Autómatas y Lenguajes
Dpto. de Informática
UNSL
2011

ANÁLISIS LEXICOGRÁFICO*

1. Introducción

Generalmente un compilador es un programa complejo que se lo divide en varias partes o *fases* para simplificar su construcción, de manera que cada una de las fases son sólo parte del proceso de compilación. La estructura típica de un compilador es mostrada en la figura 1.

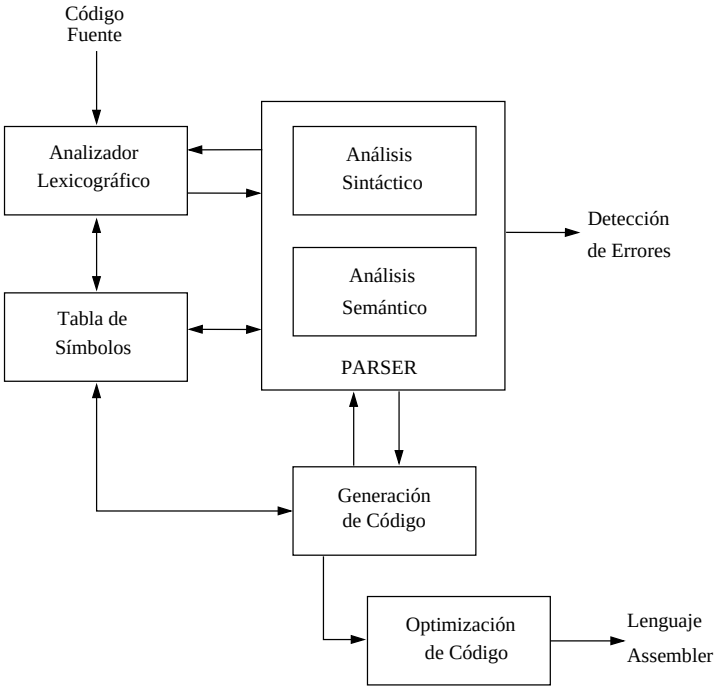


Figura 1: Estructura típica de un compilador

Un compilador se puede definir como un programa o grupo de programas que traducen código de un lenguaje a otro, en este caso el código fuente escrito en un lenguaje de programación de alto nivel, es traducido a lenguaje assembler. En este apunte se muestra como especificar e implementar analizadores lexicográficos.

*Extraido de "Compilers: Principles, Techniques, and Tools" Chapter 3 - A.V. Aho, R. Sethi, J.D. Ullman

2. Analizador Lexicográfico

Cada fase es un proceso independiente usado en el proceso de compilación. La fase del analizador lexicográfico o *scanner* transforma la entrada para que pueda ser usada por la fase del analizador sintáctico y semántico o *parser* y por las restantes fases del compilador.

Las técnicas usadas para implementar analizadores lexicográficos pueden ser aplicadas a otras áreas tales como lenguajes de consulta y sistemas de recuperación de información. El problema subyacente de cada una de las aplicaciones es la especificación y diseño de programas que ejecuten acciones guiadas por patrones o **patterns** que aparecen en las cadenas. Existe un *lenguaje de acción por patrón* llamado **Lex** que permite diseñar y especificar aplicaciones que realicen acciones guiadas por patrones tales como los analizadores lexicográficos. En el lenguaje de acción por patrón, las expresiones regulares permiten especificar los patrones y el compilador para el lenguaje Lex las convierte a Autómatas Finitos.

El analizador lexicográfico es la primera fase de un compilador. Tiene como tarea principal leer los caracteres de entrada y producir como salida una secuencia de unidades léxicas o **tokens** que serán utilizadas por el analizador sintáctico. En la figura 2 se esquematiza la interacción entre los analizadores. Generalmente, el analizador lexicográfico se implementa como una subrutina del analizador sintáctico. Cuando recibe un solicitud desde el analizador sintáctico de "deme próximo token", lee caracteres de la entrada hasta que pueda especificar un token y lo envía.

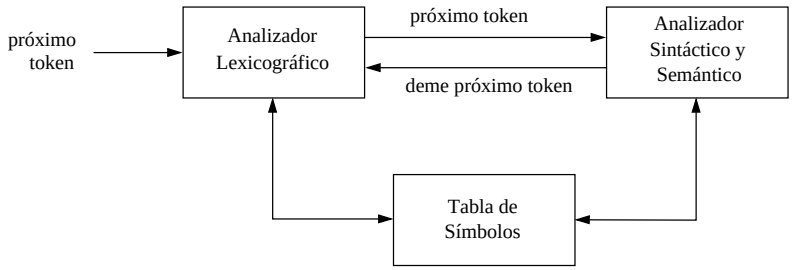


Figura 2: Análisis Lexicográfico

Dado que el analizador lexicográfico es la parte del compilador que lee el texto fuente, puede también llevar a cabo ciertas tareas de la interfase con el usuario. Una de las tareas es eliminar (descartar) los comentarios, espacios en blanco, tabs, caracteres newline, etc.

