuando el primer objeto del escáner es creado e inicializado. Después de desempacar el acceso interno a la tabla DFA es exactamente el mismo que en la opción **%table** - el único trabajo extra que será realizado en tiempo de

ejecución es el proceso de desempacado el cual es bastante rápido (no se nota en los casos normales). Esto en tiempo de complejidad proporcional al tamaño de la tabla DFA expandida, y si es estática, es hecho solo una vez para una cierta clase escáner - no importa con que frecuencia sea instanciada. De nuevo, ver sección 6 Mejoras en la mejora de esos escáneres. Con **%pack** no habrá prácticamente limitación para el tamaño del escáner. **%pack** es la opción por defecto y será usada cuando no sea especificado el método de generación de código.

4.2.8 Conjuntos de carácteres

- **%7bit**
  Causa que el escáner generado use una entrada con un conjunto de caracteres de 7 bits (códigos de carácter entre 0-127). Si un carácter de entrada con un código superior a 127 es encontrado en una entrada en tiempo de ejecución, el escáner lanzará un **ArrayIndexOutofBoundsException**. No por esto, debería considerar usar la directiva **%unicode**. Ver también sección 5.

- **%full**
  8 bit
  Ambas opciones causan que el escáner generado utilice un conjunto de caracteres de 8 bits (códigos de carácter en el intervalo 0-255). Si un carácter de entrada con un código mayor de 255 es encontrado como entrada en tiempo de ejecución, el escáner lanzará una **ArrayIndexOutofBoundsException**. Note que incluso en su plataforma usa solo un byte por carácter, el valor Unicode de un carácter puede ser mayor de 255. Si estás escaneando archivos de texto, deberías considerar usar la directiva **%unicode**. Ver también sección 5 para información acerca de la codificación de caracteres.

- **%unicode**
  %16bit
  Ambas opciones causan que el escáner generado use el conjunto de caracteres Unicode de 16 bit (códigos de carácter entre 0-65535). No habrá desbordamiento en tiempo de ejecución cuando uses este tipo de caracteres de entrada. **%unicode** no significa que el escáner leerá dos bytes al mismo tiempo. Que es una lectura y que constituye un carácter depende de la plataforma donde se ejecute. Ver también sección 5 para información acerca de la codificación de caracteres.

- **%caseless**
  %ignorecase
  Esta opción causa que JFlex no distinga entre mayúscula y minúscula. Esto habilita una forma fácil de especificar un escáner para un lenguaje insensible a las palabras. La opción **%caseless** no cambia el texto marcado y no tiene efecto en las clases de carácter. Que letras son mayúsculas y minúsculas, viene definido por el Unicode estándar y determinado por JFlex con los métodos de Java **Character.toUpperCase** y **Character.toLowerCase**.

4.2.9 Líneas, carácter y contado de columnas

- **%char**
  Habilita el contado de caracteres. La variable miembro **entera yychar** contiene el número de caracteres (empezando con 0) para el principio de la entrada del principio de token actual.

- **%line**
  Habilita el contado de líneas. La variable miembro **entera yyline** contiene el número de líneas (empezando en 0) del principio de la entrada del token actual.

- **%colum**
  Habilita el contado de columnas. La variable miembro **entera yycolum** contiene de caracteres (empezando con cero) del principio de la línea actual al principio del token actual.

4.2.10 Opciones obsoletas de JLex

- **%notunix**
  Esta opción de JLex está obsoleta en JFlex pero todavía se reconoce como una directiva válida. Solía usarse para cambiar entre los tipos de fin de línea de Windows y Unix (**\r\n** y **\n**) para el operador **$** en expresiones regulares. JFlex siempre reconoce los terminadores de línea dependientes de la plataforma.

- **%yyeof**
  Esta opción de JLex está obsoleta en JFlex pero todavía se reconoce como una directiva válida. En JLex declara un miembro público constante **YYEOF**. JFlex lo declara en cualquier caso.

4.2.11 Declaraciones de estado

Las declaraciones de estado son de la siguiente forma:
%s[tate] "identificador de estado" [, "identificador de estado", ...] para inclusivos o
%x[state] "identificador de estado" [, "identificador de estado", ...] para exclusivos
Si hubiese más de una línea de declaraciones de estado, cada una comenzando **%state** o **%xstate** (el primer carácter es suficiente, **%s** y **%x** también funcionan). Los identificadores de estado son letras seguidas por una secuencia de letras, dígitos y subguiones. Los identificadores de estado pueden estar separados por espacios en blanco o comas.

La secuencia
%state STATE1
%xstate STATE3, XYZ, STATE_10
%state ABC STATE5
declara el conjunto de identificadores **STATE1**, **STATE3**, **XYZ**, **STATE_10**, **ABC**, **STATE5** como estados léxicos, **STATE1**, **ABC**, **STATE5** como inclusivos, y **STATE3**, **XYZ**, **STATE_10** como exclusivos. Ver también sección 4.3.3 en la forma en la cual los estados léxicos influyen en el reconocimiento de la entrada.

4.2.12 Definiciones de macros

La definición de una macro tiene la forma
identificador de macro = expresión regular
Esto significa, que la definición de una macro es un identificador de macro (letra seguida por una secuencia de letras, dígitos o subguiones), que pueden ser usadas más tarde para referenciar la macro, seguido por un espacio en blanco opcional, seguido por un **'='**, seguido por un espacio en blanco opcional, seguido de una expresión regular (ver sección 4.3 *Reglas léxicas* para más información acerca de las expresiones regulares).

La expresión regular debe estar bien formada y no puede contener los operadores ^, / o $. **De manera distinta a JLex, las macros no son solo piezas de texto que se expanden por copia**- son parseadas y deben estar bien formadas.
Esto es una característica. Elimina muchas dificultades al encontrar errores en especificaciones léxicas (como si no tuviesen paréntesis en las macros más complicadas – lo cuales no son necesarios con JFlex). Ver sección 7.1 *Portando desde JLex*.

Desde que se permite hacer uso de macros en las definiciones, es posible usar una gramática como notación para especificar la estructura léxica deseada. Las macros solo son abreviaciones de las expresiones regulares que representas. Hay terminales de una gramática pero no pueden ser usados recursivamente de ninguna manera. JFlex detecta ciclos en las definiciones de macros y las reporta en tiempo de generación. JFlex también avisa acerca de macros que han sido definidas pero no se utilizan en la sección de las reglas léxicas de la especificación.

4.3 Reglas léxicas

La sección de las reglas léxicas de una especificación JFlex contiene un conjunto de expresiones regulares y acciones (código Java) que son ejecutados cuando el escáner equipara la expresión regular asociada.

4.3.1 Sintaxis

La sintaxis de la sección de reglas léxicas se describe en la siguiente gramática BNF (los símbolos terminales están encerrados entre comillas simples):

```
Regla_Léxica  ::= Regla+
Regla         ::= [Lista_de_Estados] ['^'] Expresión_Regular [Mirar_Alante]
                | [Lista_de_Estados] '<<EOF>>' Acción
                | Grupo_de_Estados
Grupo_de_Estados ::= Lista_de_Estados '{' Regla+ '}'
Lista_de_Estados  ::= '<' Identificador (`,` Identificador)* '>'
Mirar_Alante      ::= '$' | '/' Expresión_Regular
Acción            ::= '{' Código Java '}' | '|'

Expresión_Regular ::= Expresión_Regular '|' Expresión_Regular
                    | Expresión_Regular Expresión_Regular
```

```
                    | '(' Expresión_Regular ')'
                    | ('!'|'~') Expresión_Regular
                    | Expresión_Regular ('*'|'+'|'?')
                    | Expresión_Regular "{" Número ["," Número] "}"
                    | '[' ['^'] (Carácter | Carácter '-' Carácter)* ']'
                    | Clase_Predefinida
                    | '{' Identificador '}'
                    | '"' Cadena_de_Caracteres+ '"'
                    | Carácter

Clase_Predefinida ::= '[:jletter:]'
                    | `[:jletterdigit:]`
                    | '[:letter:]'
                    | '[:digit:]'
                    | '[:uppercase:]'
                    | '[:lowercase:]'
                    | '.'
```

La gramática usa los siguientes símbolos terminales:

- **Código Java**
  una secuencia de **bloques de código** como se describe en la especificación del lenguaje Java [7], sección 14.2.

