4º Ingeniería Informática
II26 Procesadores de lenguaje
Analizador léxico

Esquema del tema

1. Introducción
2. Repaso de conceptos de lenguajes formales
3. Categorías léxicas
4. Especificación de las categorías léxicas
5. Autómatas de estados finitos
6. Implementación del analizador léxico
7. Introducción a un generador automático de analizadores léxicos: flex
8. Algunas aplicaciones de los analizadores léxicos
9. Resumen del tema

1. Introducción

Vimos que la primera fase del análisis es el análisis léxico. El principal objetivo del analizador léxico es leer el flujo de caracteres de entrada y transformarlo en una secuencia de componentes léxicos que utilizará el analizador sintáctico.

Al tiempo que realiza esta función, el analizador léxico se ocupa de ciertas labores de "limpieza". Entre ellas está eliminar los blancos o los comentarios. También se ocupa de los problemas que pueden surgir por los distintos juegos de caracteres o si el lenguaje no distingue mayúsculas y minúsculas.

Para reducir la complejidad, los posibles símbolos se agrupan en lo que llamaremos categorías léxicas. Tendremos que especificar qué elementos componen estas categorías, para lo que emplearemos expresiones regulares. También será necesario determinar si una cadena pertenece o no a una categoría, lo que se puede hacer eficientemente mediante autómatas de estados finitos.

2. Repaso de conceptos de lenguajes formales

2.1. Por qué utilizamos lenguajes formales

Como acabamos de comentar, para transformar la secuencia de caracteres de entrada en una secuencia de componentes léxicos utilizamos autómatas de estados finitos. Sin embargo, estos autómatas los especificaremos utilizando expresiones regulares. Tanto unos como otras son ejemplos de utilización de la teoría de lenguajes formales. Es natural preguntarse si es necesario dar este rodeo. Existen varias razones que aconsejan hacerlo.

La primera razón para emplear herramientas formales es que nos permiten expresarnos con precisión y, generalmente, de forma breve. Por ejemplo, para describir la categoría de los enteros, podemos intentar utilizar el castellano y decir algo así como que son "secuencias de dígitos". Pero entonces no queda claro cuales son esos dígitos (por ejemplo, en octal, los dígitos van del cero al siete). Cambiamos entonces a "secuencias de dígitos, cada uno de los cuales puede ser un `0`, un `1`, un `2`, un `3`, un `4`, un `5`, un `6`, un `7`, un `8` o un `9`". Todavía queda otro problema más, ¿valen las cadenas vacías? Normalmente no, así que llegamos a "un número entero consiste en una secuencia de uno o más dígitos, cada uno de los cuales puede ser un `0`, un `1`, un `2`, un `3`, un `4`, un `5`, un `6`, un `7`, un `8` o un `9`". Sin embargo, con expresiones regulares, podemos decir lo mismo con $[0–9]^+$.

Otra ventaja de las herramientas formales, es que nos permiten razonar sobre la corrección de nuestros diseños y permiten conocer los límites de lo que podemos hacer. Por ejemplo, el lema de bombeo nos permite saber que no podremos utilizar expresiones regulares para modelar componentes léxicos que tengan el mismo número de paréntesis abiertos que cerrados, por lo que averiguar si algo está bien parentizado será tarea del analizador sintáctico.

Como última ventaja del empleo de lenguajes formales, comentaremos la existencia de herramientas para automatizar la implementación. Por ejemplo, el paso de las expresiones regulares a un programa que las reconozca se puede hacer mediante un generador de analizadores léxicos como `flex`.

2.2. Alfabetos y lenguajes

Al trabajar con lenguajes formales, utilizaremos una serie de conceptos básicos. En primer lugar, un *alfabeto* Σ es un conjunto finito de símbolos. No nos interesa la naturaleza de los símbolos. Dependiendo de la aplicación que tengamos en mente, éstos pueden ser: caracteres, como al especificar el analizador léxico; letras o palabras, si queremos trabajar con lenguaje natural; categorías léxicas, al especificar el analizador sintáctico; direcciones en el plano, al hacer OCR, etc. Justamente esta abstracción es lo que hace que los lenguajes formales se puedan aplicar ampliamente.

Una *cadena* es una secuencia finita de símbolos del alfabeto. Nuevamente, no estamos interesados en la naturaleza precisa de las cadenas. Si estamos especificando el analizador léxico, podemos ver la entrada como una cadena; si trabajamos con lenguaje natural, la cadena puede ser una pregunta a una base de datos; para el analizador sintáctico, una cadena es el programa una vez pasado por el analizador léxico; en OCR, una cadena es la descripción de una letra manuscrita, etc.

La cadena de longitud cero se denomina *cadena vacía* y se denota con λ. Para referirnos al conjunto de cadenas de longitud k, utilizamos Σ^k. Si nos referimos al conjunto de todas las cadenas que se pueden escribir con símbolos del alfabeto, usamos Σ^*. Se cumplirá que:

$$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

Date cuenta de que Σ^* es un conjunto infinito, pero que cualquiera de sus cadenas es finita.

Finalmente, un *lenguaje formal* es un subconjunto de Σ^*. Tal como está definido, puede ser finito o infinito. Es más, la definición no impone la necesidad de que el lenguaje tenga algún sentido o significado; un lenguaje formal es simplemente un conjunto de cadenas.

Una vez hemos decidido que vamos a utilizar un lenguaje formal, nos enfrentaremos a dos problemas: especificarlo y reconocerlo. En nuestro caso, las especificaciones serán de dos tipos: por un lado expresiones regulares para los lenguajes que emplearemos en el análisis léxico y por otro gramáticas incontextuales en el análisis sintáctico.

Una vez especificado el lenguaje será necesario encontrar la manera de reconocerlos, esto es, resolver la pregunta si una cadena dada pertenece al lenguaje. Veremos que los métodos de análisis que emplearemos serán capaces de responder a la pregunta de forma eficiente: con un coste lineal con la talla de la cadena.

3. Categorías léxicas

3.1. Conceptos básicos

Para que el analizador léxico consiga el objetivo de dividir la entrada en partes, tiene que poder decidir por cada una de esas partes si es un componente separado y, en su caso, de qué tipo.

De forma natural, surge el concepto de *categoría léxica*, que es un tipo de símbolo elemental del lenguaje de programación. Por ejemplo: identificadores, palabras clave, números enteros, etc.

Los *componentes léxicos* (en inglés, *tokens*) son los elementos de las categorías léxicas. Por ejemplo, en C, `i` es un componente léxico de la categoría **identificador**, `232` es un componente léxico de la categoría **entero**, etc. El analizador léxico irá leyendo de la entrada y dividiéndola en componentes léxicos.

Para algunos componentes, fases posteriores del análisis necesitan información adicional a la categoría a la que pertenece. Por ejemplo, el valor de un entero o el nombre del identificador. Utilizamos los *atributos* de los componentes para guardar esta información.

Un último concepto que nos será útil es el de *lexema*: la secuencia concreta de caracteres que corresponde a un componente léxico.

Por ejemplo, en la sentencia `altura=2;` hay cuatro componentes léxicos, cada uno de ellos de una categoría léxica distinta:

Categoría léxica	Lexema	Atributos
identificador	`altura`	—
asignación	`=`	—
entero	2	valor: 2
terminador	;	—

3.2. Categorías léxicas más usuales

Algunas familias de categorías léxicas típicas de los lenguajes de programación son:

Palabras clave Palabras con un significado concreto en el lenguaje. Ejemplos de palabras clave en C son `while`, `if`, `return`... Cada palabra clave suele corresponder a una categoría léxica. Habitualmente, las palabras clave son reservadas. Si no lo son, el analizador léxico necesitará información del sintáctico para resolver la ambigüedad.

Identificadores Nombres de variables, nombres de función, nombres de tipos definidos por el usuario, etc. Ejemplos de identificadores en C son `i`, `x10`, `valor_leido`...

Operadores Símbolos que especifican operaciones aritméticas, lógicas, de cadena, etc. Ejemplos de operadores en C son `+`, `*`, `/`, `%`, `==`, `!=`, `&&`...

Constantes numéricas Literales[1] que especifican valores numéricos enteros (en base decimal, octal, hexadecimal...), en coma flotante, etc. Ejemplos de constantes numéricas en C son `928`, `0xF6A5`, `83.3E+2`...

Constantes de carácter o de cadena Literales que especifican caracteres o cadenas de caracteres. Un ejemplo de literal de cadena en C es `"una cadena"`; ejemplos de literal de carácter son `'x'`, `'\0'`...

Símbolos especiales Separadores, delimitadores, terminadores, etc. Ejemplos de estos símbolos en C son `{`, `}`, `;`... Suelen pertenecer cada uno a una categoría léxica separada.

Hay tres categorías léxicas que son especiales:

Comentarios Información destinada al lector del programa. El analizador léxico los elimina.

Blancos En los denominados "lenguajes de formato libre" (C, Pascal, Lisp, etc.) los espacios en blanco, tabuladores y saltos de línea sólo sirven para separar componentes léxicos. En ese caso, el analizador léxico se limita a suprimirlos. En otros lenguajes, como Python, no se pueden eliminar totalmente.

Fin de entrada Se trata de una categoría ficticia emitida por el analizador léxico para indicar que no queda ningún componente pendiente en la entrada.

[1] Los literales son secuencias de caracteres que representan valores constantes.

© Universitat Jaume I 2008-2009

4. Especificación de las categorías léxicas

El conjunto de lexemas que forman los componentes léxicos que podemos clasificar en una determinada categoría léxica se expresa mediante un *patrón*. Algunos ejemplos son:

Categoría léxica	Lexemas	Patrón
entero	`12 5 0 192831`	Secuencia de uno o más dígitos.
identificador	`x x0 area`	Letra seguida opcionalmente de letras y/o dígitos.
operador	`+ -`	Caracteres `+`, `-`, `*` o `/`.
asignación	`:=`	Carácter `:` seguido de `=`.
while	`while`	La palabra `while`.

Ya hemos comentado las dificultades de especificar correctamente las categorías mediante lenguaje natural, así que emplearemos expresiones regulares.

4.1. Expresiones regulares

Antes de describir las expresiones regulares, recordaremos dos operaciones sobre lenguajes. La *concatenación* de dos lenguajes L y M es el conjunto de cadenas que se forman al tomar una del primero, otra del segundo y concatenarlas. Más formalmente: la *concatenación* de los lenguajes L y M es el lenguaje $LM = \{xy \mid x \in L \wedge y \in M\}$. La notación L^k se utiliza para representar la concatenación de L consigo mismo $k-1$ veces, con los convenios $L^0 = \{\lambda\}$ y $L^1 = L$.

La otra operación que definiremos es la *clausura* de un lenguaje L, representada como L^* y que es el conjunto de las cadenas que se pueden obtener mediante la concatenación de un número arbitrario de cadenas de L. Más formalmente $L^* = \bigcup_{i=0}^{\infty} L^i$ (recuerda que los lenguajes son conjuntos y por tanto tiene sentido hablar de su unión).

Ejercicio 1

Sean $L = \{\mathtt{a}, \mathtt{aa}, \mathtt{b}\}$ y $M = \{\mathtt{ab}, \mathtt{b}\}$. Describe LM y M^3 por enumeración.