Hay varias razones para separar la fase de análisis de la compilación en análisis lexicográfico y análisis sintáctico:

1. La razón más importante es hacer un diseño más simple. La separación del análisis lexicográfico del análisis sintáctico permite a menudo simplificar las fases.

2. La eficiencia del compilador es mejorada. Un analizador lexicográfico separado nos permite construir un proceso especializado y potencialmente más eficiente, dado que se pierde una gran cantidad de tiempo leyendo el programa fuente y particionándolo en tokens.

3. Es posible mantener la portabilidad del compilador. Las particularidades del alfabeto de entrada pueden ser restringidas al analizador lexicográfico. Por ejemplo, símbolos especiales (↑ en Pascal) pueden ser aislado y tratados por el analizador lexicográfico.

2.1. Tokens, Patrones y Lexemas

Al hablar de Análisis Lexicográfico aparecen palabras tales como *tokens*, *patrones*, etc., las cuales tienen significados específicos. Un conjunto de cadenas de caracteres puede ser descripto por una regla llamada *patrón* (pattern), a la que se le asocia un *token*. Un patrón *unifica* con cada una de las cadenas de caracteres en el conjunto, de manera que el mismo token es asociado para cada una de las cadenas de caracteres del conjunto (ver figura 3). Un *lexema* es una secuencia de caracteres en el programa fuente que a través de un patrón unifica con un token.

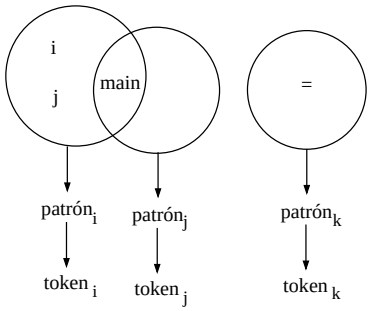


Figura 3: Relación entre secuencia de caracteres, patrones y tokens

Por ejemplo, la sentencia Pascal:

```
const pi = 3,1416;
```

es una cadena de caracteres, donde la subcadena de caracteres `pi` es un lexema para el token `identificador` o `id`.

No existe una relación uno a uno entre lexemas y tokens. Por ejemplo, un token `nombre` o `numero` puede tener asociados muchos lexemas posibles, mientras que un token `while` siempre se asocia con un único lexema. Sin embargo, la situación se complica para tokens que se "superponen", tales como `>`, `>=`, `>>`, etc. En general, un analizador lexicográfico reconoce el token que coincide con el lexema más largo, por ejemplo dada la entrada `>>`, será reconocido un token `shift` en vez de dos tokens `mayor`.

A continuación se muestran ejemplos de tokens y lexemas:

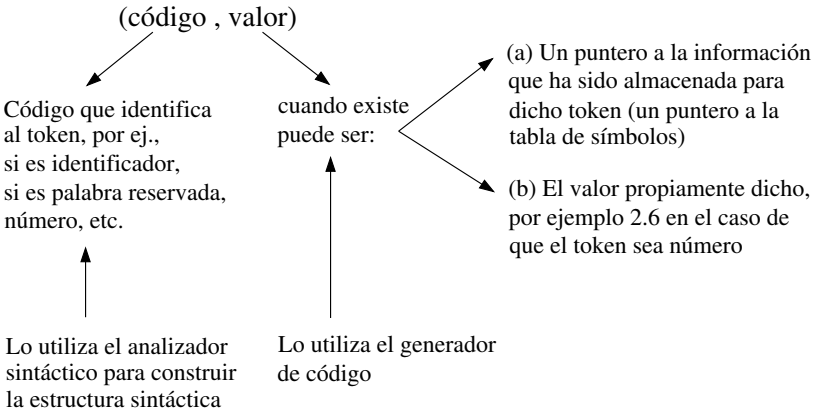
token	**lexemas**	**descripción informal del patrón**
`id`	`pi`, `count`, `D2`	letra seguida por letras o dígitos
`num`	`3.15`, `0.8`	cualquier constante numérica
`op rel.`	`<`,`<=`, `>`	< o <= o >

Para un analizador lexicográfico, los tokens representan los **símbolos terminales de la gramática subyacente**. Los tokens más habituales son:

- palabras reservadas
- identificadores
- operadores
- constantes

- símbolos de puntuación
- paréntesis

Cuando el analizador lexicográfico aisla un token envía al analizador sintáctico un código interno que lo representa. En general, un token suele representarse por un par:



Dependiendo de la existencia o no del segundo elemento del par, se puede dividir a los tokens en dos categorías:

a) (código,valor): identificadores, constantes, rótulos, etc.

b) (código,λ): palabras reservadas, operadores, signos, etc.