- **Número**
  un entero decimal no negativo.

- **Identificador**
  una letra **[a-zA-Z]** seguida por una secuencia de ceros o más letras, dígitos o subguiones **[a-zA-Z0-9_]**

- **Carácter**
  una secuencia de escape o algún carácter unicode que no sea uno de los meta caracteres:
  | ( ) { } [ ] < > \ . * + ? ^ $ / . " ~ !

- **Cadenas de caracteres**
  una secuencia de escape o algún carácter unicode que no sea uno de los meta caracteres:
  \ "

- **Una secuencia de escape**

  - **\n \r \t \f \b**
  - una **\x** seguida por dos dígitos hexadecimales **[a-fA-F0-9]** (denota una secuencia de escape estándar ASCII),
  - una **\u** seguida por cuatro dígitos hexadecimales **[a-fA-F0-9]** (denota un secuencia de escape unicode),
  - una barra inversa seguida por tres dígitos octales desde 000 a 377 (denotando una secuencia de escape estándar ASCII), o
  - una barra inversa seguida por otro carácter unicode.

Dese cuenta que la secuencia de escape \n se refiere al carácter ASCII LF, no al final de línea. Si quisieras colocar el final de línea, deberías usar la expresión **\r|\n|\r\n** si prefieres las convenciones Java, o **\r|\n|\r\n|\u2028|\2029|\u000B|\u000C|\u0085** si quieres cumplir el Unicode al completo (ver también [5]).
Como en la versión 1.1 de JFlex los espacios en blanco **" "** y **"\t"** pueden ser usados para mejorar la legibilidad de las expresiones regulares. Serán ignorados por JFlex. En las clases de caracteres y en las cadenas, por contra los espacios en blanco se mantienen por si mismos (así que la cadena **" "** reconoce exactamente un carácter de espacio y **[\n]** reconoce un LF ASCII o un carácter espacio).
JFlex aplica las siguientes precedencias para las operaciones estándar (de mayor a menor):

- **operadores unarios postfijos ('*', '+', '?', {n}, {n,m})**
- **operadores unarios prefijos ('!', '~')**
- **concatenación (Expr_Reg ::= Expr_Reg Expr_Reg)**
- **union (Expr_Reg ::= Expr_Reg '|' Expr_Reg)**

Asi que la expresión **a|abc|!cd*** para instanciarlo es parseado como **(a|(abc))|((!c)(d*))**.

4.3.2 Semántica

Esta sección da una descripción informal de cada texto que es equiparado con una **expresión regular**.
Una expresión regular que consta solamente de

- un **Carácter** unifica este carácter.
- una clase de carácter **'[' (Carácter | Carácter '-' Carácter)* ']'** unifica un carácter en esa clase. Un **carácter** será considerado un elemento de la clase, si es listado en la clase o si su código reside dentro del rango de caracteres **Carácter '-' Carácter**.
  Así que **[a0-3\n]** al instanciarlo unifica los caracteres
  a 0 1 2 3 \n
  Si la lista de caracteres es vacía, la expresión no unifica con nada, ni siquiera con la cadena vacía. Esto podría ser útil en combinación con el operador de negación '!'.
- una clase de carácter negado **'[^' (Carácter | Carácter '-' Carácter)*** ']' unifica con todos los caracteres no listados en la clase. Si la lista de caracteres es vacía, la expresión unifica con cualquier carácter del conjunto de caracteres de entrada.
- una cadena **'"' Cadena_de_caracteres+ '"'** unifica exactamente con el texto encerrado en las dobles comillas. Todos los meta caracteres excepto \ y " pierden su significado especial dentro de una cadena. Ver más acerca de la opción **%ignorecase**.

- el uso de una macro **'{' Identificador '}'** unifica con cadenas que unifiquen con la macro del nombre **"Identificador"**.

- una clase de carácter predefinida unifica con cualquier carácter de la clase. Existen las siguientes clases de caracteres predefinidas
  . contiene todos los caracteres excepto **\n**.
  Todas las demás clases de caracteres predefinidas vienen en la especificación del Unicode de Java y determinada por funciones Java de la clase **java.lang.Character**.

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

  Son especialmente útiles cuando se trabaja con el juego de caracteres Unicode.

Si **a** y **b** son expresiones regulares, entonces

a | b (union)
es la expresión regular que reconoce una entrada **a** o una entrada **b**.

a b (concatenation)
es la expresión regular que reconoce la entrada **ab**

a* (cierre convexo ó cierre de kleene)
reconoce cero o más repeticiones de a

a+ (iteracion ó clausura positiva)
es equivalente **aa***

a? (opción)
reconoce lambda ó **a**

!a (negación)
reconoce todo excepto la entrada **a**. Úselo con cuidado: la construcción !a puede acarrear un tiempo exponencial en la transformación de NFA a DFA de la NFA para **a**. Note que con la negación y la unión tenemos (aplicando DeMorgan) intersección y diferencia de conjuntos: la intersección de **a** y **b** es **!(!a|!b)**, la expresión que reconoce **a** y no **b** es **!(!a|b)**

~a (todo hasta)
reconoce todo hasta la primera aparición de a. La expresión **~a** es equivalente a **!([^]* a [^]* | “”) a**. Un comentario tradicional de C es de la forma **“/*” ~”*/”**

a{n} (repetición)
es equivalente a concatenar **n** veces **a**. Así que **a{4}** es equivalente a la expresión **aaaa**. El decimal entero **n** debe ser positivo.

a{n,m} reconoce al menos **n** ocurrencias y como máximo **m** ocurrencias de **a**. Así que **a{2,4}** es equivalente a la expresión **aaa?a?**. Ambos **n** y **m** son enteros decimales no negativos y **m** no puede ser menor que **n**.

(a) reconoce lo mismo que **a**.

En un regla léxica, una expresión regular **r** debe estar precedida por un **‘^’** (el operador de comienzo de línea). Entonces **r** solo es reconocido al comienzo de la línea de entrada. Una línea comienza antes de cada ocurrencia de **\r | \n | \r\n | \u2028 | \u2029 | \u000B | \u000C | \u0085** (ver también [5]) y al principio de la entrada. El fin de línea precedente en la entrada no se procesa y puede ser reconocido por otra regla.
En un regla léxica, una expresión regular **r** debe estar seguida por una expresión precedida. Una expresión precedida es un **‘$’** o **‘/’** seguida por un expresión regular arbitraria. En ambos casos la expresión precedida no es añadida en el reconocimiento, pero es considerada mientras se determina que regla es la más larga que reconoce el texto (ver también 4.3.3 *Como se reconoce la entrada*).
En el caso **‘$’ r** solo es reconocida al final de la línea de entrada. El final de línea es denotado por la expresión regular **\r | \n | \r\n | \u2028 | \u2029 | \u000B | \u000C | \u0085**. Así que a$ es equivalente a **a / \r | \n | \r\n | \u2028 | \u2029 | \u000B | \u000C | \u0085**. Desde que en JFlex **$** es un verdadero contexto arrastrado, el final de fichero no cuenta como final de línea.
Para una precedencia arbitraria (también llamada contexto de fin) la expresión solo reconoce cuando es seguida por la entrada que empareja el contexto de fin. Desafortunadamente esta expresión no es de precedendia realmente arbitraria: en una regla **r1/r2**, o el texto representado por r1 tiene una longitud fija (e.g. si **r1** es un string) o el principio del contexto de fin **r2** no debe emparejar el final de **r1**. Así, el ejemplo **"abc" / "a" |"b"** es aceptable porque **"abc"** tiene una longitud fija. **"a"|"ab" / "x"*** es correcta porque ningún prefijo **"x" *** encaja con un postfijo de **"a"|"ab"**, pero **"x"|"xy" / "yx"** no es posible, porque el postfijo **"y"** de **"x"|"xy"** es también prefijo de **"yx"**. JFlex divulgará tales casos en el tiempo de generación. El algoritmo de Jflex usado actualmente para emparejar expresiones del contexto de fin es el descrito en [1].
En la versión 1.2, JFlex permite reglas **<<EOF>>** de estilo lex/flex en las especificaciones léxicas. Una regla
[StateList] <<EOF>> { código de acción }
es muy similar al comando de **%eofval** (sección 4.2.3). La diferencia reside en el **StateList** opcional que puede preceder a la regla **<< EOF >>**.