4.2. Expresiones regulares básicas

Definiremos las expresiones regulares de manera constructiva. Dada una expresión regular r, utilizaremos $L(r)$ para referirnos al lenguaje que representa. Como base, emplearemos las expresiones para el lenguaje vacío, la cadena vacía y las cadenas de un símbolo:

- La expresión regular $\emptyset$ denota el lenguaje vacío: $L(\emptyset) = \emptyset$.
- La expresión regular λ denota el lenguaje que únicamente contiene la cadena vacía[2] $L(\lambda) = \{\lambda\}$.
- La expresión regular $\mathtt{a}$, donde $\mathtt{a} \in \Sigma$, representa el lenguaje $L(\mathtt{a}) = \{\mathtt{a}\}$.

La verdadera utilidad de las expresiones regulares surge cuando las combinamos entre sí. Para ello disponemos de los operadores de clausura, unión y concatenación:

- Sea r una expresión regular, la expresión regular $(r)^*$ representa el lenguaje $L((r)^*) = (L(r))^*$.
- Sean r y s dos expresiones regulares. Entonces:
  - La expresión regular (rs) representa el lenguaje $L((rs)) = L(r)L(s)$.
  - La expresión regular $(r|s)$ representa el lenguaje $L((r|s)) = L(r) \cup L(s)$.

[2]...y que no tiene nada que ver con el lenguaje vacío, como ya sabrás.

A partir de estas definiciones, vamos a ver qué significa la expresión regular $((\mathtt{a}|\mathtt{b})^*)(\mathtt{ab})$. En primer lugar, vemos que es la concatenación de otras dos expresiones: $(\mathtt{a}|\mathtt{b})^*$ y `ab`. La primera designa el lenguaje que es la clausura de la disyunción de los lenguajes formados por una `a` y una `b`, respectivamente. Podemos expresarlo más claramente como las cadenas formadas por cualquier número de aes y bes. La segunda expresión corresponde al lenguaje que únicamente contiene la cadena `ab`. Al concatenar ambas, obtenemos cadenas formadas por un número arbitrario de aes y bes seguidas de `ab`. Dicho de otra forma, son cadenas de aes y bes que terminan en `ab`.

Para describir los literales enteros, podríamos utilizar la expresión regular:

$$((((((((((\mathtt{0}|\mathtt{1})|\mathtt{2})|\mathtt{3})|\mathtt{4})|\mathtt{5})|\mathtt{6})|\mathtt{7})|\mathtt{8})|\mathtt{9})((((((((((\mathtt{0}|\mathtt{1})|\mathtt{2})|\mathtt{3})|\mathtt{4})|\mathtt{5})|\mathtt{6})|\mathtt{7})|\mathtt{8})|\mathtt{9}))^*)$$

El exceso de paréntesis hace que esta expresión sea bastante difícil de leer. Podemos simplificar la escritura de expresiones regulares si establecemos prioridades y asociatividades. Vamos a hacer que la disyunción tenga la mínima prioridad, en segundo lugar estará la concatenación y finalmente la clausura tendrá la máxima prioridad. Además, disyunción y concatenación serán asociativas por la izquierda. Cuando queramos que el orden sea distinto al marcado por las prioridades, utilizaremos paréntesis. Con este convenio, podemos representar los enteros así:

$$(\mathtt{0}|\mathtt{1}|\mathtt{2}|\mathtt{3}|\mathtt{4}|\mathtt{5}|\mathtt{6}|\mathtt{7}|\mathtt{8}|\mathtt{9})(\mathtt{0}|\mathtt{1}|\mathtt{2}|\mathtt{3}|\mathtt{4}|\mathtt{5}|\mathtt{6}|\mathtt{7}|\mathtt{8}|\mathtt{9})^*$$

4.3. Expresiones regulares extendidas

Aun cuando las expresiones regulares resultan más legibles al introducir las prioridades, sigue habiendo construcciones habituales que resultan incómodas de escribir, por ejemplo la expresión que hemos utilizado para los enteros. Veremos en este apartado algunas extensiones que, sin aumentar el poder descriptivo de las expresiones regulares, permiten escribirlas más cómodamente.