2.2. Diseño e Implementación de Analizadores Lexicográficos

Para realizar el diseño e implementación de analizadores lexicográficos es necesario contar con:

1. Algún método para describir los posibles patrones, que denotan los posibles conjuntos de cadenas de caracteres que pueden aparecer en la entrada, y que deben asociarse con los tokens. Las *expresiones regulares* son útiles para describir formalmente los patrones de todos los tokens que pueden aparecer en los lenguajes de programación.

2. Un reconocedor de tokens. Los *autómatas finitos* reconocen los lenguajes denotados por las expresiones regulares y pueden ser utilizados para reconocer los tokens descriptos por las expresiones regulares.

3. Generalmente, cuando se reconoce un determinado tipo de token se deben realizar acciones ya establecidas. Por lo cual, se les agregan acciones a los estados (a algunos o a todos o a ninguno) del autómata finito reconocedor de los tokens. Estos autómatas finitos se los denomina *autómatas finitos con acciones semánticas* (ver figura 4).

Construir un analizador lexicográfico no es una tarea dificultosa y debe integrar los tres puntos anteriores. El analizador debe encontrar o reconocer lexemas en la entrada determinando el límite entre cada lexema. Luego de delimitar un lexema le debe asociar el código del token que le corresponde.

A continuación se muestran distintas estrategias para tratar ciertos casos especiales donde puede existir dificultad para determinar el límite del lexema y/o el token asociado:

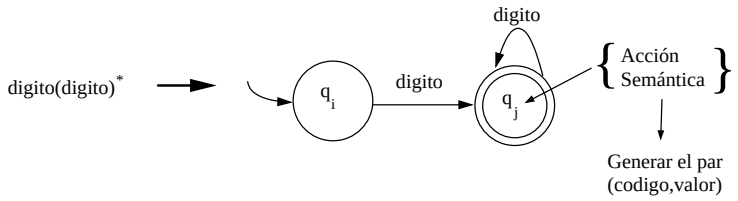


Figura 4: Representación de un Autómata Finito con Acciones Semánticas en los estados (considerar que `digito` corresponde con la expresión regular `(0+1+ ... +9)`)

a) *Palabras claves*: se tratan de distinta forma dependiendo de que sean reservadas o no:

1. *Si las palabras claves son* ***reservadas****:*
   Implica que no pueden usarse como nombres de identificadores (por ejemplo. `program` en Pascal). Su reconocimiento puede eliminarse como camino alternativo del autómata y tratarlas como identificadores. La diferencia es que primero se deben guardan las palabras reservadas en una tabla con los códigos correspondientes a cada token. Cuando el analizador lexicográfico reconoce un identificador, primero recorre la tabla y determina si el identificador se encuentra dentro de la tabla. Si está en la tabla devuelve (código,λ), sino controla la Tabla de Símbolos para establecer si el identificador apareció con anterioridad en la entrada. Si lo encuentra en la tabla de símbolos devuelve el par (código,valor), sino lo almacena en ella y retorna el mismo par. En este caso, el analizador lexicográfico es el que administra la Tabla de Símbolos.
2. *Si las palabras claves son* ***no reservadas****:*
   Implica que deben incorporarse al autómata transiciones que las reconozcan. Por ejemplo, en un lenguaje tipo FORTRAN, la siguiente sentencia:

   ```
   IF(i,j) = 5
   ```

   es una sentencia de asignación aritmética válida. Aunque también es válida la siguiente sentencia:

   ```
   IF(i) j = 5
   ```

   ya que es una sentencia condicional `IF`.
   De manera que `IF` puede tener al menos dos significados diferentes:
   - identificador
   - palabra clave

   ¿Cómo hace el analizador lexicográfico para determinar si debe retornar al analizador sintáctico el par (`id`,`IF`) o (`codigo_de_IF`,λ)?
   La manera de discriminar es especificando un contexto para la palabra clave `IF` y esto se hace teniendo en cuenta que la sentencia condicional `IF` tiene la forma:

   ```
   IF (condicion) sentencia
   ```

   el paréntesis derecho puede usarse para una indexación, agrupar operandos en una expresión aritmética, etc. en este caso debe ir seguido de un operador o del fin de sentencia y con toda seguridad no puede ir seguido de letra.
   Para determinar si `IF` es una palabra clave se debe mirar hacia adelante buscando

los símbolos que siguen después del ")" (sin perder los caracteres leídos, ¿por qué?). Una vez que se ha determinado cual es el significado para `IF`, se debe enviar el correspondiente código del token (si es identificador o palabra clave) al analizador sintáctico. Es claro que en este caso la determinación del significado de la secuencia de caracteres está guiado por la gramática subyacente pues el analizador debe incorporar conocimiento sintáctico para enviar el token correcto.

b) *Espacios en blanco, tabs, newlines y los identificadores*:

1. *Espacios en blanco, tabs, newlines* ***significativos***: en cuyo caso si hay blancos entre los tokens y en particular entre identificadores, entonces dichos blancos son tomados como separadores.
2. *Espacios en blanco, tabs, newline* ***no significativos***: no causa mayor problema pues siempre existe algún delimitador que separe a dos identificadores adyacentes. Por ejemplo:

   ```
   x := otro valor; y := x;
   ```

   El blanco existente en `otro valor` se descarta y el identificador queda delimitado por el símbolo de asignación y el punto y coma (`;`).

3. Detección de Errores Lexicográficos

Los errores que pueden ser detectados en la fase de análisis lexicográfico son muy escasos ya que su visión del problema es muy reducida. Al analizador sólo le interesa aislar lexemas que representen tokens bien formados y los errores que puede detectar son aquellos relacionado a tokens mal formados.

Si por ejemplo, en un programa C se encuentra la secuencia de caracteres `fi` en el siguiente contexto:

```
fi (a == f(x))....
```

un analizador lexicográfico no puede saber que si se trata de la palabra clave `if` o si es un identificador de función no declarado. El analizador lexicográfico debe retornar el token identificador ya que `fi` es un identificador válido, y el error será detectado y tratado por alguna otra fase del compilador.

Los errores que pueden ser detectados por el analizador son por ejemplo:

- Caracteres ilegales en el programa fuente.
- El analizador puede controlar el número de caracteres que aparecen en un identificador si el lenguaje impone alguna clase de restricción al respecto.
- En muchos lenguajes los comentarios pueden ser incluídos en cualquier parte del programa fuente y por lo general entre un par de delimitadores. Los comentarios son procesados y eliminados por el analizador pero con la precaución de controlar que no se omitan dichos delimitadores.
- Si un número decimal se define por la expresión regular

  ```
  (0|...|9)+.(0|...|9)+,
  ```

luego no se admiten números decimales finalizados en punto (.). Si después del punto no viene un dígito, el analizador puede optar por una política de indicar el error, descartar el punto y continuar con el análisis hasta encontrar el próximo token.

4. Un lenguaje para especificar analizadores lexicográficos

Existe una amplia variedad de herramientas que permiten la especificación de analizadores lexicográficos basadas en expresiones regulares. A continuación se describe una herramienta particular de Java™ escrita en Java™ llamada **JFLex**[*](Java Fast Lexical Analyzer Generator). La herramienta está basada en la implementación en lenguaje C para la familia de sistemas operativos UNIX™ llamada Lex.

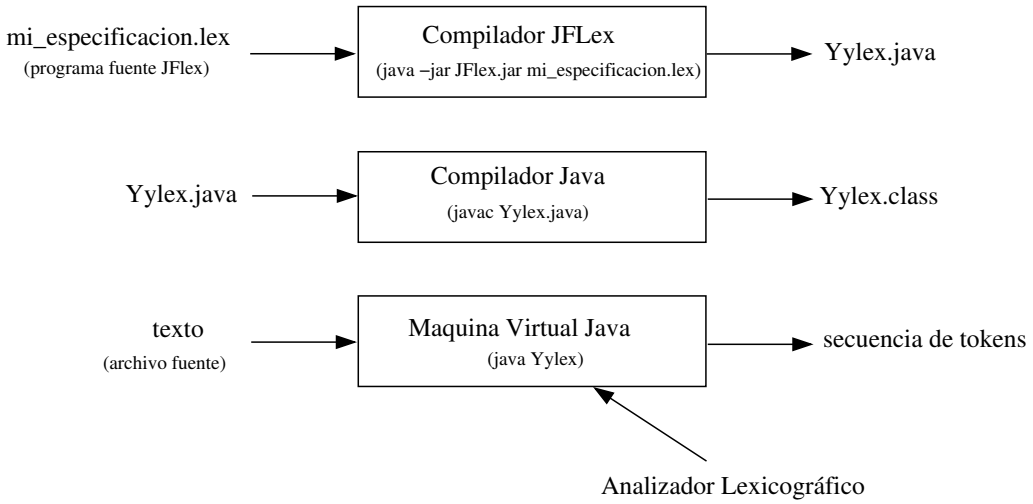


Figura 5: Utilización de la herramienta JFlex para generar un Analizador Lexicográfico