El código de la acción será ejecutado solamente cuando el final del fichero sea leído y el scanner esté actualmente en uno de los estados léxicos enumerados en **StateList**. El mismo **StateGroup** (ver la sección 4.3.3 *Cómo se reconoce la entrada*) y las reglas de prioridad como en el caso de regla normal se aplican (es decir si hay más de una regla **<<EOF>>** para cierto estado léxico, la acción que aparezca en primer lugar en la especificación será ejecutada). las reglas **<<EOF>>** eliminan los ajustes de las opciones de **%cup** y de **%byaccj** y no se deben mezclar con la directiva **%eofval**.
Una acción consiste en un trozo de código Java o es la acción especial |. Esta acción especial es una abreviatura para la acción de la siguiente expresión.
Ejemplos:

expresion1 **|**
expresion2 **|**
expresion3 ***{ acción }***

es equivalente de forma expandida a

expresion1 ***{ acción }***
expresion2 ***{ acción }***
expresion3 ***{ acción }***

Son de utilidad cuando se trabaja con expresiones de contexto de fin. La expresión **a | (c / d) | b** no es sintácticamente correcta, pero puede ser expresada fácilmente utilizando la acción | .

a **|**
c / d **|**
b **{acción }**

4.3.3 Cómo reconocer la entrada

Al recibir la entrada, el scanner determina la expresión regular que encaja con la porción de entrada más larga. Si hay más de una expresión regular que encaja con la porción más larga de entrada, el scanner elige la primera expresión que aparece en la especificación.
Después de determinar la expresión regular activa, se ejecuta la acción asociada a ésta. Si no hay expresión regular que encaje, el scanner termina el programa con un mensaje de error (si se ha utilizado el directorio de **%standalone**, el scanner imprime la entrada incomparable con **java.lang.System.out** en lugar de otro y finaliza el escaneado).
Los estados léxicos se pueden utilizar para restringir aún más el conjunto de las expresiones regulares que encajan con la entrada actual.

- Una expresión regular puede ser emparejada solamente cuando su conjunto asociado de estados léxicos incluye el estado léxico actualmente activo del scanner o si el conjunto de estados léxicos asociados es vacío y el estado léxico actualmente activo es inclusivo. Los estados exclusivos e inclusivos solamente se diferencian en este punto: reglas con un conjunto vacío de estados asociados.

- El estado léxico actualmente activo del explorador se puede cambiar dentro de una acción de una expresión regular usando el método **yybegin()**.

- El scanner comienza en el estado léxico inclusivo **YYINITIAL**, que es declarado siempre por defecto.

- El sistema de estados léxicos asociados a una expresión regular es el **StateList** que precede la expresión. Si una regla está contenida en uno o más **StateGroups**, entonces sus estados también se asocian a esa regla, es decir acumulan los **StateGroups**. Ejemplo:

```
%states A, B
%xstates C
% %
Expr1 { yybegin(A); action }
<YYINITIAL, A> expr2 { action }
<A> {
expr3 { action }
<B,C> expr4 { action }
}
```

  La primera línea declara dos estados léxicos (inclusivos) **A** y **B**, la segunda línea un estado léxico exclusivo **C**. El estado (inclusivo) **YYINITIAL** por defecto está siempre implícitamente allí, no necesita ser declarado. La regla con **expr1** no tiene ninguna lista de estados, y se empareja así con todos los estados salvo con los exclusivos, es decir **A**, **B**, y **YYINITIAL**. En su acción el scanner se cambia al estado **A**. La segunda regla **expr2** se puede verificar solamente cuando el scanner esté en el estado **YYINITIAL** o **A**. La regla **expr3** se puede emparejar solamente en el estado A y expr4 en los estados **A**, **B**, y **C**.

- Los estados léxicos se declaran y se utilizan como constantes **enteras** de Java en la clase generada bajo el mismo nombre que se utiliza en la especificación. No hay garantía de que los valores de estas constantes enteras sean distintos. Son punteros a la tabla generada de DFA, y si JFlex reconoce dos estados como léxicamente equivalentes (si se utilizan con el mismo conjunto de expresiones regulares), entonces las dos constantes tendrán el mismo valor.

4.3.4 La clase generada

JFlex genera exactamente un fichero conteniendo una clase de la especificación (a menos que se haya declarado otra clase en la primera sección de especificación).
La clase generada contiene (entre otras cosas) las tablas de DFA, un buffer de la entrada, los estados léxicos de la especificación, un constructor, y el método del scanner con las acciones suministradas al usuario.
El nombre de la clase es por defecto **Yylex**, habitualmente se utiliza con la directiva **%class** (véase también la sección 4.2.1). El buffer de entrada del analizador léxico está conectado con un flujo de entradas sobre el objeto de **java.io.Reader** que se pasa al analizador en el constructor generado. Si se desea proporcionar un constructor propio al analizador, se debe llamar siempre al generado en él para inicializar el buffer de entrada. Los buffer de entrada no deben ser accedidos directamente, solamente bajo el aviso de la API (véase también la sección 4.3.5). Su implementación interna puede cambiar sin aviso.

La interfaz principal al mundo exterior es el método de exploración generado (conocido por defecto como **yylex**, que por defecto devuelve el tipo **Yytoken**). La mayoría de sus aspectos son los habituales (el nombre, el tipo devuelto, las excepciones declaradas etc., consideradas también en la sección 4.2.2). Si se invoca, consumirá la entrada hasta que una de las expresiones de la especificación se empareja u ocurra un error. Si se empareja una expresión, se ejecuta la acción correspondiente. Puede devolver un valor del tipo especificado (en cuyo caso la saldremos del método de la exploración con este valor), o si no devuelve un valor, el explorador continúa consumiendo la entrada hasta que se empareja la expresión siguiente. Si el final del fichero se alcanza, el explorador ejecuta la acción del EOF, y (también sobre cada una de las llamadas al método de la exploración) devuelve el valor especificado de EOF (véase también la sección 4.2.3).