Un aviso: así como las expresiones regulares básicas pueden considerarse una notación estándar, las extensiones que vamos a presentar ahora no lo son. Esto quiere decir que, según el programa que se emplee, algunas se considerarán válidas y otras no; también puede suceder que haya otras extensiones disponibles. También hay que advertir que, mientras no digamos otra cosa, trataremos con el conjunto de caracteres ASCII imprimibles (␣ representa el espacio en blanco):

```
␣  !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
@  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
`  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

y los caracteres salto de línea `\n` y tabulador `\t`. También utilizaremos eof para referirnos al fin de entrada (seguiremos la tradición de llamarlo fin de fichero —end of file— aunque en ocasiones no se corresponda con el fin de ningún fichero).

Agrupaciones de caracteres La primera extensión que vamos a ver nos permite escribir agrupaciones de caracteres de manera cómoda. En su forma más simple, si queremos representar la elección de uno entre varios caracteres, basta con encerrarlos entre corchetes. Así [`abd`] representa lo mismo que `a`|`b`|`d`. Cuando los caracteres son contiguos, se puede utilizar un guion entre el primero y el último para indicar todo el rango. De esta manera, [`a`–`z`] es el conjunto de las letras minúsculas. También podemos combinar varios rangos y conjuntos, de modo que [`a`–`zA`–`Z0`–`9_`] es el conjunto de las letras, los dígitos y el subrayado.

Si necesitamos que el propio guion esté en el rango, debemos ponerlo bien al principio bien al final. Así, los operadores aritméticos son [`-+*/`]. El ejemplo anterior también nos muestra que, dentro de un rango, los caracteres especiales pierden su significado. Por otro lado, si lo que queremos incluir es el corchete cerrado, debemos ponerlo en la primera posición, como en [`]1`–`6`].

© Universitat Jaume I 2008-2009

Otra posibilidad que nos ofrecen los corchetes es especificar los caracteres que no están en un conjunto. Para ello, se incluye en la primera posición un circunflejo, así [^0–9] es el conjunto de los caracteres que no son dígitos. En cualquier otra posición, el circunflejo se representa a sí mismo.

Finalmente, el punto (.) representa cualquier carácter, incluyendo el fin de línea. Otra advertencia más: según del sistema, el punto puede incluir o no el fin de línea. En particular, la librería `re` de Python considera el punto equivalente a [^\n], salvo que se empleen algunas opciones especiales. Para `flex`, el punto tampoco incluye el fin de línea.

Nuevos operadores Enriqueceremos el conjunto básico de operadores regulares con dos nuevos operadores: la *clausura positiva* y la *opcionalidad*.

El primero se representa mediante el signo más e indica una o más repeticiones de la expresión regular a la que afecta. Le otorgamos la misma prioridad que a la clausura normal. De manera simbólica, decimos que r^+ equivale a rr^*, donde r es una expresión regular arbitraria.

El segundo operador, representado mediante un interrogante, representa la aparición opcional de la expresión a la que afecta. De esta manera, $r?$ es equivalente a $r|\lambda$. Su prioridad es también la de la clausura.

Carácter de escape Hemos visto que hay una serie de caracteres que tienen un significado especial. A estos caracteres se les llama metacaracteres. Si queremos referirnos a ellos podemos utilizar el carácter de escape, en nuestro caso la barra \. Así, la expresión 4* representa una secuencia de cero o más cuatros mientras que 4* representa un cuatro seguido por un asterisco. Para representar la propia barra utilizaremos \\. Sin embargo, dentro de las clases de caracteres, no es necesario escapar, así 4[*] y 4* son equivalentes (y muy distintas de [4*]). La excepción son los caracteres fin de línea, tabulador y la propia barra: [\n\t\\].

Nombres La última extensión que permitiremos es la posibilidad de introducir nombres para las expresiones. Estos nombres luego podrán utilizarse en lugar de cualquier expresión. La única restricción que impondremos es que no deben provocar ninguna recursividad, ni directa ni indirectamente.

Ejemplos Vamos a ver algunos ejemplos de expresiones regulares. En primer lugar, revisitaremos los números enteros. Si utilizamos clases de caracteres, podemos escribirlos como [0–9][0–9]*. Dándonos cuenta de que esto representa una clausura positiva, podemos escribirlo como [0–9]+. Otra posibilidad es darle el nombre **dígito** a [0–9] y escribir los enteros como **dígito**+.

Supongamos que tenemos un lenguaje en el que los comentarios comienzan por << y terminan por >>, sin que pueda aparecer ninguna secuencia de dos mayores en el cuerpo del comentario. ¿Cómo podemos escribir la expresión regular en este caso? Un primer intento sería <<.*>>. Esta expresión tiene un problema ya que aceptará `<< a >> a >>` como comentario. Podemos resolver el problema si observamos que entre la apertura y el cierre del comentario puede aparecer cualquier carácter que no sea un mayor o, si aparece un mayor, debe aparecer seguido de algo que no lo sea. Es decir, que el interior del comentario es una secuencia de elementos de [^>]|>[^>]. Para lograr las repeticiones, utilizamos la clausura, con lo que, una vez añadidos el comienzo y final, obtenemos la expresión <<([^>]|>[^>])*>>. Como refinamiento, podemos utilizar el operador de opcionalidad para obtener <<(>?[^>])*>>.

Ejercicio 2

¿Por qué no vale la expresión regular <<[^(>>)]*>> en el caso anterior?

Ejercicio 3

La expresión regular que hemos visto antes acepta `007` y similares como literales enteros. Escribe una expresión regular para los enteros de modo que se admita o el número cero escrito como un único dígito o una secuencia de dígitos comenzada por un número distinto de cero.

Ejercicio 4

Escribe una expresión regular para el conjunto de las palabras reservadas `integer`, `real` y `char` escritas en minúsculas y otra que permita escribirlas con cualquier combinación de mayúsculas y minúsculas.

Ejercicio 5

En muchos lenguajes de programación, los identificadores se definen como secuencias de letras, dígitos y subrayados que no empiecen por un dígito. Escribe la expresión regular correspondiente.

Ejercicio 6

Escribe una expresión regular para las cadenas de dos o más letras minúsculas que empiezan por `a` o por `b` tales que la última letra coincide con la primera.

Ejercicio 7

Las siguientes expresiones regulares son intentos fallidos de modelar los comentarios en C. Da un contraejemplo para cada una de ellas:

- `/*.**/`
- `/*[^*/]**/`
- `/*([^*]|*[^/])**/`

Ejercicio* 8

Intenta demostrar que las siguientes expresiones para los comentarios en C son correctas, sin mirar el ejercicio 21.

- `/*([^*]**+[^*/])*[^*]**+/`
- `/*(**[^*/]|/)**+/`

Ejercicio 9

Escribe una expresión regular para los comentarios en C++. Estos tienen dos formas: una es la misma que en C y la otra abarca desde la secuencia `//` hasta el final de la línea.

Ejercicio 10

Diseña expresiones regulares para los siguientes lenguajes regulares:

a) Secuencias de caracteres encerradas entre llaves que no incluyen ni el carácter `|` ni la llave cerrada.

b) Secuencias de caracteres encerradas entre llaves en las que el carácter `|` sólo puede aparecer si va precedido por la barra invertida; la llave cerrada no puede aparecer y la barra invertida sólo puede aparecer si precede una barra vertical u otra barra invertida.

c) Números enteros entre 0 y 255 sin ceros iniciales.

© Universitat Jaume I 2008-2009

Ejercicio 11

¿Qué lenguajes representan las siguientes expresiones regulares?

- `0(0|1)*0`
- `(0|1)*0(0|1)(0|1)`
- `0*10*10*10*`
- `(00|11)*((01|10)(00|11)*(01|10)(00|11)*)*`

Ejercicio* 12

Las expresiones regulares también son cadenas de un lenguaje. Dado que tienen que estar bien parentizadas, este lenguaje no es regular. Sin embargo partes de él sí lo son, por ejemplo, las clases de caracteres.

Diseña una expresión regular que exprese el conjunto de clases de caracteres que podemos utilizar en una expresión regular.

5. Autómatas de estados finitos

Las expresiones regulares permiten describir con cierta comodidad los lenguajes regulares. Sin embargo, no es fácil decidir a partir de una expresión regular si una cadena pertenece al correspondiente lenguaje. Para abordar este problema utilizamos los autómatas de estados finitos. Estos son máquinas formales que consisten en un conjunto de estados y una serie de transiciones entre ellos. Para analizar una frase, el autómata se sitúa en un estado especial, el estado inicial. Después va cambiando su estado a medida que consume símbolos de la entrada hasta que esta se agota o no se puede cambiar de estado con el símbolo consumido. Si el estado que alcanza es un estado final, decimos que la cadena es aceptada; en caso contrario, es rechazada.

Distinguimos dos tipos de autómatas según cómo sean sus transiciones. Si desde cualquier estado hay como mucho una transición por símbolo, el autómata es determinista. En caso de que haya algún estado tal que con un símbolo pueda transitar a más de un estado, el autómata es no determinista. Nosotros trabajaremos únicamente con autómatas deterministas.

5.1. Autómatas finitos deterministas

Un *autómata finito determinista* es una quíntupla (Q, Σ, E, q_0, F) donde:

- Q es un conjunto finito de *estados*.
- Σ es un alfabeto.
- $E \subseteq Q \times \Sigma \times Q$ es un conjunto de *arcos* tal que si (p, a, q) y (p, a, q') pertenecen a E, se tiene que $q = q'$(condición de determinismo).
- $q_0 \in Q$ es el *estado inicial*.
- $F \subseteq Q$ es el conjunto de estados finales.

Para definir el funcionamiento del AFD debemos empezar por el concepto de *camino*. Un camino en un AFD es una secuencia $p_1, s_1, p_2, s_2, \dots, s_{n-1}, p_n$ tal que para todo i entre 1 y $n-1$ $(p_i, s_i, p_{i+1}) \in E$. Esto quiere decir que podemos ver el camino como una secuencia de arcos en la que el estado al que llega un arco es el de partida del siguiente. Decimos que el camino parte de p_1 y llega a p_n. Llamamos entrada del camino a la cadena $s_1 \dots s_{n-1}$, que será vacía si $n = 1$.

Una cadena x es *aceptada* por un AFD si existe un camino que parta de q_0, tenga como entrada x y llegue a un estado $q \in F$. El lenguaje *aceptado* o *reconocido* por un AFD es el conjunto de las cadenas aceptadas por él.

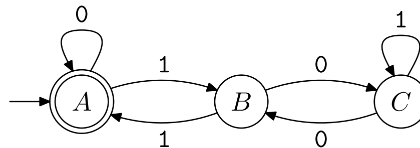
Los autómatas se pueden representar gráficamente. Para ello se emplea un grafo en el que cada estado está representado por un nodo y cada arco del autómata se representa mediante un arco

en el grafo. Los estados finales se marcan mediante un doble círculo y el estado inicial mediante una flecha.

Por ejemplo, el AFD $\mathcal{A} = (\{A, B, C\}, \{\texttt{0}, \texttt{1}\}, E, A, \{A\})$, con

$$E = \{(A, \texttt{0}, A), (A, \texttt{1}, B), (B, \texttt{0}, C), (B, \texttt{1}, A), (C, \texttt{0}, B), (C, \texttt{1}, C)\}, \tag{1}$$

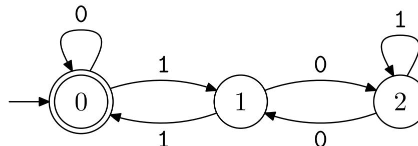
se puede representar gráficamente así:



Es interesante plantearse qué lenguaje reconoce este autómata. Podemos hacer un listado de algunas de las cadenas que acepta:

Longitud	Cadenas
0	λ
1	0
2	00, 11
3	000, 011, 110
4	0000, 0011, 0110, 1001, 1100, 1111

Si interpretamos la cadena vacía como cero y las restantes como números escritos en binario, podemos ver que todas las cadenas representan múltiplos de tres. Para convencernos de que no es casualidad, podemos reescribir los estados como 0, 1 y 2:



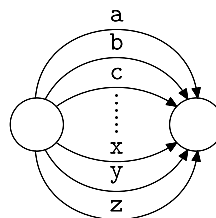
Ahora, puedes demostrar por inducción que si una cadena x lleva al estado i, el resto de dividir el número representado por x entre tres es justamente i. Como aceptamos sólo aquellas cadenas que terminan en el estado cero, una cadena es aceptada si y solo sí representa un número que dividido por tres da de resto cero, que es lo que queríamos.

EJERCICIO* 13

Demuestra que para cualquier n, el lenguaje de los múltiplos de n en base 2 es regular.

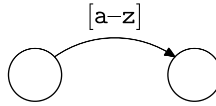
Ídem para base 10.

Cuando representemos autómatas, habrá muchas veces en las que encontremos estructuras como la siguiente:

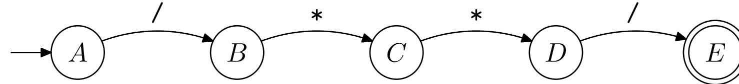


© Universitat Jaume I 2008-2009

Este tipo de estructuras las simplificaremos mediante agrupaciones de caracteres[3]. Así la estuctura anterior quedaría:

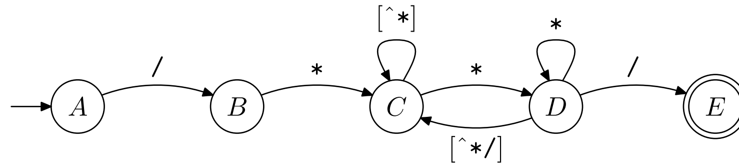


Vamos a intentar ahora escribir un autómata para los comentarios en C. En primer lugar podemos crear la parte del comienzo y el final del comentario:



Ahora nos queda decir que entre estas dos partes "cabe" cualquier cosa. Tendremos que poner un arco desde C a sí mismo con los símbolos que no son el asterisco. Observemos ahora qué pasa si tras ver un asterisco en C vemos un carácter que no es una barra (que terminaría el comentario). Hay que distinguir dos casos: si el nuevo carácter es también un asterisco, debemos permanecer en D; si el nuevo carácter no es un asterisco ni una barra, debemos retroceder al estado C.

Teniendo todo esto en cuenta, el autómata nos queda así:



Ejercicio 14

Diseña un autómata para los comentarios en C++.

5.1.1. Reconocimiento con AFDs

Como puedes imaginar, el algoritmo de reconocimiento con AFDs es bastante sencillo. Simplemente necesitamos llevar la cuenta del estado en que estamos y ver cómo podemos movernos:

Algoritmo ReconoceAFD $(x : \Sigma^*, \mathcal{A} = (Q, \Sigma, E, q_0, F) : AFD)$;
  $q := q_0$;
  $i := 1$;
  mientras $i \leq |x|$ **y** existe $(q, x_i, q') \in E$ **hacer**
    $q := q'$;
    $i := i + 1$;
  fin mientras
  si $i > |x|$ **y** $q \in F$ **entonces**
    devolver sí;
  si no
    devolver no;
  fin si
Fin ReconoceAFD.

Como puedes comprobar fácilmente, el coste de reconocer la cadena x es $O(|x|)$, que incluso ¡es independiente del tamaño del autómata!

Ejercicio* 15

Obviamente, la independencia del coste del tamaño del autómata depende de su implementación. Piensa en posibles implementaciones del autómata y cómo influyen en el coste.

[3] Date cuenta de que estamos utilizando un método para simplificar el dibujo, pero no podemos escribir en los arcos expresiones regulares cualesquiera.

5.2. Construcción de un AFD a partir de una expresión regular

Podemos construir un AFD a partir de una expresión regular. La idea básica es llevar en cada estado del autómata la cuenta de en qué parte de la expresión "estamos". Para esto, emplearemos lo que llamaremos *ítems*, que serán expresiones regulares con un punto centrado (·). El punto indicará qué parte de la expresión corresponde con la entrada leída hasta alcanzar el estado en el que está el ítem. Por ejemplo, dada la expresión regular (ab)*cd(ef)*, un posible ítem sería (ab)*c·d(ef)*. Este ítem lo podríamos encontrar tras haber leído, por ejemplo, la entrada ababc. Si el ítem tiene el punto delante de un carácter, de una clase de caracteres o al final de la expresión diremos que es un *ítem básico*. Los estados de nuestro autómata serán conjuntos de ítems básicos sobre la expresión regular. Utilizamos conjuntos porque en un momento dado podemos estar en más de un punto en la expresión regular. Por ejemplo, en la expresión regular (a)*b tras leer una a, podemos estar esperando otra a o una b, así pues, estaríamos en el estado {(·a)*b, (a)*·b}.

5.2.1. Algunos ejemplos

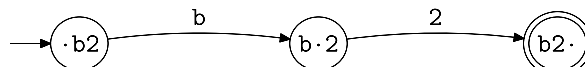
Empezaremos por un ejemplo muy sencillo. Vamos a analizar la cadena b2 con la expresión regular b2. Al comienzo, no hemos analizado nada y el ítem que muestra esto es ·b2. El primer carácter es la b, que es trivialmente aceptado por b, así que avanzamos el punto para llegar a b·2. Ahora leemos el 2 y avanzamos a b2·. Dado que hemos terminado la entrada y hemos llegado al final de la expresión, podemos concluir que aceptamos la cadena. Podemos resumir el proceso en la siguiente tabla:

Estado	Entrada	Observaciones
{·b2}	b	Estado inicial
{b·2}	2	
{b2·}	—	Aceptamos

Cosas que hemos aprendido con este ejemplo: que si al leer un carácter, el punto está delante de él, podemos moverlo detrás. Por otro lado, hemos visto que aceptaremos la cadena de entrada si tenemos el punto al final de la expresión y se ha terminado la entrada. Dicho de otro modo, los estados que tengan ítems con el punto en la última posición serán finales. Construir un autómata a partir de aquí es trivial:

- Los estados serán: {·b2}, {b·2} y {b2·}.
- Como estado inicial usamos {·b2}.
- Con la b pasamos de {·b2} a {b·2}.
- Con el 2 pasamos de {b·2} a {b2·}.
- El único estado final es {b2·}.

Así obtenemos el autómata siguiente[4]:



Lógicamente, las cosas se ponen más interesantes cuando la expresión regular no es únicamente una cadena. Vamos a ver cómo podemos trabajar con clases de caracteres. Como ejemplo, analizaremos la cadena b2 con la expresión regular [a–z][0–9]. Al comienzo, no hemos analizado nada, así que el ítem que muestra esto es ·[a–z][0–9]. El primer carácter es la b, que es aceptado por [a–z], así que avanzamos el punto para llegar a [a–z]·[0–9]. Ahora leemos el 2 y avanzamos a [a–z][0–9]·.

[4]Aunque en los estados tendremos conjuntos de ítems, no pondremos las llaves para no sobrecargar las figuras.

© Universitat Jaume I 2008-2009

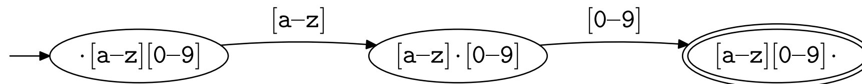
Como antes, aceptamos `b2` ya que hemos llegado al final de la expresión y se ha terminado la entrada. En forma de tabla, hemos hecho:

Estado	Entrada	Observaciones
{·[a–z][0–9]}	`b`	Estado inicial
{[a–z]·[0–9]}	2	
{[a–z][0–9]·}	—	Aceptamos

Como vemos, el punto se puede mover sobre una clase de caracteres si el carácter en la entrada es uno de los de la clase. Podemos construir ahora un AFD de la manera siguiente:

- Los estados serán: {·[a–z][0–9]}, {[a–z]·[0–9]} y {[a–z][0–9]·}.
- Como estado inicial usamos {·[a–z][0–9]}.
- Con cada una de las letras de [a–z] pasamos del estado inicial a {[a–z]·[0–9]}.
- Con los dígitos pasamos de {[a–z]·[0–9]} a {[a–z][0–9]·}.
- El único estado final es {[a–z][0–9]·}.

Así, obtenemos el autómata



Aunque es tentador suponer que si el punto está delante de una clase de caracteres tendremos un arco que sale con esa clase, no te dejes engañar, es sólo una casualidad. En algunos casos no es así, mira, por ejemplo, el ejercicio 17.

Vamos ahora a añadir clausuras. Supongamos que tenemos la expresión regular ([a–z])*[0–9]. Hemos puesto paréntesis para tener más claro dónde está el punto; también tendremos que hacerlo con las disyunciones. Hagamos como antes, vamos a analizar `b2`. Empezamos con el ítem que tiene el punto delante de la expresión regular. Con esto tenemos ·([a–z])*[0–9]. Dado que el punto no está delante de ningún carácter o clase de caracteres, tendremos que encontrar qué ítems básicos constituyen nuestro estado inicial. Como el punto está delante de la clausura, podemos bien leer una letra (o más) o bien pasar a leer un dígito. Lo que vamos a hacer es reflejar las dos posibilidades en nuestro estado mediante dos ítems: (·[a–z])*[0–9] y ([a–z])*·[0–9]. Como estamos viendo la `b`, sólo podemos mover el punto que está delante de [a–z]. Pasamos entonces a ([a–z]·)*[0–9]. Nuevamente tenemos un ítem no básico, así que volvemos a buscar los correspondientes ítems básicos. En este caso, podemos volver al comienzo de la clausura o salir de ella. Es decir, tenemos que pasar al estado que contiene (·[a–z])*[0–9] y ([a–z])*·[0–9]. Con el 2 llegamos a ([a–z])*[0–9]· y aceptamos. Si resumimos el proceso en forma de tabla, tenemos:

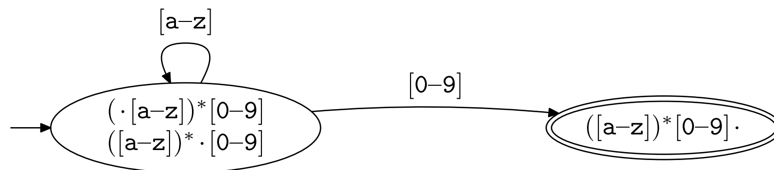
Estado	Entrada	Observaciones
{(·[a–z])*[0–9], ([a–z])*·[0–9]}	`b`	Estado inicial
{(·[a–z])*[0–9], ([a–z])*·[0–9]}	2	Estado inicial
{([a–z])*[0–9]·}	—	Aceptamos

Así pues, un punto situado inmediatamente antes de una clausura puede "atravesar" esta —lo que equivale a decir que ha generado la cadena vacía— o ponerse al comienzo de su interior para generar una o más copias de ese interior. Así mismo, cuando el punto está delante del paréntesis

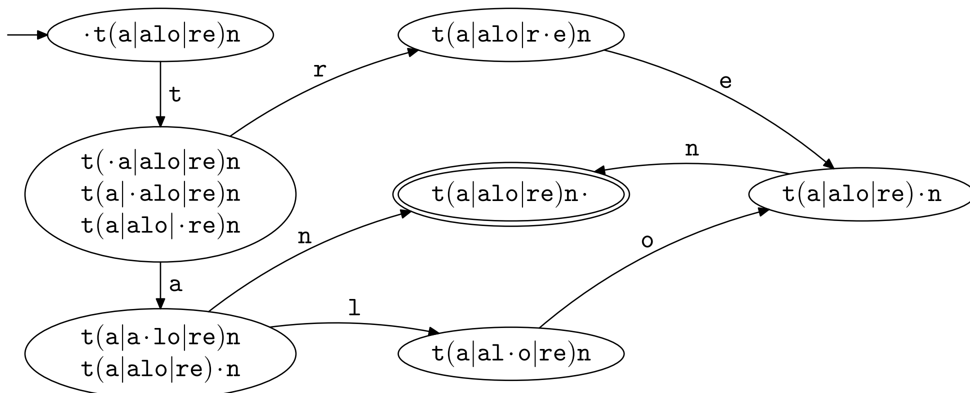
de cierre de la clausura puede “salir” de ella o volver al principio. Para ver cómo se permiten las repeticiones, vamos a analizar `ab4`:

Estado	Entrada	Observaciones
$\{(\cdot[\text{a–z}])^*[0–9],$ $([\text{a–z}])^*\cdot[0–9]\}$	a	Estado inicial
$\{(\cdot[\text{a–z}])^*[0–9],$ $([\text{a–z}])^*\cdot[0–9]\}$	b	Estado inicial
$\{(\cdot[\text{a–z}])^*[0–9],$ $([\text{a–z}])^*\cdot[0–9]\}$	4	Estado inicial
$\{([\text{a–z}])^*[0–9]\cdot\}$	—	Aceptamos

Puedes darte cuenta de que, como era de esperar, tanto `a` como `b` llevan al mismo estado. Podemos construir el autómata correspondiente:



Por último, vamos a ver cómo trabajar con las disyunciones. Vamos a intentar construir el autómata para t(a|alo|re)n. El estado inicial no tiene problemas: {·t(a|re|alo)n}. Con una t pasaremos al ítem no básico t·(a|alo|re)n. A partir de él, se producen tres ítems básicos, que constituyen el siguiente estado de nuestro autómata: {t(·a|alo|re)n, t(a|·alo|re)n, t(a|alo|·re)n}. Desde este estado, con una a obtenemos dos ítems: t(a·|alo|re)n y t(a|a·lo|re)n. El segundo es básico, así que sólo tendremos que ver lo que pasa con el primer ítem. Podemos ver que basta con pasar el punto detrás del paréntesis para obtener t(a|alo|re)·n. De manera similar, si tuviéramos el punto en t(a|alo·|re)n o en t(a|alo|re·)n, pasaríamos a t(a|alo|re)·n. Teniendo en cuenta esto, el autómata nos queda:



5.2.2. Formalización

Intentaremos ahora sistematizar lo que hemos ido haciendo. Empezaremos por el proceso de, a partir de un conjunto de ítems, encontrar los ítems básicos que constituirán el estado del autómata.

Cuadro 1: Transformaciones de los ítems no básicos.

Ítem original	Nuevos ítems
$\alpha\cdot(\beta)\gamma$	$\alpha(\cdot\beta)\gamma$
$\alpha(\beta\cdot)\gamma$	$\alpha(\beta)\cdot\gamma$
$\alpha\cdot(\beta)^*\gamma$	$\alpha(\cdot\beta)^*\gamma$ $\alpha(\beta)^*\cdot\gamma$
$\alpha(\beta\cdot)^*\gamma$	$\alpha(\cdot\beta)^*\gamma$ $\alpha(\beta)^*\cdot\gamma$

Ítem original	Nuevos ítems
$\alpha\cdot\lambda\gamma$	$\alpha\lambda\cdot\gamma$
$\alpha\cdot(\beta_1\|\beta_2\|\ldots\|\beta_n)\gamma$	$\alpha(\cdot\beta_1\|\beta_2\|\ldots\|\beta_n)\gamma$ $\alpha(\beta_1\|\cdot\beta_2\|\ldots\|\beta_n)\gamma$ $\ldots$ $\alpha(\beta_1\|\beta_2\|\ldots\|\cdot\beta_n)\gamma$
$\alpha(\ldots\|\beta_i\cdot\|\ldots)\gamma$	$\alpha(\ldots\|\beta_i\|\ldots)\cdot\gamma$

Llamaremos *clausura* a este proceso y el algoritmo que seguiremos será el siguiente:

Algoritmo clausura (S)
$C := S$;
$v := \emptyset$; // ítems ya tratados
mientras haya ítems no básicos en C **hacer**
sea i un ítem no básico de C;
$C := C - \{i\}$;
$v := v \cup \{i\}$;
$C := C \cup \text{transformaciones}(i) - v$;
fin mientras
devolver C;
Fin clausura

La idea básica es sustituir cada ítem no básico por una serie de transformaciones que se corresponden con los movimientos que hacíamos del punto durante los ejemplos. En el cuadro 1 puedes encontrar las transformaciones para las expresiones regulares básicas. Es interesante darse cuenta de la diferencia entre los paréntesis abiertos que encierran una subexpresión y aquellos que corresponden a una disyunción o una clausura.

Una vez sabemos cómo calcular la clausura de un estado, podemos construir los movimientos del autómata. Si estamos en un estado que contiene una serie de ítems básicos, el estado al que iremos al consumir un carácter de la entrada será la clausura del conjunto resultante de avanzar el punto una posición en aquellos ítems que lo tengan bien delante del carácter, bien delante de una clase de caracteres que lo incluyan. Así, en los ejemplos anteriores teníamos que con `b` se pasaba de $\{(\cdot[\mathsf{a}–\mathsf{z}])^*[0–9], ([\mathsf{a}–\mathsf{z}])^*\cdot[0–9]\}$ a la clausura de $\{([\mathsf{a}–\mathsf{z}]\cdot)^*[0–9]\}$, ya que la `b` es compatible con [a–z] pero no con [0–9]. El algoritmo para calcular el siguiente de un estado dado un símbolo resulta muy sencillo:

Algoritmo siguiente(q, x)
devolver clausura($\{\alpha\beta\cdot\gamma \mid \alpha\cdot\beta\gamma \in q \wedge x \in L(\beta)\}$);
Fin

Ahora tenemos todas las herramientas para construir nuestro autómata. El estado inicial será la clausura del ítem formado anteponiendo el punto a la expresión regular. Después iremos viendo todos los posibles movimientos desde este estado. Mantendremos un conjunto con los estados que vayamos creando y que nos queden por analizar. Como estados finales, pondremos aquellos que

tengan el punto al final. Con todo esto, el algoritmo resultante es:

Algoritmo construcción(α)
  q_0 :=clausura($\cdot\alpha$);
  $Q := \{q_0\}$; $E := \emptyset$
  $\Pi := Q$; // pendientes
  mientras $\Pi \neq \emptyset$ **hacer**
    q :=arbitrario(Π);
    $\Pi := \Pi - \{q\}$;
    para todo $x \in \Sigma$ **hacer**
      si siguiente(q, x) $\neq \emptyset$ **entonces**
        q' :=siguiente(q, x);
        si $q' \notin Q$ **entonces**
          $Q := Q \cup \{q'\}$;
          $\Pi := \Pi \cup \{q'\}$;
        fin si
        $E := E \cup \{(q, x, q')\}$;
      fin si
    fin para todo
  fin mientras
  $F := \{q \in Q \mid \exists\alpha\cdot \in q\}$;
  devolver (Q, Σ, E, q_0, F);
Fin

Ejercicio 16

Construye los AFDs correspondientes a las siguientes expresiones regulares:

a) $(\mathtt{a}|\lambda)\mathtt{b}^*$

b) $(\mathtt{a}|\lambda)\mathtt{b}^*\mathtt{b}$

c) $((\mathtt{a}|\lambda)\mathtt{b}^*)^*$

d) $((\mathtt{a}|\lambda)\mathtt{b}^*)^*\mathtt{b}$

Ejercicio 17

Construye el AFD correspondiente a la expresión regular $[\mathtt{a}\text{–}\mathtt{z}]^*[\mathtt{a}\text{–}\mathtt{z0}\text{–}\mathtt{9}]$. Ten cuidado al elegir las clases de caracteres de los arcos.

Ejercicio 18

Completa el cuadro 1 para los operadores de opcionalidad y clausura positiva. Construye los AFDs correspondientes a las siguientes expresiones regulares:

a) `ab?c`

b) `ab?b`

c) $\mathtt{ab}^+\mathtt{c}$

d) $\mathtt{ab}^+\mathtt{b}$

Ejercicio 19

Utiliza el método de los ítems para escribir AFDs para las siguientes categorías léxicas:

a) Las palabras `if`, `in` e `input`.

b) Números expresados en hexadecimal escritos como dos símbolos de porcentaje seguidos de dígitos hexadecimales de modo que las letras estén todas, bien en mayúsculas, bien en minúsculas.

c) Números reales compuestos de una parte entera seguida de un punto y una parte decimal. Tanto la parte entera como la decimal son obligatorias.

d) Los nombres válidos como **identificador**: palabras de uno o más caracteres que empiezan por una letra y continúan con letras o dígitos. Las letras pueden ser mayúsculas o minúsculas.

EJERCICIO* 20

¿Son mínimos los autómatas obtenidos con el método de los ítems?

EJERCICIO* 21

Construye los AFDs correspondientes a las expresiones del ejercicio 8.

5.3. Compromisos entre espacio y tiempo

La construcción anterior nos permite pasar de una expresión regular a un AFD y emplear este para analizar cadenas. Sin embargo, esta puede no ser la solución óptima para todos los casos en que se empleen expresiones regulares. Hay por lo menos dos maneras más de trabajar con las expresiones regulares: emplear autómatas finitos no deterministas (AFN) y utilizar *backtracking*.

La construcción de un autómata no determinista es más sencilla que la de uno determinista y proporciona autómatas de menor número de estados que los deterministas (el número de estados de un AFD puede ser exponencial con la talla de la expresión regular, el del AFN es lineal). A cambio, el análisis es más costoso (con el AFN el coste temporal es $O(|x|\,|r|)$, por $O(|x|)$ para el AFD, donde $|x|$ es la talla de la cadena y $|r|$ la de la expresión regular). Esto hace que sea una opción atractiva para situaciones en las que la expresión se vaya a utilizar pocas veces.

Cuando se utiliza backtracking, lo que se hace es "ir probando" a ver qué partes de la expresión se corresponden con la entrada y se vuelve atrás si es necesario. Por ejemplo, supongamos que tenemos la expresión regular `a(b|bc)c` y la entrada `abcc`. Para analizarla, primero vemos que la `a` está presente en la entrada. Llegamos a la disyunción e intentamos utilizar la primera opción. Ningún problema, avanzamos en la entrada a la primera `c` y en la expresión a la `c` del final. Volvemos a avanzar y vemos que la expresión se ha terminado y no la entrada. Lo que hacemos es volver atrás (es decir, retrocedemos en la entrada hasta el momento de leer la `b`) y escoger la segunda opción de la disyunción. El análisis prosigue ahora normalmente y la cadena es aceptada. Podemos ver que el uso de backtracking minimiza la utilización del espacio (sólo necesitamos almacenar la expresión), pero puede aumentar notablemente el tiempo de análisis (hasta hacerlo exponencial con el tamaño de la entrada).

Cuando se trabaja con compiladores e intérpretes, las expresiones se utilizan muchas veces, por lo que merece la pena utilizar AFDs e incluso minimizarlos (hay algoritmos de minimización con coste $O(|Q|\,\log(|Q|))$, donde $|Q|$ es el número de estados del autómata). En otras aplicaciones, por ejemplo facilidades de búsqueda de un procesador de textos, la expresión se empleará pocas veces, así que es habitual que se emplee backtracking o autómatas no deterministas si se quieren tener cotas temporales razonables.

Como resumen de los costes, podemos emplear el siguiente cuadro. Recuerda que $|r|$ es la talla de la expresión regular y $|x|$ la de la cadena que queremos analizar.

Método	Memoria	Tiempo
AFD	exponencial	$O(\|x\|)$
AFN	$O(\|r\|)$	$O(\|x\|\,\|r\|)$
Backtracking	$O(\|r\|)$	exponencial

De todas formas, en la práctica los casos en los que los costes son exponenciales son raros, por lo que el número de estados del AFD o el tiempo empleado por backtracking suele ser aceptable.

Ejercicio* 22

¿Puedes dar ejemplos de expresiones regulares que requieran un coste exponencial para su transformación en AFD o su análisis por backtracking?

6. Implementación del analizador léxico

Hasta ahora hemos visto cómo especificar los lenguajes asociados a las diferentes categorías léxicas. Sin embargo, el analizador léxico no se utiliza para comprobar si una cadena pertenece o no a un lenguaje. Lo que debe hacer es dividir la entrada en una serie de componente léxicos, realizando en cada uno de ellos determinadas acciones. Algunas de estas acciones son: comprobar alguna restricción adicional (por ejemplo que el valor de un literal entero esté dentro de un rango), preparar los atributos del componente y *emitir* u *omitir* dicho componente.

Así pues, la especificación del analizador léxico deberá incluir por cada categoría léxica del lenguaje el conjunto de atributos y acciones asociadas. Un ejemplo sería:

categoría	**expresión regular**	**acciones**	**atributos**
entero	[0–9]$^+$	calcular valor comprobar rango emitir	valor
real	[0–9]$^+$\.[0–9]$^+$	calcular valor comprobar rango emitir	valor
identificador	[a–zA–Z][a–zA–Z0–9]*	copiar lexema emitir	lexema
asignación	:=	emitir	
rango	\.\.	emitir	
blanco	[␣\t\n]$^+$	omitir	
eof	eof	emitir	

6.1. La estrategia avariciosa

Inicialmente puede parecer que podemos especificar completamente el analizador dando una lista de expresiones regulares para las distintas categorías. Sin embargo, debemos definir algo más: cómo esperamos que se unan estas categorías para formar la entrada.

Analicemos el siguiente fragmento de un programa utilizando las categorías anteriores:

```
c3:= 10.2
```

En principio, esperamos encontrar la siguiente secuencia de componentes léxicos:

id lex: c3	asignación	real valor: 10.2

Sin embargo, no hemos definido ningún criterio por el que la secuencia no pudiera ser esta:

id lex: c	entero valor: 3	asignación	entero valor: 1	real valor: 0.2

Lo que se suele hacer es emplear la denominada “estrategia avariciosa”: en cada momento se busca en la parte de la entrada no analizada el prefijo más largo que pertenezca al lenguaje asociado a alguna categoría léxica. De esta manera, la división de la entrada coincide con la que esperamos.

6.2. Máquina discriminadora determinista

Para reconocer simultáneamente las distintas categorías léxicas utilizaremos un tipo de autómata muy similar a los AFDs: la *máquina discriminadora determinista* (MDD). El funcionamiento

© Universitat Jaume I 2008-2009

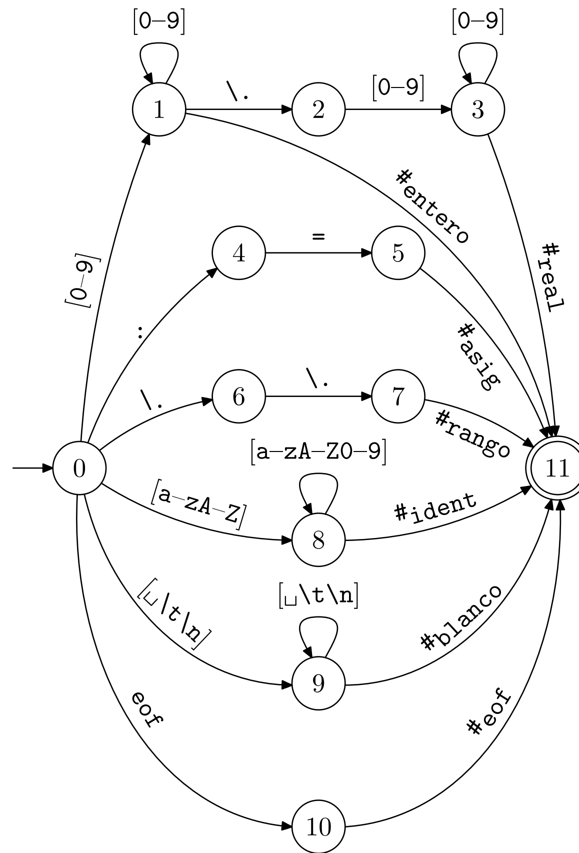


Figura 1: AFD asociado a la expresión regular completa.

de las MDDs es muy similar al de los AFDs, las diferencias son dos: además de cadenas completas, aceptan prefijos de la entrada y tienen asociadas acciones a los estados finales.

Estudiaremos la construcción a partir del ejemplo de especificación anterior. Empezamos por añadir a cada expresión regular un símbolo especial:

Categoría	Expresión regular
entero	$[0\text{–}9]^+\#_{\text{entero}}$
real	$[0\text{–}9]^+\backslash.[0\text{–}9]^+\#_{\text{real}}$
identificador	$[\text{a–zA–Z}][\text{a–zA–Z0–9}]^*\#_{\text{ident}}$
asignación	$:=\#_{\text{asig}}$
rango	$\backslash.\backslash.\#_{\text{rango}}$
blanco	$[␣\backslash\text{t}\backslash\text{n}]^+\#_{\text{blanco}}$
eof	$\text{eof}\#_{\text{eof}}$

Después unimos todas las expresiones en una sola:

$$[0\text{–}9]^+\#_{\text{entero}}|[0\text{–}9]^+\backslash.[0\text{–}9]^+\#_{\text{real}}|$$
$$[\text{a–zA–Z}][\text{a–zA–Z0–9}]^*\#_{\text{ident}}|:=\#_{\text{asig}}|\backslash.\backslash.\#_{\text{rango}}|$$
$$[␣\backslash\text{t}\backslash\text{n}]^+\#_{\text{blanco}}|\text{eof}\#_{\text{eof}}$$

Ahora construimos el AFD asociado a esta expresión regular. Este AFD lo puedes ver en la figura 1.

Por construcción, los únicos arcos que pueden llegar a un estado final serán los etiquetados con símbolos especiales. Lo que hacemos ahora es eliminar los estados finales (en realidad sólo habrá uno) y los arcos con símbolos especiales. Como nuevos estados finales dejamos aquellos desde los que partían arcos con símbolos especiales. A estos estados asociaremos las acciones correspondientes al reconocimiento de las categorías léxicas. Nuestro ejemplo queda como se ve en la figura 2.

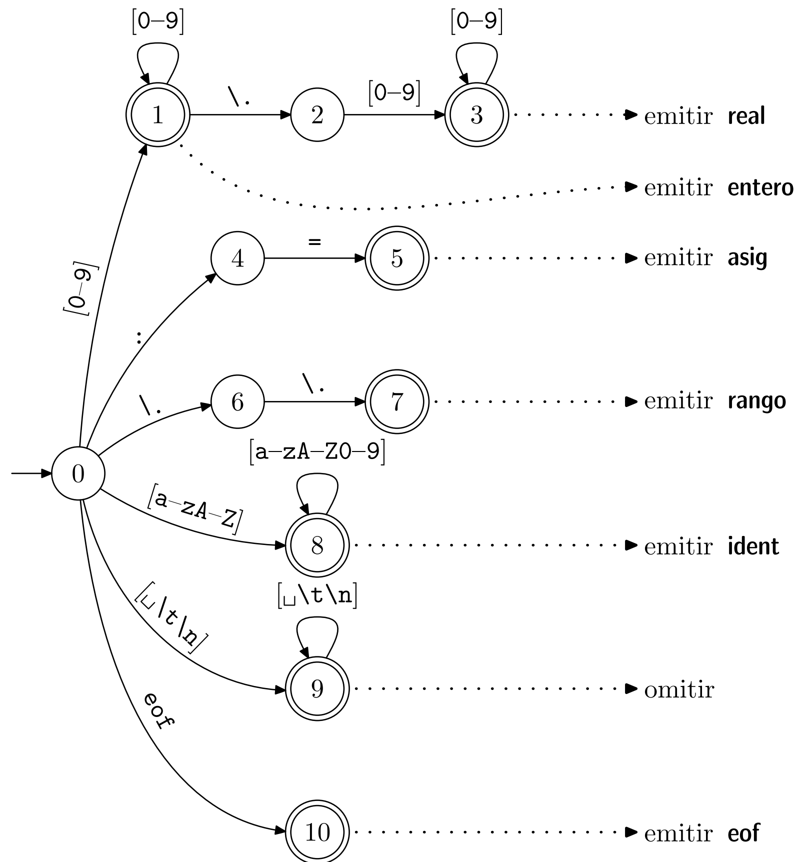


Figura 2: MDD correspondiente al ejemplo. Hemos simplificado las acciones para evitar sobrecargar la figura.

6.2.1. Funcionamiento de la MDD

¿Cómo funciona una MDD? La idea del funcionamiento es muy similar a la de los AFD, salvo por dos aspectos:

- La MDD no intenta reconocer la entrada sino segmentarla.
- La MDD actúa repetidamente sobre la entrada, empezando en cada caso en un punto distinto, pero siempre en su estado inicial.

Sea x la cadena que queremos segmentar y $\mathcal{M}$ nuestra MDD. Queremos encontrar una secuencia de cadenas $\{x_i\}_{i=1}^n$ tales que:

- Su concatenación sea la cadena original: $x_1 \dots x_n = x$.
- Cada una de ellas es aceptada por $\mathcal{M}$, es decir, que para cada x_i existe un camino en $\mathcal{M}$ que parte del estado inicial y lleva a uno final.

© Universitat Jaume I 2008-2009

- Queremos que sigan la estrategia avariciosa. Esto quiere decir que no es posible encontrar para ningún i una cadena y que sea aceptada por $\mathcal{M}$, que sea prefijo de $x_i \dots x_n$ y que sea más larga que x_i.

Afortunadamente, encontrar esta segmentación es muy sencillo. Empezamos por poner $\mathcal{M}$ en su estado inicial. Después actuamos como si $\mathcal{M}$ fuera un AFD, es decir, vamos cambiando el estado de la máquina según nos indica la entrada. En algún momento llegaremos a un estado q desde el que no podemos seguir avanzando. Esto puede suceder bien porque no existe ninguna transición de q con el símbolo actual, bien porque se nos haya terminado la entrada. Si q es final, acabamos de encontrar un componente léxico. Ejecutamos las acciones asociadas a q y volvemos a empezar. Si q no es final, tenemos que "desandar" lo andado, hasta que lleguemos al último estado final por el que hayamos pasado. Este será el que indique el componente léxico encontrado. Si no existiera tal estado, estaríamos ante un error léxico.

La descripción dada puede parecer un tanto liosa, vamos a ver un ejemplo. Supongamos que tenemos la entrada siguiente:

```
␣␣a:=␣9.1
```

Inicialmente la máquina está en el estado 0. Con los dos primeros espacios, llegamos al estado 9. El siguiente caracter (la `a`) no tiene transición desde aquí, así que miramos el estado actual. Dado que es final, ejecutamos la acción correspondiente, en este caso *omitir*. Volvemos al estado 0. Ahora estamos viendo la `a`, así que transitamos al estado 8. Nuevamente nos encontramos con un carácter que no tiene transición. En este caso, la acción es *emitir*, así que emitimos el identificador, tras haber copiado el lexema. De manera análoga, se emitiría un **asig**, se omitiría el espacio y se emitiría un real.

Vamos a ver ahora qué pasa con la entrada siguiente:

```
9..12
```

Como antes, empezamos en el estado 0. Con el `9` llegamos al estado 1. Después, con el `.` vamos al estado 2. Ahora resulta que no podemos transitar con el nuevo punto. Lo que tenemos que hacer es retroceder hasta el último estado final por el que hemos pasado, en este caso el 1. Lógicamente, al retroceder, debemos "devolver" los símbolos no consumidos, en nuestro caso los dos puntos. De esta manera, hemos encontrado que `9` es un entero y lo emitimos tras calcular el valor y comprobar el rango. ¿Qué hacemos ahora? Volvemos al estado cero. El carácter que vamos a leer ahora es nuevamente el primero de los dos puntos. De esta manera transitamos al estado 6 y después al 7. Emitimos rango y volvemos a empezar para emitir otro entero.

Por último, veamos qué pasa con:

```
a.3
```

En primer lugar, emitimos el identificador con lexema `a`. Después transitamos con el punto al estado 6. En este momento no podemos transitar con el tres. Pero además resulta que no hemos pasado por estado final alguno. Lo que sucede es que hemos descubierto un error.

Ejercicio 23

Construye una MDD para la siguiente especificación:

categoría	expresión regular	acciones	atributos
número	`-?[0-9]+(\.[0-9]*)?`	calcular valor averiguar tipo emitir	valor, tipo
asignacion	`[-+*/]?=`	separar operador emitir	operador
operador	`[-+*/]`	copiar lexema emitir	lexema
comentario	`|[^\n]*\n`	omitir	
blanco	`[␣\t\n]`	omitir	
id	`[a-zA-Z_][a-zA-Z0-9_]*`	copiar lexema emitir	lexema
eof	eof	emitir	

¿Cómo se comportaría ante las siguientes entradas?

a) `a1+=␣a+1`

b) `a␣1+=␣a-1`

c) `a1+␣=␣a-␣1`

d) `a1+=␣a-1␣|+1`

6.3. Tratamiento de errores

Hemos visto que si la MDD no puede transitar desde un determinado estado y no ha pasado por estado final alguno, se ha detectado un error léxico. Tenemos que tratarlo de alguna manera adecuada. Desafortunadamente, es difícil saber cuál ha sido el error. Si seguimos con el ejemplo anterior, podemos preguntarnos la causa del error ante la entrada `a.3`. Podría suceder que estuviésemos escribiendo un real y hubiésemos sustituido el primer dígito por la `a`. Por otro lado, podría ser un rango mal escrito por haber borrado el segundo punto. También podría suceder que el punto sobrase y quisiéramos escribir el identificador `a3`. Como puedes ver, aún en un caso tan sencillo es imposible saber qué ha pasado.

Una posible estrategia ante los errores sería definir una "distancia" entre programas y buscar el programa más próximo a la entrada encontrada. Desgraciadamente, esto resulta bastante costoso y las ganancias en la práctica suelen ser mínimas. La estrategia que seguiremos es la siguiente:

- Emitir un mensaje de error y detener la generación de código.
- Devolver al flujo de entrada todos los caracteres leídos desde que se detectó el último componente léxico.
- Eliminar el primer carácter.
- Continuar el análisis.

Siguiendo estos pasos, ante la entrada `a.3`, primero se emitiría un identificador, después se señalaría el error al leer el `3`. Se devolverían el tres y el punto a la entrada. Se eliminaría el punto y se seguiría el análisis emitiendo un entero.

6.3.1. Uso de categorías léxicas para el tratamiento de errores

La estrategia anterior garantiza que el análisis continúa y que finalmente habremos analizado toda la entrada. Sin embargo, en muchas ocasiones los mensajes de error que puede emitir son muy poco informativos y puede provocar errores en las fases siguientes del análisis. En algunos casos, se puede facilitar la emisión de mensajes de error mediante el uso de “pseudo-categorías léxicas”.

Supongamos que en el ejemplo anterior queremos tratar los siguientes errores:

- No puede aparecer un punto seguido de un entero. En este caso queremos que se trate como si fuera un número real con parte entera nula.
- No puede aparecer un punto aislado. En este caso queremos que se asuma que en realidad era un rango.

Para lograrlo, podemos añadir a la especificación las siguientes líneas:

categoría	expresión regular	acciones	atributos
errorreal	`\.`[0–9]$^+$	informar del error calcular valor emitir real	valor
errorrango	`\.`	informar del error emitir rango	

Con esta nueva especificación, la entrada `v:=.3` haría que se emitiesen un identificador, una asignación y un real, además de informar del error.

Pese a las ventajas que ofrecen estas categorías, hay que tener mucho cuidado al trabajar con ellas. Si en el caso anterior hubiéramos añadido `[0-9]+\.` a la pseudo-categoría **errorreal** con la intención de capturar reales sin la parte decimal, habríamos tenido problemas. Así, la entrada `1..2` devolvería dos errores (`1.` y `.2`).

Una categoría que suele permitir de manera natural este tipo de estrategia es el literal de cadena. Un error frecuente es olvidar las comillas de cierre de una cadena. Si las cadenas no pueden abarcar varias líneas, es fácil (y un ejercicio deseable para ti) escribir una expresión regular para detectar las cadenas no cerradas.

6.3.2. Detección de errores en las acciones asociadas

Otros errores que se detectan en el nivel léxico son aquellos que sólo pueden encontrarse al ejecutar las acciones asociadas a las categorías. Un ejemplo es el de encontrar números fuera de rango. En principio es posible escribir una expresión regular para, por ejemplo, los enteros de 32 bits. Sin embargo, esta expresión resulta excesivamente complicada y se puede hacer la comprobación fácilmente mediante una simple comparación. Esta es la estrategia que hemos seguido en nuestro ejemplo. Aquí sí que es de esperar un mensaje de error bastante informativo y una buena recuperación: basta con emitir la categoría con un valor predeterminado (por ejemplo cero).

6.4. Algunas categorías especiales

Ahora comentaremos algunas categorías que suelen estar presentes en muchos lenguajes y que tienen ciertas peculiaridades.

Categorías que se suelen omitir Hemos visto en nuestros ejemplos que los espacios generalmente no tienen significado. Esto hace que sea habitual tener una expresión del tipo [␣`\t\n`]$^+$ con *omitir* como acción asociada. Aún así, el analizador léxico sí que suele tenerlos en cuenta para poder dar mensajes de error. En particular, es bueno llevar la cuenta de los `\n` para informar de la línea del error. En algunos casos, también se tienen en cuenta los espacios para poder dar la posición del error dentro de la línea.

Otra categoría que es generalmente omitida es el comentario. Como en el caso anterior, el analizador léxico suele analizar los comentarios para comprobar cuántos fines de línea están incluidos en ellos y poder informar adecuadamente de la posición de los errores.

El fin de fichero Aunque ha aparecido como un símbolo más al hablar de las expresiones regulares, lo cierto es que el fin de fichero no suele ser un carácter más. Dependiendo del lenguaje en que estemos programando o incluso dependiendo de las rutinas que tengamos, el fin de fichero aparecerá de una manera u otra. En algunos casos es un valor especial (por ejemplo en la función `getc` de C), en otros debe consultarse explícitamente (por ejemplo en Pascal) y en otros consiste en la devolución de la cadena vacía (por ejemplo en Python). Como esto son detalles de implementación, nosotros mostraremos explícitamente el fin de fichero como si fuera un carácter eof.

Es más, en ocasiones ni siquiera tendremos un fichero que finalice. Puede suceder, por ejemplo, que estemos analizando la entrada línea a línea y lo que nos interesa es que se termina la cadena que representa la línea. También puede suceder que estemos en un sistema interactivo y analicemos una cadena introducida por el usuario.

Para unificar todos los casos, supondremos que el analizador léxico indica que ya no tiene más entrada que analizar mediante la emisión de esta categoría y no utilizaremos eof como parte de ninguna expresión regular compuesta.

Las palabras clave De manera implícita, hemos supuesto que las expresiones regulares que empleamos para especificar las categorías léxicas definen lenguajes disjuntos dos a dos. Aunque esta es la situación ideal, en diversas ocasiones puede suponer una fuente de incomodidades. El caso más claro es el que sucede con las palabras clave y su relación con los identificadores.

Es habitual que los identificadores y las palabras clave tengan “la misma forma”. Por ejemplo, supongamos que los identificadores son cadenas no vacías de letras minúsculas y que `if` es una palabra clave. ¿Qué sucede al ver la cadena `if`? La respuesta depende en primer lugar del lenguaje. En bastantes lenguajes, las palabras clave son *reservadas*, por lo que el analizador léxico deberá clasificar la cadena `if` en la categoría léxica **if**. Sin embargo, en lenguajes como PL/I, el siguiente fragmento