En la figura 5 se muestra el uso generalmente dado a la herramienta JFlex. Primero, se crea un programa `mi_especificacion.lex` utilizando el lenguaje JFlex con la especificación para un analizador lexicográfico. Se debe considerar que en la especificación sólo se detallan las acciones necesarias a realizar cuando se detecta un token, pero no existen detalles de las computaciones realizadas para detectarlo. Entonces, `mi_especificacion.lex` es procesada utilizando el compilador JFlex para producir una clase Java™ llamada `Yylex` (`Yylex.java`). La clase `Yylex` contiene entre otras variables y métodos, una representación tabular de un diagrama de transición, construido a partir de las expresiones regulares de `mi_especificacion.lex`, y un método estándar llamado **yylex()** que usa la tabla para reconocer lexemas. Las acciones asociadas con las expresiones regulares son trozos de código Java™ y transferidos directamente a `Yylex.java`. Finalmente, `Yylex.java` es procesado para obtener código compilado `Yylex.class` utilizando el compilador Java™. El código compilado es el analizador lexicográfico, que transforma la entrada (`texto.txt`) en una secuencia de tokens.

[*]Accesible para descarga pública en http://jflex.de/

4.1. Especificaciones JFlex

Una especificación JFlex consiste de tres partes o secciones:

- Código del Usuario: Fragmento de código que es copiado directamente en el archivo `Yylex.java`.
- Opciones y Declaraciones: En esta sección se especifican las *opciones* que personalizan la clase generada por el compilador JFlex, además de las *declaraciones de estados léxicos* y *definiciones de macros*. Una *definición de macro* permite especificar un identificador de macro que denotará a una expresión regular. Se debe considerar que no están permitidos los ciclos entre las definiciones. A continuación se muestra como se definen:

  $$\begin{array}{l} D_1 = P_1 \\ D_2 = P_2 \\ \quad\vdots \quad\quad \vdots \\ D_n = P_n \end{array}$$

  donde $D_j \neq D_i$ ($\forall \mathbf{i} \neq \mathbf{j}$) y P_i es un expresión regular construida a partir de $\Sigma \cup \{D_1, \ldots, D_{i-1}\}$.
- Reglas Léxicas: La sección de reglas léxicas contiene un *conjunto de expresiones regulares* y *acciones* (en código Java^{TM}) que se construye de la siguiente manera:

  $$\begin{array}{ll} R_1 & \{acción_1\} \\ R_2 & \{acción_2\} \\ \quad\vdots & \quad\vdots \\ R_m & \{acción_m\} \end{array}$$

  donde R_i está definida sobre $\Sigma \cup \{D_1, \ldots, D_n\}$ $(1 \leq i \leq m)$ y es una expresión regular llamada patrón y $\{acción_i\}$ es un fragmento de código en lenguaje Java^{TM} que se ejecuta cuando un token (secuencia de símbolos en la entrada) unifica con la expresión regular R_i. La acción puede ser vacía y equivale a decir que cuando el patrón es reconocido simplemente se lo ignora. Cada una de las expresiones regulares R_i $(1 \leq i \leq m)$ son tabuladas y el código de cada una de las $acción_i$ $(1 \leq i \leq m)$ es insertando en un **case** de una sentencia **switch** dentro de la función **yylex()**.
  Un posible pseudocódigo de la función miembro **yylex()** es el siguiente:

```
public Yytoken yylex() {
 ...
 switch (...) {
    ...
   case i: {acción_i}
    ...
  }
 ...
}
```

4.2. Funcionamiento del Analizador Lexicográfico

A continuación se analiza el funcionamiento del analizador creado utilizando la herramienta JFlex y unido a su relación con un analizador sintáctico.