4.3.5 Métodos del explorador y campos accesibles en acciones (API)

Los métodos generados y campos miembros en el explorador JFlex son prefijados con yy para indicar que sean generados y permitir nombrar conflictos con código de usuario copiados dentro de la clase. Puesto que el código del usuario es parte de la misma clase, JFlex no tiene ninguna alternativa del lenguaje como el modificador **private** para indicar qué miembros y métodos son internos y que pertenecen al API. En su lugar, JFlex sigue un convenio de nombramiento: todo con una raya después del prefijo yy, como **yy_startRead**, debe ser considerado interno y conforme a cambio sin el aviso que lanza JFlex. Los métodos y los miembros de la clase generada que no contienen una raya pertenecen al API que la clase del explorador proporciona a los usuarios en el código de acción de la especificación. Serán estables y apoyados entre los lanzamientos de JFlex tanto como sea posible.
Habitualmente, la API consiste en los métodos y los campos miembros:

- **String yytext()**
  Devuelve la región del texto de entrada que encaja.

- **int yylength()**
  Devuelve la longitud de la región del texto de entrada que encaja (no requiere un objeto **String** para ser creada).

- **int yycharat(int pos)**
  Devuelve en la posición pos del texto que encaja. Es equivalente a **yytext().charAt(pos)**, pero más rápido. **pos** debe ser una valor entre 0 e **yylength()-1**.

- **void yyclose()**
  Cierra el flujo de entrada. Las sucesivas llamadas al método de la exploración devolverán el valor de end of file.

- **void yyreset(java.io.Reader reader)**
  Cierra el flujo de entrada actual, y resetea el escáner para un nuevo flujo de entrada. Todas las variables internas son reseteadas, el antiguo flujo de entrada no puede ser reutilizado (el contenido del

buffer interno se pierde). El estado léxico viene dado por **YY_INITIAL**.

- **void yypushStream(java.io.Reader reader)**
  Almacena el flujo de entrada actual en una pila, y lee de un nuevo flujo. El estado léxico, línea, carácter y contador de columna se vuelven inaccesibles. El flujo de entrada actual puede ser restablecido con **yypopstream** (usualmente en una acción **<<EOF>>**).

  Un ejemplo típico de lo anterior son los ficheros include del preprocesador C. Las crrespondientes especificaciones JFlex son algo así:

```
"#include" {FILE} {yypushStream(new FileReader(getFile(yytext()))); }
..
<<EOF>> { if (yymoreStreams()) yypopStream(); else return EOF; }
```

  Este método esta sólo disponible en el fichero **skeleton.nested**. Se encuentra en el directorio **src** de JFlex.

- **void yypopStream()**
  Cierra el flujo de entrada actual y continua la lectura desde la cima de la pila de flujos. Este método esta sólo disponible en el fichero **skeleton.nested**. Se encuentra en el directorio **src** de JFlex.

- **boolean yymoreStreams()**
  Devuelve true si y solo si aún hay flujos dejados por **yypopStream** para ser leídos en la pila de flujos.
  Este método esta sólo disponible en el fichero **skeleton.nested**. Se encuentra en el directorio **src** de JFlex.

- **int yystate()**
  Devulve el estado léxico actual del escáner.

- **void yybegin(int lexicalState)**
  Introduce el estado léxico l**exicalState**.

- **void yypushback(int number)**
  Desplaza **number** caracteres del texto que ha encajado en el flujo de entrada. Serán leídos de nuevo en la siguiente llamada del método de exploración. El número de caracteres que serán leídos de nuevo debe ser no mayor a la longitud del texto que encaja. Los caracteres desplazados no serán incluidos en **yylength** ni en **yytext()** tras la llamada a **yypushback**. Notar que en Java las cadenas son invariantes. Una acción como esta

```
String matched = yytext();
yypushback(1);
return matched;
```

  devolverá todo el texto que reconoció, mientras que

```
yypushback(1);
return yytext();
```

  devolverá el texto que encajó salvo el último carácter.

- **int yyline**
  Contiene la línea actual de la entrada (comenzando por cero, solo se activa con la directiva **%line**).

- **int yychar**
  Contiene el contador de carácter de la entrada actual (comenzando por cero, solo se activa con la directiva **%char**).

- **int yycolumn**
  Contiene el contador de columna de la línea actual (comenzando por cero, solo se activa con la directiva **%column**).

5 Codificaciones, Plataformas y Unicode

Este capítulo tratará de dar un poco de luz sobre asuntos de Unicode y las codificaciones, y cómo negociar con datos binarios. Mis gracias van a Stephen Ostermiller para su entrada en este tema.

5.1 El Problema

Antes de introducirnos en detalle, echemos un vistazo a qué es el problema. El problema es la independencia de plataforma en Java. Para escáneres la parte interesante acerca de la independencia de plataforma son las codificaciones de caracteres y cómo son manipuladas.
Si un programa lee un archivo de disco, obtiene un flujo de bytes. Hace un tiempo, cuando la hierba era verde, y el mundo mucho más sencillo, todo el mundo sabía que el valor del byte es 65. No fue problemático ver qué bytes se referían a qué caracteres. El alfabeto latino normal sólo tiene 26 caracteres, de forma que 7 bits o 128 valores distintos son suficientes para abarcarlos, incluso si nos damos el lujo de letra mayúscula y minúscula. Hoy día, las cosas son diferentes. El mundo repentinamente creció mucho, y las personas quisieron disponer de toda clase de caracteres especiales, dado que los usaban en sus lenguajes y escrituras. Aquí empiezan los problemas. Desde que los 128 valores distintos estaban ya cubiertos, se empezó a usar los 8 bits del byte, y se expandieron los mapeos de byte /carácter para satisfacer las nuevas necesidades. Algunas personas por ejemplo pudieron haber dicho "usemos el carácter 213 del alemán ä". Otros pudieron haberse encontrado con que el carácter 213 era é, porque no necesitaban al alemán y escribieron francés en lugar de eso. Mientras se usa un programa y ficheros de datos en una plataforma no es problemático, y todo se hace consistentemente.
Ahora entra en juego Java y repentinamente aparece problema: ¿Cómo consigo que el mismo programa vea ä en un cierto byte cuando se ejecute en Alemania y vea é para este byte si ejecuta en Francia?
La solución de Java para esto es usar Unicode internamente. Unicode tiene la intención de ser un súperconjunto de todos los conjuntos de caracteres conocidos y es por consiguiente una base perfecta para codificar cosas que podrían ser usadas en el mundo entero. Para hacer operar cosas correctamente, hemos de saber de dónde somos y cómo trazar un mapa de valores de byte para los caracteres Unicode y viceversa.

5.2 Escaneando archivos de texto

Escanear archivos de texto es la aplicación estándar para los escáneres como JFlex. Por eso es el más conveniente. Supongamos que trabajamos en una plataforma X, escribimos nuestra especificación del analizador léxico, puede usar cualquier carácter Unicode en él y compilar el programa. Sus usuarios trabajan en cualquier plataforma Y (posiblemente pero no necesariamente algo diferente de X), escriben sus archivos de entrada en Y y ellos ejecutan su programa en Y. No hay ningún problema.
Java hace esto como sigue: Si queremos leer cualquier cosa en Java que supuestamente contenga un texto, entonces usaremos un **FileReader** o algún **InputStream** conjuntamente con un **InputStreamReader**.
InputStreams devuelve los bytes "crudos", el **InputStreamReader** convierte los bytes en caracteres Unicode con la plataforma de codificación por defecto. Si un archivo de texto es producido en la misma plataforma, entonces la plataforma de codificación por defecto debería hacer el mapeo correctamente. Desde que el JFlex también usa lectores y Unicode internamente, este mecanismo también opera para las especificaciones del escáner. Si escribimos una A en el editor de texto y el editor usa la codificación de la plataforma (la A es 65), entonces luego Java traduce esto en la A lógica Unicode internamente. Si un usuario escribe una A en una plataforma completamente diferente (la A es 237 allí), entonces luego Java también traduce esto en la A lógica Unicode internamente. El mapeo es realizado después de esa traducción.
Por este mapeo de bytes para los caracteres, siempre debemos usar la opción del %unicode en la especificación de nuestro analizador léxico si queremos mapear archivos de texto. **%8bit** no es suficiente, incluso si sabemos que nuestra plataforma sólo usa 1 byte por carácter. La codificación Cp1252 usó en muchas máquinas Windows 256 caracteres, pero el carácter ´ con Cp1252 de código **\x92** tiene el valor Unicode **\u2019**, lo cual es mayor que 255 y provocaría un lanzamiento del escáner de **ArrayIndexOutOfBoundsException** si es encontrado.
Así para el caso usual no hay que hacer nada excepto usar la opción del %unicode en su especificación del analizador léxico.
Las cosas pueden ir mal cuando se produce un archivo de texto en la plataforma X y se consume en una diferente Y. Digamos que tenemos un archivo escrito en un PC Windows usando la codificación Cp1252. Luego movemos este archivo a un PC Linux con codificación ISO 8859-1 y allí ejecutamos su escáner en él. Java ahora piensa que el archivo está codificado en ISO 8859-1 (la codificación de la plataforma por defecto) cuando realmente está codificado en Cp1252. La mayoría de caracteres Cp1252 e ISO 8859-1 son los mismos, pero para los valores de byte **\x80** y **\x9f** disienten: ISO 8859-1 no está definido allí. Podemos resolver el problema diciéndole a Java explícitamente cuál codificación usar. Al construir **InputStreamReader**, podemos dar la codificación como argumento. La línea