```
if then then then = else ; else else = then;
```

es perfectamente válido. En este caso, el analizador léxico necesita información sintáctica para decidir si `then` es una palabra reservada o un identificador.

Si decidimos que las palabras clave son reservadas, tenemos que resolver dos problemas:

- Cómo lo reflejamos en la especificación del analizador léxico.
- Cómo lo reflejamos al construir la MDD.

Puede parecer que el primer problema está resuelto; simplemente se escriben las expresiones regulares adecuadas. Sin embargo, las expresiones resultantes pueden ser bastante complicadas. Intentemos escribir la expresión para el caso anterior, es decir, identificadores consistentes en cadenas de letras minúsculas, pero sin incluir la cadena `if`. La expresión es:

$$\texttt{[a–hj–z][a–z]}^*|\texttt{i}(\texttt{[a–eg–z][a–z]}^*)?|\texttt{if[a–z]}^+$$

Ejercicio* 24

Supón que en el lenguaje del ejemplo anterior tenemos tres palabras clave reservadas: `if`, `then` y `else`. Diseña una expresión regular para los identificadores que excluya estas tres palabras clave.

Este sistema no sólo es incómodo para escribir, facilitando la aparición de errores, también hace que añadir una nueva palabra clave sea una pesadilla. Lo que se hace en la práctica es utilizar un sistema de prioridades. En nuestro caso, cuando un lexema pueda clasificarse en dos categorías

© Universitat Jaume I 2008-2009

léxicas distintas, escogeremos la categoría que aparece en primer lugar en la especificación. Así para las palabras del ejercicio, nuestra especificación podría ser:

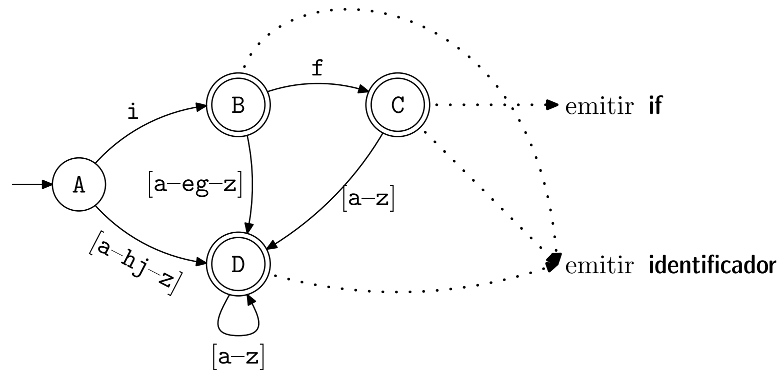
categoría	expresión regular	acciones	atributos
if	`if`	emitir	
then	`then`	emitir	
else	`else`	emitir	
identificador	$[a–z]^+$	copiar lexema emitir	lexema

Ahora que sabemos cómo especificar el analizador para estos casos, nos queda ver la manera de construir la MDD. En principio, podemos pensar en aplicar directamente la construcción de la MDD sobre esta especificación. Sin embargo este método resulta excesivamente complicado. Por ello veremos otra estrategia que simplifica la construcción de la MDD con el precio de algunos cálculos extra.

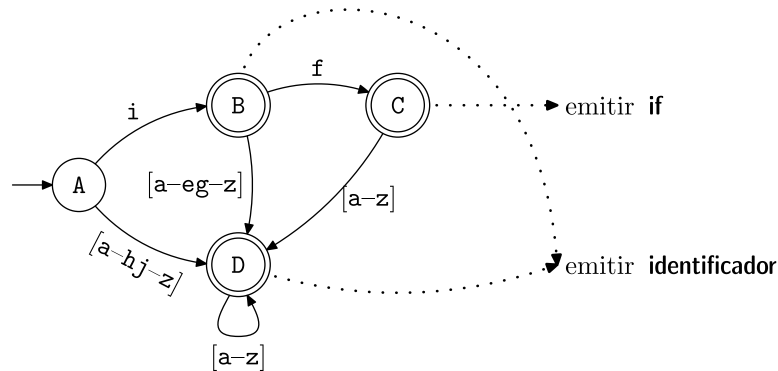
Para ver lo complicada que puede resultar la construcción directa, volvamos al caso en que sólo tenemos identificadores formados por letras minúsculas y una palabra reservada: `if`. Empezamos por escribir la expresión regular con las marcas especiales:

$$\texttt{if\#}_{\texttt{if}} | [a–z]^+ \texttt{\#}_{\texttt{identificador}}$$

Con esta expresión construimos la MDD:

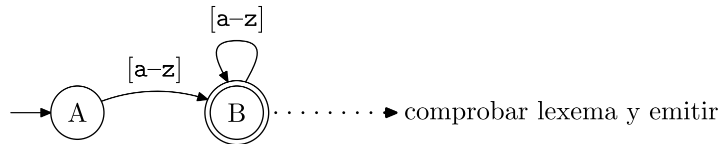


Vemos que la ambigüedad aparece en el estado `C`, que tiene dos posibles acciones. Hemos decidido que **if** tiene preferencia sobre **identificador**, así que dejamos el autómata de esta manera:



Vemos que incluso para algo tan sencillo, el autómata se complica. Evidentemente, a medida que crece el número de palabras reservadas, la complejidad también aumenta.

La solución que podemos adoptar es la siguiente:



De esta manera, cualquier lexema parecido a un identificador terminará en el estado `B`. Bastará entonces comprobar si realmente es un identificador o una palabra clave y emitir la categoría adecuada. De esta manera, también resulta sencillo aumentar el número de palabras clave sin grandes modificaciones de la MDD.

Se nos plantea ahora el problema de comprobar de manera eficiente si la palabra encontrada efectivamente es una palabra clave. Para esto hay diversas opciones:

- Hacer busqueda dicotómica sobre una lista ordenada.
- Utilizar una tabla de dispersión (*hash*). Como la lista de palabras claves es conocida a priori, podemos utilizar una función de dispersión perfecta.
- Utilizar un *trie*.
- Etc...

6.5. El interfaz del analizador léxico

¿Cómo será la comunicación del analizador léxico con el sintáctico? Como ya comentamos en el tema de estructura del compilador, el analizador sintáctico irá pidiendo componentes léxicos a medida que los necesite. Así pues, lo mínimo que deberá ofrecer el analizador léxico es una función (o método en caso de que utilicemos un objeto para construirlo) que permita ir recuperando el componente léxico actual. Puede ser cómodo dividir esta lectura en dos partes:

- Una función que indique cuál es el último componente leído. Esta función no avanza la lectura.
- Una función que avance la lectura hasta encontrar otro componente y lo devuelva.

Dado que la primera es generalmente trivial (bastará con devolver el contenido de alguna variable local o de un atributo), veremos cómo implementar la segunda.

Además de estas funciones, existen una serie de funciones auxiliares que debe proporcionar el analizador:

- Una función (o método) para especificar cuál será el flujo de caracteres de entrada. Generalmente, esta función permitirá especificar el fichero desde el que leeremos. En caso de que el analizador sea un objeto, la especificación del flujo de entrada puede ser uno de los parámetros del constructor.
- Una función (o método) para averiguar en qué línea del fichero nos encontramos. Esto facilita la emisión de mensajes de error. Para dar mejores mensajes, puede también devolver el carácter dentro de la línea.
- Relacionada con la anterior, en algunos analizadores podemos pedir que escriba la línea actual con alguna marca para señalar el error.
- Si el analizador léxico necesita información del sintáctico, puede ser necesario utilizar funciones para comunicarla.

La complejidad o necesidad de estas funciones variará según las necesidades del lenguaje. Así, si nuestro lenguaje permite la inclusión de ficheros (como las directivas `#include` de C), será necesario poder cambiar con facilidad el flujo de entrada (o, alternativamente, crear más de un analizador léxico).

© Universitat Jaume I 2008-2009

6.6. El flujo de entrada

Hasta ahora hemos asumido más o menos implícitamente que los caracteres que utilizaba el analizador léxico provenían de algún fichero. En realidad, esto no es necesariamente así. Puede suceder que provengan de la entrada estándar[5], o de una cadena (por ejemplo, en caso de que estemos procesando los argumentos del programa).

Para evitar entrar en estos detalles, podemos suponer que existe un módulo u objeto que utiliza el analizador léxico para obtener los caracteres. Este módulo deberá ofrecer al menos los siguientes servicios:

- Un método para especificar desde dónde se leen los caracteres (fichero, cadena, dispositivo...).
- Un método para leer el siguiente carácter.
- Un método para devolver uno o más caracteres a la entrada.

Según las características del lenguaje para el que estemos escribiendo el compilador, la devolución de caracteres será más o menos complicada. En su forma más simple, bastará una variable para guardar un carácter. En otros casos, será necesario incluir alguna estructura de datos, que será una pila o una cola, para poder guardar más de un carácter. A la hora de leer un carácter, lo primero que habrá que hacer es comprobar si esta variable está o no vacía y después devolver el carácter correspondiente.

Un aspecto importante es la eficiencia de la lectura desde disco. No es raro que una implementación eficiente de las siguientes fases del compilador haga que la lectura de disco sea el factor determinante para el tiempo total de compilación. Por ello es necesario que se lean los caracteres utilizando algún almacén intermedio o *buffer*. Si se utiliza un lenguaje de programación de alto nivel, es normal que las propias funciones de lectura tengan ya implementado un sistema de *buffer*. Sin embargo, suele ser necesario ajustar alguno de sus parámetros. Aprovechando que los ordenadores actuales tienen memoria suficiente, la mejor estrategia puede ser leer completamente el fichero en memoria y tratarlo como una gran cadena. En algunos casos esto no es posible. Por ejemplo, si se está preparando un sistema interactivo o si se pretende poder utilizar el programa como filtro.

6.7. Implementación de la MDD

Llegamos ahora a la implementación de la MDD propiamente dicha. Podemos dividir las implementaciones en dos grandes bloques:

- Implementación mediante tablas.
- Implementación mediante control de flujo.

En el primer caso, se diseña una función que es independiente de la MDD que se esté utilizando. Esta se codifica en una tabla que se pasa como parámetro a la función. Esta estrategia tiene la ventaja de su flexibilidad: la función sirve para cualquier analizador y las modificaciones sólo afectan a la tabla. A cambio tiene el problema de la velocidad, que suele ser inferior a la otra alternativa.

La implementación mediante control de flujo genera un código que simula directamente el funcionamiento del autómata. Esta aproximación es menos flexible pero suele ser más eficiente.

6.7.1. Implementación mediante tablas

Esta es la implementación más directa de la MDD. Se utilizan tres tablas:

- La tabla *movimiento* contiene por cada estado y cada símbolo de entrada el estado siguiente o una marca especial en caso de que el estado no esté definido.

[5]En UNIX esto no supone diferencia ya que la entrada estándar se trata como un fichero más, pero en otros sistemas sí que es importante.

- La tabla *final* contiene por cada estado un booleano que indica si es o no final.
- La tabla *acción* contiene por cada estado las acciones asociadas. Asumiremos que la acción devuelve el token que se tiene que emitir o el valor especial *indefinido* para indicar que hay que omitir.

Debemos tener en cuenta que la tabla más grande, `movimiento`, será realmente grande. Con que tengamos algunos cientos de estados y que consideremos 256 carácteres distintos es fácil subir a más de cien mil entradas. Afortunadamente, es una tabla bastante dispersa ya que desde muchos estados no podemos transitar con muchos caracteres. Además existen muchas filas repetidas. Por esto existen diversas técnicas que reducen el tamaño de la tabla a niveles aceptables, aunque no las estudiaremos aquí.

Lo que tenemos que hacer para avanzar el analizador es entrar en un bucle que vaya leyendo los caracteres y actualizando el estado actual según marquen las tablas. A medida que avanzamos tomamos nota de los estados finales por los que pasamos. Cuando no podemos seguir transitando, devolvemos al flujo de entrada la parte del lexema que leímos después de pasar por el último final y ejecutamos las acciones pertinentes.

Las variables que utilizaremos son:

- q: el estado de la MDD.
- l: el lexema del componente actual.
- uf: último estado final por el que hemos pasado.
- ul: lexema que teníamos al llegar a uf.

Date cuenta de que al efectuar las acciones semánticas, debemos utilizar como lexema ul.

Teniendo en cuenta estas ideas, nos queda el algoritmo de la figura 3. Observa que hemos utilizado la palabra devolver con dos significados distintos: por un lado es la función que nos permite devolver al flujo de entrada el o los caracteres de anticipación; por otro lado, es la sentencia que termina la ejecución del algoritmo. Debería estar claro por el contexto cuál es cual.

A la hora de implementar el algoritmo, hay una serie de simplificaciones que se pueden hacer. Por ejemplo, si el flujo de entrada es “inteligente”, puede que baste decirle cuántos caracteres hay que devolver, sin pasarle toda la cadena.

La manera de tratar los errores dependerá de la estrategia exacta que hayamos decidido. Puede bastar con activar un *flag* de error, mostrar el error por la salida estándar y relanzar el análisis. Otra posibilidad es lanzar una excepción y confiar en que otro nivel la capture.

6.7.2. Implementación mediante control de flujo

En este caso, el propio programa refleja la estructura de la MDD. Lógicamente, aquí sólo podemos dar un esquema. La estructura global del programa es simplemente un bucle que lee un carácter y ejecuta el código asociado al estado actual. Esta estructura la puedes ver en la figura 4.

El código asociado a un estado dependerá de si este es o no final. En caso de que lo sea, tenemos que guardárnoslo en la variable uf y actualizar ul. Después tenemos que comprobar las transiciones y, si no hay transiciones posibles, devolver c al flujo de entrada y ejecutar las acciones asociadas al estado. Si las acciones terminan en omitir, hay que devolver la máquina al estado inicial, en caso de que haya que emitir, saldremos de la rutina. El código para los estados finales está en la figura 5.

El código para los estados no finales es ligeramente más complicado, porque si no hay transiciones debemos retroceder hasta el último final. El código correspondiente está en la figura 6.

Ten en cuenta que, si se está implementando a mano el analizador, suele ser muy fácil encontrar mejoras sencillas que aumentarán la eficiencia del código. Por ejemplo, muchas veces hay estados que transitan hacia sí mismos, lo que se puede modelar con un bucle más interno. En otros casos, si se lleva cuidado, no es necesario guardar el último estado final porque sólo habrá una posibilidad cuando no podamos transitar. Por todo esto, es bueno que después (o antes) de aplicar las reglas mecánicamente, reflexiones acerca de las particularidades de tu autómata y cómo puedes aprovecharlas.

© Universitat Jaume I 2008-2009