- Cuando el analizador lexicográfico es activado (por ejemplo, por el analizador sintáctico):

  1. Comienza a leer la entrada de a un caracter a la vez, hasta que encuentra el prefijo más largo que unifica con una de las expresiones regulares R_i.
     - Si una secuencia de caracteres unifica con dos o más patrones $R_{i_1}, R_{i_2}, \ldots, R_{i_j}$, entonces escoge R_k donde $k = \text{mín}\{i_1, i_2, \ldots, i_j\}$.
     - Si alguna secuencia de caracteres en la entrada no unifica con ningún patrón R_i, esa secuencia es copiada en la salida estándar. Las reglas de traducción deberán ser lo suficientemente generales para que cada entrada posible sea reconocida y evitar que la secuencia sea copiada en la salida estándar.
  2. Lo almacena en un buffer accesible desde el método `yytext()`
  3. Ejecuta la acción *acción*$_i$.
  4. Si dentro de la *acción*$_i$ existe una sentencia `return` (un retorno explícito),
     - el analizador retorna el control a la función invocante (por ejemplo, el analizador sintáctico),
     - de otro modo, continua analizando la entrada (retorna al paso 1).

5. Un ejemplo de una especificación Lex

Si se desean reconocer los siguientes tokens:

expresión regular	**token**
`eb`[1]	
`if`	`if`
`then`	`then`
`id`[2]	`id`
`num`[3]	`num`
`<`	`op_rel_men`
`>`	`op_rel_may`

[1] `eb` corresponde con una expresión regular que denota el espacio en blanco, tabs. y newline (nueva línea).
[2] `id` corresponde con una expresión regular que denota cualquier secuencia de caracteres que comienza con una letra y es seguida de letras o digitos.
[3] `num` corresponde con una expresión regular que denota cualquier secuencia que comienza con uno o mas dígitos (0-9) le puede seguir un punto (.) y una secuencia de uno o más dígitos y puede ser seguida de una letra E, un signo (+ o -) y una secuencia de dígitos (número entero o decimal en notación científica).

La especificación JFlex en un archivo denominado por ejemplo `Ejemplo1.flex` podría ser la siguiente:

```
import java.io.*;
%%

%{
public static final int COD_OP_REL_MEN      = 1;
public static final int COD_OP_REL_MAY      = 2;
public static final int COD_IF              = 3;
public static final int COD_THEN            = 4;
public static final int COD_ID              = 5;
public static final int COD_NUM             = 6;

class Yytoken {
  public int codigo;
  public String valor;

  Yytoken (int c, String v) {
    codigo = c; valor = v;
  }

  public String imprimir() {
    /* Imprimir el token */
    System.out.println("("+codigo+","+valor+")");}
}
%}

%line
%char
%full

delim=[ \t\n]
eb={delim}+
letra=[A-Za-z]
digito=[0-9]
id={letra}({letra}|{digito})+
num={digito}+(\.{digito}+)?(E[+\-]?{digito}+)?

%%

{eb}  {/*ninguna accion y no retornar*/}

if    {return (new Yytoken(COD_IF,""));}

then  {return (new Yytoken(COD_THEN,""));}

{id}  {return (new Yytoken(COD_ID,yytext()));}

{num} {return (new Yytoken(COD_NUM,yytext()));}
```

```
"<"   {return (new Yytoken(COD_OP_REL_MEN,yytext()));}

">"   {return (new Yytoken(COD_OP_REL_MAY,yytext()));}

.     {/*Simbolo no reconocido*/}
```

Y un posible trozo de código incluido en un archivo denominado (`Ejemplo1.java`) que invoque al analizador `Yylex` puede ser el siguiente:

```
public class Ejemplo1 {
  public static void main(String argv[]) {
    try {
      Yylex s = new Yylex(new FileReader(argv[0]));
      Yylex.Yytoken t = s.yylex();
      while (t != null){t.imprimir();t = s.yylex();}
    }
      catch (Exception e) {}
  }
}
```

Considerando los dos archivos anteriores, los posibles pasos a seguir para obtener el analizador (según lo visto en la figura 5) son los siguientes:

- Compilar las especificaciones JFlex, invocando al compilador JFlex para obtener un archivo fuente en lenguaje Java™ llamado `Yylex.java`:

  ```
  java -jar JFlex.jar Ejemplo1.flex
  ```

- Compilar conjuntamente los código fuente `Yylex.java` y `Ejemplo1.java` para obtener el archivo compilado (`Ejemplo1.class`) que corresponde al analizador:

  ```
  javac Yylex.java Ejemplo1.java
  ```

- Para analizar un archivo fuente (`texto.txt`) se invoca al archivo compilado obtenido anteriormente junto con el archivo fuente a analizar y se obtendrá la impresión de los token por pantalla:

  ```
  java Ejemplo1 texto.txt
  ```

A continuación se realiza un explicación detallada de las especificaciones JFlex que se mostraron con anterioridad.