Reader r = new InputStreamReader (input, "Cp1252");

resolverá el problema.
Por supuesto que la codificación a usar también puede provenir de los mismos datos: por ejemplo, cuando mapeamos una página HTML, se puede haber incrustado información acerca de su codificación de carácter en los encabezados.

Más información acerca de las codificaciones, cuáles son soportadas, cómo son llamadas, y cómo colocarlas puede ser encontrada en la documentación oficial Java en el capítulo acerca de la internacionalización. El enlace **http://java.sun.com/j2se/1.3/docs/guide/intl** conduce a una versión online de esto para Sun JDK 1.3.

5.3 Escaneando binarios

Mapeando binarios es a la vez más fácil y más difícil que archivos exploradores del texto. Es más fácil porque queremos los bytes "crudos" y no su significado, o se realizará ninguna traducción. Es más difícil porque no es fácil no obtener traducción cuando usamos a los lectores Java.
El problema (para binarios) es que los escáneres del JFlex son diseñados para trabajar en texto. Por eso la interfaz es la clase **Reader** (hay un constructor para instancias **InputStream**, pero está ahí mismo para la conveniencia y envuelve a un **InputStreamReader** alrededor de él para traer a los caracteres, no los bytes). Podemos obtener un escáner binario cuando escribimos su clase personalizada **InputStreamReader** que no hace explícitamente traducción, sino justamente copia valores de byte para los códigos de caracteres. Suena fácil, pero hay algunos problemas. En la especificación del escáner sólo podemos introducir códigos de caracteres positivos (para los bytes desde el **\x00** hasta **\xFF**). El tipo de byte de Java por otra parte es un entero de 8 bits con signo (- 128 para 127), así que tenemos que convertirlos en nuestro **Reader** personalizado. Si no estamos seguros, o la plataforma de desarrollo puede cambiar, es mejor usar el código de escape de caracteres en todos los lugares, dado que no cambia su significado.
Para ilustrar estos puntos, el ejemplo en **ejemplos/binary** contiene un escáner binario muy pequeño que trata de detectar si un archivo es un archivo de la clase Java. A este efecto mira si el archivo comienza con el número mágico **\xCAFEBABE**.

6 Unas pocas palabras sobre mejoras

Este capítulo da algunos resultados empíricos acerca de la velocidad de escáneres JFlex generados en contraste por las generadas por JLex, compara un escáner JFlex con uno escrito a mano, y presenta algunas técnicas sobre cómo hacer que nuestra especificación produzca un mapeo más rápido.

6.1 Comparación de JLex y JFlex

Los escáneres generados por la herramienta JLex son muy rápidos. No obstante es posible mejorar el funcionamiento de los escáneres generados usando JFlex. La siguiente tabla muestra los resultados que fueron producidos por la especificación del escáner de un lenguaje pequeño (el ejemplo del sitio Web de JLex) de programación de un juguete. El escáner fue generado usando JLex y tres métodos de generación JFlex. Luego fue ejecutado en un sistema W98 usando Sun 's JDK 1.3 con distintas entradas de prueba en ese lenguaje de programación del juguete.

Los valores presentados en la tabla denotan el tiempo de la primera llamada para el método explorador para devolver el valor EOF y la velocidad en tanto por ciento. Las pruebas fueron ejecutadas en el modo mixto JVM (HotSpot) y el modo interpretado puro. El modo mixto JVM causa un factor de mejora de funcionamiento.

KB	JVM	JLex	%switch	speedup	%table	speedup	%pack	speedup
496	hotspot	325 ms	261 ms	24.5 %	261 ms	24.5 %	261 ms	24.5 %
187	hotspot	127 ms	98 ms	29.6 %	94 ms	35.1 %	96 ms	32.3 %
93	hotspot	66 ms	50 ms	32.0 %	50 ms	32.0 %	48 ms	37.5 %
496	interpr.	4009 ms	3025 ms	32.5 %	3258 ms	23.1 %	3231 ms	24.1 %
187	interpr.	1641 ms	1155 ms	42.1 %	1245 ms	31.8 %	1234 ms	33.0 %
93	interpr.	817 ms	573 ms	42.6 %	617 ms	32.4 %	613 ms	33.3 %

El tiempo de mapeo del analizador léxico examinado en la tabla de arriba incluye acciones léxicas que a menudo necesitan crear instancias nuevas del objeto, la siguiente tabla exterioriza el tiempo de ejecución para la misma especificación con acciones léxicas vacías para comparar los motores puros del mapeo.

KB	JVM	JLex	%switch	speedup	%table	speedup	%pack	speedup
496	hotspot	204 ms	140 ms	45.7 %	138 ms	47.8 %	140 ms	45.7 %
187	hotspot	83 ms	55 ms	50.9 %	52 ms	59.6 %	52 ms	59.6 %
93	hotspot	41 ms	28 ms	46.4 %	26 ms	57.7 %	26 ms	57.7 %
496	interpr.	2983 ms	2036 ms	46.5 %	2230 ms	33.8 %	2232 ms	33.6 %
187	interpr.	1260 ms	793 ms	58.9 %	865 ms	45.7 %	867 ms	45.3 %
93	interpr.	628 ms	395 ms	59.0 %	432 ms	45.4 %	432 ms	45.4 %

El tiempo de ejecución de instrucciones simples depende de la plataforma y la implementación de la máquina virtual Java en la que el programa es ejecutado. Por consiguiente las tablas de arriba no pueden ser utilizadas como una referencia.

La siguiente tabla fue producida por la misma especificación léxica y la misma entrada en un sistema Linux también utilizando Sun 's JDK 1.3.
Con acciones:

KB	JVM	JLex	%switch	speedup	%table	speedup	%pack	speedup
496	hotspot	246 ms	203 ms	21.2 %	193 ms	27.5 %	190 ms	29.5 %
187	hotspot	99 ms	76 ms	30.3 %	69 ms	43.5 %	70 ms	41.4 %
93	hotspot	48 ms	36 ms	33.3 %	34 ms	41.2 %	35 ms	37.1 %
496	interpr.	3251 ms	2247 ms	44.7 %	2430 ms	33.8 %	2444 ms	33.0 %
187	interpr.	1320 ms	848 ms	55.7 %	958 ms	37.8 %	920 ms	43.5 %
93	interpr.	658 ms	423 ms	55.6 %	456 ms	44.3 %	452 ms	45.6 %

Sin acciones:

KB	JVM	JLex	%switch	speedup	%table	speedup	%pack	speedup
496	hotspot	136 ms	78 ms	74.4 %	76 ms	78.9 %	77 ms	76.6 %
187	hotspot	59 ms	31 ms	90.3 %	48 ms	22.9 %	32 ms	84.4 %
93	hotspot	28 ms	15 ms	86.7 %	15 ms	86.7 %	15 ms	86.7 %
496	interpr.	1992 ms	1047 ms	90.3 %	1246 ms	59.9 %	1215 ms	64.0 %
187	interpr.	859 ms	408 ms	110.5 %	479 ms	79.3 %	487 ms	76.4 %
93	interpr.	435 ms	200 ms	117.5 %	237 ms	83.5 %	242 ms	79.8 %

Aunque todos los mapeos del JFlex fueron más rápidos que los generados por JLex, las diferencias leves entre los métodos de generación de código del JFlex aparecen cuando son comparados en ejecución en el sistema W98. La siguiente tabla compara un escaneo escrito a mano para el lenguaje Java sacado del sitio Web de CUP con el escáner JFlex generado por Java que viene con JFlex en el directorio de **ejemplos**. Fueron probados en archivos diferentes .java en una máquina Linux con Sun's JDK 1.3.

lines	KB	JVM	handwritten scanner	JFlex generated scanner	
19050	496	hotspot	824 ms	248 ms	235 % faster
6350	165	hotspot	272 ms	84 ms	232 % faster
1270	33	hotspot	53 ms	18 ms	194 % faster
19050	496	interpreted	5.83 s	3.85 s	51 % faster
6350	165	interpreted	1.95 s	1.29 s	51 % faster
1270	33	interpreted	0.38 s	0.25 s	52 % faster

Un ejemplo de un escáner escrito a mano considerablemente más lento que el equivalente generado no es prueba de que todos los escáneres generados sean más rápidos que los escritos a mano. Es claramente imposible probar algo parecido. Desde un punto de vista de ingeniería del software sin embargo, no hay excusa para escribir a mano un escáner dado que esta tarea toma más tiempo, es más dificultoso y por consiguiente más propenso a errores, que escribir de forma compacta, legible y fácil para cambiar especificación léxica.

6.2 Cómo escribir una especificación JFlex más rápido

A pesar de que los escáneres JFlex generados funcionan correctamente sin optimizaciones especiales, hay ciertas heurísticas que los pueden hacer aun más rápidos. Esos son (en orden de ganancia de funcionamiento):

- Evite reglas que requieren Backtracking

Desde la página principal flex C/C++ [11]: "deshacerse de Backtracking supone desorden y a menudo puede ser una cantidad de trabajo enorme para un escáner complicado". Backtracking es introducido por la regla más larga y ocurre por ejemplo en este conjunto de expresiones:

"averylongkeyword"

Con la entrada **"averylongjoke"** el escáner tiene que leer cuidadosamente todos los caracteres para decidir si la regla debe ser realizada. Todos los caracteres de **"verylong"** tienen que ser leídos otra vez para el siguiente proceso. Backtracking puede ser evitado en general añadiendo reglas de error que encajen con esas condiciones de error

"av" | "ave" | "avery" | "averyl"..

Mientras esto poco práctico en la mayoría de escáneres, hay posibilidad de añadirle una regla para una lista larga de palabras claves

"keyword1" { return symbol(KEYWORD1); }
..
"keywordn" { return symbol(KEYWORDn); }
[a-z]+ { error("not a keyword"); }

La mayoría de escáneres de lenguaje ya tienen una regla como esta para algunos tipos de identificadores variables de longitud.

- Evitar contadores de línea y columna.
  Cuesta múltiples comparaciones adicionales por carácter de entrada y el texto tiene que ser remapeado para contar. En la mayoría de escáneres se logra contar las líneas incluyendo en la especificación un terminador de línea incrementando yyline cada vez que se encuentre. El contar columnas también podría ser incluido en acciones. Esto será más rápido, pero podrá en algunos casos volverse muy desordenado.

- Evitar expresiones lookahead y el operador de fin de línea '$'.
  El contexto debe ser leído y luego (por no haberse consumido) leído otra vez.

- Evitar el operador de comienzo de línea '^'
  Esto cuesta comparaciones adicionales múltiples. A veces son necesarios caracteres lookahead (cuando la última lectura de carácter es \r el escáner tiene que leer un carácter delante para comprobar si el siguiente es un \n o no).

- Hacer encajar tanto texto como sea posible en una regla.
  Una regla es reconocida en las iteraciones propias del escáner. Después de cada acción es necesario reestablecer estados internos del escáner si es preciso.

Note que escribir más reglas en una especificación no retarda el escáner generado (exceptuando cuando hay que cambiar de decisión para otro método de generación de código por el tamaño mayor).
Las dos reglas principales de optimización aplicadas también en especificaciones léxicas son:

1. no lo hagas

2. (Sólo para expertos) no lo hagas todavía

Algunas pautas de funcionamiento contradicen un estilo legible y compacto de la especificación. En caso de duda o cuándo los requisitos no se cumplen: no usarlos, la especificación siempre puede ser optimizada en un estado posterior del proceso de desarrollo.

7 Exportando

7.1 Exportando desde JLex

JFlex fue diseñado para leer especificaciones JLex y generar un escáner que se comporta como uno generado por JLex con la única diferencia de ser más rápido.
Esto funciona como se espera en todas las especificaciones JLex bien estructuradas.
Echemos un vistazo a lo que entendemos por "bien estructurado". Una especificación JLex está bien estructurada cuando

- genera un escáner de trabajo con JLex

- no contiene los caracteres ! y ~ . Son operadores en JFlex mientras el que JLex los trata como caracteres normales de entrada. Fácilmente podemos exportar una especificación JLex a JFlex reemplazando cada ! con \ ! y cada ~ con \ ~ en todas las expresiones regulares.

- tiene sólo expresiones regulares completas rodeadas por paréntesis en el macro de definiciones. De otra manera podría ser un problema principal. En JLex, un lado derecho de un macro es justo un trozo de texto, éste es copiado para el punto donde el macro es usado. Con esto, si tuviésemos

  ```
  macro1 = ("hello"
  macro2 = {macro1})*
  ```

  sería posible (con macro2 expandiéndose para **("hello")***). Esto no se permite JFlex y tendremos que transformar estas definiciones. Hay algunos errores sutiles que pueden ser introducidos por los macros del JLex. Consideremos una definición como **macro = a | b** y la usaremos con **{macro}***. Esta expansión en JLex nos lleva a la expresión **a | b *** y no a la pretendida **(a | b)***.
  El JFlex usa siempre la segunda forma de expansión, dado que ésta es la forma natural de pensar acerca de abreviaciones para expresiones regulares.
  La mayoría de especificaciones no deberían sufrir este problema, porque los macros a menudo sólo contienen clases de carácter como el **alpha = [a - zA – Z]** y definiciones más peligrosas como

ident = {alpha}({alpha}|{digit})*

usadas para escribir reglas como

{ident} { .. action .. }

y expresiones no más complejas como

{ident}* { .. action .. }

donde el tipo de error presentado arriba aparece.