En la sección de opciones y declaraciones se encuentran encerradas por los símbolos %{ y %} la definición de variables estáticas y de la clase que almacena el token. Se debe recordar que cualquier cosa que aparezca entre estos símbolos es copiado directamente en el archivo de salida. Además de las opciones también se incluyen las definiciones de macros que consisten de un nombre y una expresión regular denotada por el nombre. Por ejemplo, la primer definición

es nombrada como `delim` y corresponde a la clase de caracteres `[ \t\n]`, la cual es una expresión regular que unifica con cualquiera de los tres símbolos: espacio en blanco, tab (`\t`) o newline (`\n`). La segunda definición denotada por el nombre `eb` corresponde con cualquier secuencia de uno o más símbolos de la macro nombrada con `delim`. La palabra `delim` debe ser encerrada entre llaves en JFlex, para distinguirla del patrón consistente de las cinco letras `d e l i m`.

En la definición de `letra` se observa nuevamente la utilización de la *clase de caracteres*. La abreviatura `[A-Za-z]`, donde `A-Z` significa cualquiera de las letras mayúsculas entre A y Z y `a-z` significa cualquier letras minúsculas entre a y z. En la definición de `id` se usan paréntesis, los cuales son metasímbolos en JFlex con su significado natural de agrupar operandos y operadores. La barra vertical (`|`) es otro de los metasímbolo de JFlex que representa unión.

En la definición de la expresión regular para `num` se observa la utilización del metasímbolo `?` que significa la ocurrencia de cero o una vez de la expresión regular precedente. También se observa que `\` es un metasímbolo de JFlex con su significado natural de preceder a un símbolo para quitarle su significado de metasímbolo. Por ejemplo, el punto decimal en la definición de `num` es expresado por `\.` debido a que el punto por sí mismo es un metasímbolo que representa la clase de caracteres de todos los caracteres excepto newline. También se puede observar que en `[+\-]`, se antepone el símbolo `\` al símbolo menos (`-`), debido a que el símbolo menos es un metasímbolo usado para denotar rangos `[A-Z]`. Otra manera de lograr que los caracteres tengan su significado natural, aún si ellos son metasímbolos, es encerrándolos entre comillas.

Ahora se consideran las reglas de léxicas en la sección siguiente a los segundos %%. La primera regla dice que si en la entrada existe una secuencia de caracteres que unifica con la expresión regular denotada por `eb` (cualquier secuencia de blancos, tabs o newline), no realiza ninguna acción y no retornar al invocante. En este punto se debe recordar que la estructura del analizador es tal que lo que él hace es reconocer tokens hasta que la acción asociada con uno sea el retorno explícito al invocante (`return`).

La segunda regla dice que si en la entrada existe una secuencia de caracteres que unifica con la expresión regular denotada por `if` (palabra clave `if`), significa que reconoció un token `if` y debe asignar el código `COD_IF` y un valor nulo al par (código, valor). Lo mismo sucede si reconoce el token asociado a la palabra clave `then`. Además, se puede observar que en las acciones que corresponden con las reglas asociada a identificadores (`id`) y a número (`num`) utilizan el método `yytext()` para almacenar en la variable `valor` de la clase `Yytoken` el correspondiente lexema.

La entrada al analizador obtenido a partir de las especificaciones anteriores puede ser la siguiente secuencia de caracteres:

```
\t\tif j < 36\n\t\t\tthen
```

Los dos `\t` (tabs) son el prefijo inicial de la entrada que unifica con el patrón `eb` y como la acción es no hacer nada, y además, no existe un retorno explícito, el analizador comienza a buscar otro token.

Continuando el análisis encuentra la próxima secuencia de caracteres `if` que unifica con el patrón `if` y con el patrón `id`. La solución al conflicto se determina a partir de la posición en la cual se encuentran las reglas. De manera que el patrón para la palabra clave `if` precede al patrón para `id` y finalmente el token resultante es el que corresponde con la palabra clave `if` realizando la respectiva acción. Generalmente se utiliza la estrategia de listar las palabras claves antes que el patrón para identificadores.

Luego de realizar la acción para el patrón `if` retorna al invocante y cuando el analizador sea invocado nuevamente continuará con el análisis de la cadena de entrada a partir de la última secuencia encontrada.

6. Expresiones Regulares

El metalenguaje para especificar una expresión regular en este contexto es más completo de lo que a continuación se detalla, sin embargo será suficiente para nuestro propósito (ver el manual de JFlex incluido dentro del paquete).