7.2 Exportando desde lex/flex

Esta sección intenta dar una visión general de actividades y problemas posibles al exportar una especificación léxica desde las herramientas lex y flex [11] de C/C++ disponibles en la mayoría de sistemas Unix para JFlex.
La mayor parte de las características C/C++ no están presentes en JFlex, pero la mayoría de las especificaciones léxicas lex/flex "limpias" pueden ser exportadas a JFlex sin demasiado trabajo.
Este capítulo no profundiza demasiado en este tema y se basa principalmente en mi experiencia personal. Si encuentra problemas, dispone de mejores soluciones para las proposiciones aquí tratadas o tenga simplemente algunas ideas que compartir, contacte conmigo por email: Gerwin Klein <lsf@jflex.de>. Incorporaré sus experiencias en este manual (con todo el crédito debido para usted, por supuesto).

7.2.1 Estructura Básica

Una especificación léxica tiene la siguiente estructura básica:

definitions
% %
rules
% %
user code

El capítulo de código del usuario contiene algún código C usado en acciones de la parte de **reglas** de la especificación. Para JFlex la myoría de ese código tendrá que ser incluido en la directiva de la clase código **%{..%}** en la sección de **opciones y declaraciones** (después de traducir el código C para Java, por supuesto).

7.2.2 Macros y Sintaxis de Expresiones Regulares

La sección de **definiciones** de una especificación flex es realmente similar a la parte de las **opciones y declaraciones** de las especificaciones de JFlex.
Definiciones de macros en flex tienen la forma:
<identifier> <expression>
Para exportárlos a los macros de JFlex, basta insertar **a = between <identifier > y < expression>**.

La sintaxis y la semántica de expresiones regulares en flex son casi lo mismo que en JFlex. Hemos de ser cautos con algunas secuencias de escape presentes en flex (como \a) que no son soportadas en JFlex. Estas secuencias de escape deberían ser transformadas en su equivalente octal o hexadecimal.
Otro punto es la clase carácter predefinida. Flex ofrece los directamente soportados por C, JFlex ofrece los soportados por Java. Estas clases algunas veces tendrán que estar listadas manualmente (si hay necesidad para este rasgo, entonces puede ser implementada en una versión futura del JFlex).

7.2.3 Reglas Léxicas

Dado que flex se basa en su mayor parte en Unix, los operadores **'^'** (el comienzo de línea) y **'$'** (el fin de línea), se considera el carácter del **\n** como único terminador de la línea. Esto usualmente debería causar problemas, pero debemos prepararnos para las ocurrencias de **\r** o **\r\n** o uno de los caracteres **\u2028**, **\u2029**, **\u000B**, **\u000C**, o **\u0085**. Son considerados terminadores de línea en Unicode y por consiguiente no pueden ser consumidos cuando ^ o $ esté presente en una regla.
El algoritmo de contexto de arrastre de flex es mejor que el usado en JFlex. Por eso las expresiones del lookahead podrían causar mayores dolores de cabeza. El JFlex emitirá un mensaje de error en tiempo de generación, si no puede generar un escáner para una cierta expresión del lookahead. (Lo siento, no tengo más ideas al respecto aún. Si alguien sabe cómo los funciona el algoritmo del lookahead de flex y cómo implementarlo eficientemente, por favor contactar conmigo).

8 Trabajando conjuntamente

8.1 JFlex y CUP

Una de las principales metas de diseño de JFlex fue hacer un interfaz con el generador parser de Java CUP [8] tan fácil como fuera posible. Esto se ha logrado dándole a la directiva **%cup** un sentido especial. Una interfaz no importa siempre tiene dos partes. Este capítulo se centra en el lado de la historia de CUP.

8.1.1 CUP versión 0.10j

Desde la versión CUP 0.10j, ésta ha sido simplificada enormemente por la interfaz nueva del escáner de CUP **java_cup.runtime.Scanner**. Los escáneres léxicos JFlex ahora implementan esta interfaz automáticamente cuando **%cup** es usado. No hay **código de analizador** gramatical en especial, **código de inicialización** o de **mapeo** con más opciones que cualquier analizador gramatical CUP. Usted puede concentrarse en su gramática.
Si su escáner léxico generado tiene el nombre de clase Scanner, entonces el analizador gramatical comienza desde un programa principal como este:

```
...
try {
```

```
parser p = new parser(new Scanner(new FileReader(fileName)));

Object result = p.parse().value;
}
catch (Exception e) {
…
```

8.1.2 Usando especificaciones existentes de JFlex/CUP con CUP

Si usted ya tiene una especificación existente y le gustaría actualizar ambos JFlex y CUP con las nuevas versiones, probablemente tendrá que ajustar su especificación.
La diferencia principal entre **%cup** en JFlex 1.2.1 e inferiores, y la versión actual de JFlex es que los escáneres JFlex ahora implementan automáticamente la interfaz **java_cup.runtime.Scanner**. Esto quiere decir, que la función exploradora cambia su nombre de **yylex()** por **next_token()**.
La diferencia principal con versiones mayores de CUP para 0.10j es que CUP ahora tiene un constructor predeterminado que acepta un **java_cup.runtime.Scanner** como argumento y eso usa este escáner por defecto (ningún escáner con código es necesario).
Si usted tiene una especificación CUP existente, entonces probablemente se parecerá algo a esto:

```
parser code {:
        Lexer lexer;
        public parser (java.io.Reader input) {
                lexer = new Lexer(input);
        }
:};
```

escanear con {: return lexer.yylex(); :};

Para actualizar CUP 0.10j, habría que cambiar lo anterior por:

```
parser code {:
        public parser (java.io.Reader input) {
                super(new Lexer(input));
        }
:};
```

Si usted no cambia el método que llama el analizador gramatical, entonces podría eliminar el constructor enteramente (y si no hay ninguna cosa en él, la sección completa de código de analizador gramatical igualmente). El procedimiento principal invocador construiría el analizador gramatical como se muestra arriba.
La especificación JFlex no necesita ser cambiada.

8.1.3 Usando versiones anteriores de CUP

A quién le guste o tenga que usar versiones anteriores de CUP, el siguiente capítulo explica la forma vieja. Por favor note, que el nombre estándar de la función exploradora con **%cup** no es **yylex()**, sino **next_token()**.

Si usted tiene una especificación del escáner que comienza así:

```
package PACKAGE;
import java_cup.runtime.*; /* conveniente, no necesario */

%%

%class Lexer
%cup
..
```

Luego lo encaja con una especificación CUP empezando así:

```
package PACKAGE;

parser code {:
        Lexer lexer;

        public parser (java.io.Reader input) {
                lexer = new Lexer(input);
        }
:};

scan with {: return lexer.next_token(); :};

..
```

Esto da por supuesto que el analizador gramatical generado obtendrá el analizador gramatical nombrado. Si no lo hiciese, tendrá que ajustar el nombre del constructor.

El analizador gramatical puede comenzar en una rutina principal como esto:

```
..
try {
        parser p = new parser(new FileReader(fileName));
        Object result = p.parse().value;
}
catch (Exception e) {
..
```

Si usted quiere que la especificación del analizador gramatical sea independiente del nombre del escáner generado, entonces usted en lugar de eso puede escribir una interfaz léxica

```
public interface Lexer {
        public java_cup.runtime.Symbol next_token() throws
                java.io.IOException;
}
```

cambie el código de analizador gramatical por:

```
package PACKAGE;
parser code {:
        Lexer lexer;

        public parser (Lexer lexer) {
                this.lexer = lexer;
        }
:};
scan with {: return lexer.next_token(); :};
..
```

usando la directiva de **%implements** haremos saber a JFlex el uso de la interfaz Lexer:

```
..
%class Scanner /* not Lexer now since that is our interface! */
%implements Lexer
%cup
..
```

y finalmente la rutina principal es:

```
...
try {
        parser p = new parser(new Scanner(new
FileReader(fileName)));
        Object result = p.parse().value;
}
catch (Exception e) {
...
```

Si usted quiere mejorar los mensajes de error producidos por los analizadores gramaticales CUP, puede pasar los métodos **report_error** y **report_fatal_error** en la sección de código del analizador gramatical " de la especificación CUP. Los nuevos métodos pueden usar **yyline** y **yycolumn** por ejemplo (almacenados en los miembros de izquierda y derecha de la clase **java_cup.runtime.Symbo**l) para proporcionar de forma más conveniente la situación del error. El lexer y el analizador gramatical para el lenguaje Java en el directorio **examples/java** de la distribución del JFlex usan este estilo en cuanto a errores.