Una secuencia de uno o más caracteres es una expresión regular, dicha secuencia puede también ir encerrada entre comillas dobles. Por ejemplo:

```
a  aa "a"  "xxx" "**++"
```

Los siguientes son símbolos del metalenguaje llamados caracteres especiales. Los mismos tendrán un determinado efecto excepto que estén encerrados entre `[]` o `""`, o bien precedidos de la barra invertida, en donde pierden su significado especial.

```
'.' , '*' , '+' , '?' , '|' , '[' , '\' , ']' , '^' , '$' , '{' , '}'
```

6.1. Símbolos del Metalenguaje

6.1.1. El punto (.)

Es un metasímbolo que concuerda con todos los caracteres, menos con `'\n'`.

6.1.2. Clases de caracteres

Una cadena de caracteres no vacía entre corchetes es una expresión regular de un caracter que concuerda con cualquier caracter en esa cadena por ejemplo:

```
[0123456789]
```

concuerda con sólo uno de los números del intervalo 0 a 9.

Otra forma de expresar lo anterior, es introduciendo el signo `'-'` a la manera de un subrango:

```
[0-9]
```

En el caso de que deseemos reconocer el signo `'-'` dentro de los corchetes, el mismo deberá ser el primer caracter. Así la siguiente expresión denota todos los números de 0 al 9, más el signo `'-'`:

```
[-0-9]
```

6.1.3. El asterisco (*)

La expresión regular seguida de un asterisco denota cero o más ocurrencias de dicha expresión regular. Por ejemplo:

`a*` denota {λ, a, aa, aaa,...}

`[A-Za-z][A-Za-z_0-9]*` podría denotar un identificador en un lenguaje de programación.

6.1.4. El signo +

La expresión regular seguida de un `+`, denota una o más ocurrencias de dicha expresión regular. Por ejemplo:

`n+` denota `{n , nn, nnn, nnnn,...}`

6.1.5. El signo de interrogación (?)

Una expresión regular seguida por `?` denota 0 o 1 ocurrencia de la expresión regular. Por ejemplo:

`aa?b?` denota `{ aab, aa, ab, a}`

6.1.6. La Unión (|)

Una expresión regular separada por `'|'` concuerda con cadenas de caracteres denotada por cualquiera de las expresiones regulares. Por ejemplo:

`aaa | bbb | ccc` denota `{aaa, bbb, ccc}`

6.1.7. Las llaves ({ })

Cuando asignamos un nombre a una expresión regular es necesario encerrarla entre llaves para expandirla y que sea reemplazada por la expresión regular que éste denota. Por ejemplo, si se define la expresión regular llamada **pal** de la siguiente manera:

pal `[a-zA-Z]+`.

Luego se puede utilizar

{pal}

que es equivalente a `[a-zA-Z]+`

6.2. Precedencia de los operadores

La precedencia de los operadores en orden de mayor a menor es la siguiente:

1. `[]`
2. `*?+`
3. concatenación, que si bien no se mencionó, es simplemente poner una expresión regular seguida de la otra para formar una nueva.
4. `|`

De manera tal que si deseamos alterarla se hace uso de los paréntesis. Por ejemplo en el siguiente caso:

`abc+` denota `{abc, abcc, abccc,...}`

mientras que con paréntesis

`(abc)+` denota `{abc, abcabc, abcabcabc,...}`

aquí hace primero la concatenación y luego aplica el operador `+`

6.3. La barra invertida

Anula el efecto de un caracter especial. Por ejemplo, si se desea construir una expresión regular que denote una secuencia de uno o más asteriscos se deberá escribir:

```
\*+
```

6.4. Otros ejemplos

`\..*`	denota	una secuencia de caracteres, que no incluye `'\n'`, la cual comienza con un punto `'.'` (uso de la barra invertida).
`a(bb|cc)*d`	denota	`{ad, abbd, accd, abbccd, accbbd, ...}`
`x[*?+.]y`	denota	`{x*y , x+y , x?y , x.y}` Aquí el efecto de los caracteres especiales está anulado por estar dentro de `[]`
`[-+]?[0-9]+`	denota	una constante entera con signo.

Otros símbolos especiales son: `'^'` y `'$'`, los cuales indican que una expresión regular debe ser encontrada al principio y al final de la línea respectivamente. Algunos ejemplo son:

`^[0-9]+`	Constante entera al principio de línea.
`^\\\\`	Doble barra invertida al principio de la línea (comentario en C++ o Java).
`A$`	Una `'a'` al final de la línea
`^a$`	Una línea que contiene sólo una `'a'`