8.2 JFlex y BYacc/J

JFlex fue creado con soporte para la extensión Java BYacc/J [9] por Bob Jamison para el analizador gramatical clásico Berkeley Yacc. Este capítulo describe cómo hacer una interfaz entre BYacc/J con JFlex. Se fundamenta en muchas sugerencias útiles y hace comentarios de Larry Bell.

Dado que la la arquitectura de Yacc es algo diferente de la de CUP, la interfaz también trabaja de ligeramente distinta. BYacc/J espera una función **int yylex()** en la clase del analizador gramatical que devuelve el siguiente token. Los valores semánticos se esperan en un **yylval** del tipo **parserval** donde **"parser"** es el nombre de la clase generada por analizador gramatical.
Veamos un ejemplo pequeño,

```
%%
%byaccj
%{
/* Almacene una referencia para el objeto yyparser del analizador
gramatical * /
private parser yyparser;

/* El constructor lleva un objeto parser (analizador gramatical)
adicional * /
public Ylex(java.io.Reader r, parser yyparser) {
        this(r);
        this.yyparser = yyparser;
}

%}
NUM = [0-9]+ ("." [0-9]+)?
NL = \n | \r | \r\n

%%
/* operadores */

..
"+" |
..
"(" |
")" { return (int) yycharat(0); }

/* nueva línea */
{NL} { return parser.NL; }

/* float */
{NUM} { yyparser.yylval = new
          parserval(Double.parseDouble(yytext()));
          return parser.NUM; }
```

El analizador léxico espera una referencia para el analizador gramatical en su constructor. Desde que Yacc permite uso de caracteres terminales como **'+'** en sus especificaciones, devolveremos el código del carácter para reconocimientos simples de caracteres (por ejemplo los operadores en el ejemplo). Los nombres simbólicos de los token son almacenados como **constantes estáticas públicas enteras** en la clase generada del analizador gramatical. Son usados como el token NL del ejemplo anterior. Finalmente, para algunos tokens, un valor semántico puede tener que ser comunicado al analizador gramatical. La regla NUM demuestra esto.

Un ejemplo de especificación de analizador gramatical Byacc/J podría parecerse a esto:

```
%{
      import Java.io.*;
%}

%token NL /* nueva línea */
%token <dval> NUM /* un número */

%type <dval> exp

%left '-' '+'
..
%right '^' /* potencia */

%%


..


exp: NUM { $$ = $1; }
| exp '+' exp { $$ = $1 + $3; }
..
| exp '^' exp { $$ = Math.pow($1, $3); }
| '(' exp ')' { $$ = $2; }
;
%%
      /* referencia al objeto lexer */
      private Yylex lexer;
      /* interface del lexer */
      private int yylex () {
            int yyl_return = -1;
            try {
                  yyl_return = lexer.yylex();
            }
            catch (IOException e) {
                  System.err.println("IO error :"+e);
            }
            return yyl_return;
      }
      /* error reporting */
      public void yyerror (String error) {
            System.err.println ("Error: " + error);
      }
      /* lexer se crea en el constructor */
      public parser(Reader r) {
            lexer = new Yylex(r, this);
      }
```

```
/* De este modo usamos parser */
public static void main(String args[]) throws IOException {
parser yyparser = new parser(new FileReader(args[0]));
yyparser.yyparse();
}
```

Aquí, la parte hecha a la medida está en su mayor parte en la sección de código de usuario: creamos el lexer en el constructor del analizador gramatical y almacenamos una referencia a él para el posterior uso en el método **int yylex()** del analizador gramatical. Este **yylex** en el analizador gramatical sólo llama al **int yylex()** del lexer generado y pasa de uno a otro el resultado. Si algo sale mal, entonces devolvemos -1 para indicar un error.
Las versiones ejecutables de las especificaciones anteriores se hallan en el directorio **examples/byaccj** de la distribución del JFlex.

9 Errores y Deficiencias

9.1 Deficiencias

El algoritmo de contexto de arrastre (lookahead) descrito en [1] y usado en JFlex es incorrecto. No surte efecto, cuando un postfijo de una expresión regular encaje con un prefijo del contexto de arrastre y el largo del texto con el que encaja la expresión regular no tenga un tamaño oportuno. El JFlex dará cuenta de estos casos como los errores en el tiempo de generación.

9.2 Errores

A día 8 de octubre de 2001 los problemas conocidos en JFlex son:

- El chequeo, si una expresión del lookahead es legal, falla en algunas expresiones. El algoritmo de lookahead funciona como se dijo anteriormente, pero JFlex no dará cuenta de todas las expresiones del lookahead que el algoritmo no pueda manipular en tiempo de generación. Algunos casos son percibidos por el chequeo, pero no todos.
  Workaround: chequear las expresiones del lookahead manualmente. Una expresión de lookahead **r1/r2** es adecuada, si ningún postfijo de **r1** encaja con un prefijo de **r2**.

Si encuentra errores nuevos, por favor use la sección de problemas no resueltos en el software del JFlex en su website[3] para dar cuenta de ellos.

10 Copias y Licencia

JFlex es software gratuito, publicado bajo las condiciones de GNU General Public License[4].

[3] http://www.jflex.de/
[4] http://www.fsf.org/copyleft/gpl.html

No hay GARANTÍA para JFlex, ni su código ni su documentación.

El código generado por JFlex deja el derecho de autor sobre especificación que produjo. Si el error fuese en su especificación, entonces usted puede usar el código generado sin restricciones.

Ver el **COPYRIGHT** para más información.

Referencias bibliográficas

[1] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools, 1986

[2] A. W. Appel, Modern Compiler Implementation in Java: basic techniques, 1997

[3] E.. Berk, JLex: A lexical analyser generator for Java, http://www.cs.princeton.edu/~appel/modern/java/JLex/

[4] K. Brouwer, W. Gellerich,E. Ploedereder, Myths and Facts about the Eficient Implementation of Finite Automata and Lexical Analysis, in: Proceedings of the 7th International Conference on Compiler Construction (CC '98), 1998

[5] M. Davis, Unicode Regular Expression Guidelines, Unicode Technical Report #18, 2000 http://www.unicode.org/unicode/reports/tr18/tr18-5.1.html

[6] P. Dencker, K. D□urre, J. Henft, Optimization of Parser Tables for portable Compilers, in: ACM Transactions on Programming Languages and Systems 6(4), 1984

[7]J. Gosling, B. Joy, G. Steele, The Java Language Specifcation, 1996, http://www.javasoft.com/docs/books/jls/

[8] S. E. Hudson, CUP LALR Parser Generator for Java, http://www.cs.princeton.edu/~appel/modern/java/CUP/

[9] B. Jamison, BYacc/J, http://troi.lincom-asg.com/~rjamison/byacc/

[10] T. Lindholm, F. Yellin, The Java Virtual Machine Speci_cation, 1996, http://www.javasoft.com/docs/books/vmspec/

[11] V. Paxon, ex - The fast lexical analyzer generator, 1995

[12] R. E. Tarjan, A. Yao, Storing a Sparse Table, in: Communications of the ACM 22(11), 1979

[13] R. Wilhelm, D. Maurer, □ Ubersetzerbau, Berlin 1997[2]