```
Algoritmo siguiente; // Con tablas
    q := q0; // Estado actual
    l := λ; // Lexema
    uf := indefinido; // Último estado final
    ul := λ; // Lexema leído hasta el último final
    mientras 0 = 1 − 1 hacer
        c := siguiente_carácter;
        l := l · c;
        si movimiento[q, c] ≠ indefinido entonces
            q := movimiento[q, c];
            si final[q] entonces uf := q; ul := l fin si
        si no
            si uf ≠ indefinido entonces
                devolver(ul⁻¹l); // Retrocedemos en el flujo de entrada
                t := ejecutar(acción[uf]);
                si t = indefinido entonces // Omitir
                    q := q0; l := λ; // Reiniciamos la MDD
                    uf := indefinido; ul := λ;
                si no
                    devolver t;
                fin si
            si no
                // Tratar error
            fin si
        fin si
    fin mientras
fin siguiente.
```

Figura 3: Algoritmo para encontrar el siguiente componente utilizando tablas. La expresión $ul^{-1}l$ representa la parte de l que queda después de quitarle del principio la cadena ul.

```
Algoritmo siguiente; // Con control de flujo
    q := q0; // Estado actual
    l := λ; // Lexema
    uf := indefinido; // Último estado final
    ul := λ; // Lexema leído hasta el último final
    mientras 0 = 1 − 1 hacer
        c := siguiente_carácter;
        l := l · c;
        opción q
            q0: // código del estado q0
            q1: // código del estado q1
            ...
            qn: // código del estado qn
        fin opción
    fin mientras
fin siguiente.
```

Figura 4: Estructura de un analizador léxico implementado mediante control de flujo.

```
// Código del estado q_i (final)
uf := q_i;
ul := lc^{-1}; // Ojo: eliminamos el último carácter
opción c
    a_1 ... a_n: q := q_a; // δ(q, c) = q_a para c ∈ a_1 ... a_n
    b_1 ... b_m: q := q_b; // δ(q, c) = q_b para c ∈ b_1 ... b_m
    ...
    z_1 ... z_k: q := q_z; // δ(q, c) = q_z para c ∈ z_1 ... z_k
si no // no hay transiciones con c
    devolver(c);
    // Acciones de q_i
    // Si las acciones incluyen omitir:
    q := q_0; l := λ;
    uf := indefinido; ul := λ;
    // Si no incluyen omitir:
    devolver t;
fin opción
```

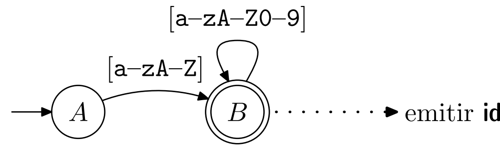
Figura 5: Código asociado a los estados finales.

```
// Código del estado q_i (no final)
opción c
    a_1 ... a_n: q := q_a; // δ(q, c) = q_a para c ∈ a_1 ... a_n
    b_1 ... b_m: q := q_b; // δ(q, c) = q_b para c ∈ b_1 ... b_m
    ...
    z_1 ... z_k: q := q_z; // δ(q, c) = q_z para c ∈ z_1 ... z_k
si no // no hay transiciones con c
    si uf ≠ indefinido entonces
        devolver(ul^{-1}l); // Retrocedemos en el flujo de entrada
        t := ejecutar(acción[uf]);
        si t = indefinido entonces
            q := q_0; l := λ;
            uf := indefinido; ul := λ;
        si no
            devolver t;
        fin si
    si no
        // Tratar error
    fin si
fin opción
```

Figura 6: Código asociado a los estados no finales.

© Universitat Jaume I 2008-2009

Bucles en los estados Supongamos que nuestro autómata tiene un fragmento como este:



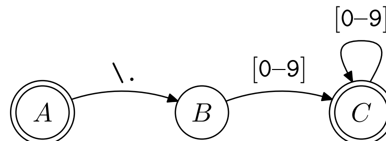
Las acciones correspondientes al estado B serían:

$uf := B$;
$ul := lc^{-1}$;
opción c
  a...z, A...Z, 0...9: $q := B$;
si no
  devolver(c);
  $t :=$ **id**(ul);
  devolver t;
fin opción

Sin embargo, es más eficiente hacer:

mientras c **en** [a...z, A...Z, 0...9] **hacer**
  $l := l \cdot c$;
  $c :=$ siguiente_carácter;
fin mientras
devolver(c);
devolver **id**(l);

No se guarda el estado final Si nuestro autómata tiene un fragmento como:



no es necesario en el estado A guardar el último estado final por el que se pasa: en B, sabemos que, si no se puede transitar, se ha venido de A y en C ya estamos en un estado final.

7. Introducción a un generador automático de analizadores léxicos: flex

`Flex` (***F**ast **Lex**ical Analyzer Generator*) es un generador automático de analizadores léxicos para el lenguaje de programación C: lee un fichero de texto con una descripción del analizador léxico y produce un programa C que implementa el analizador. Este analizador está especialmente diseñado para usarse conjuntamente con `bison` (`yacc`), un metacompilador.

`Flex` es software GNU creado a partir de `lex`, un generador de analizadores léxicos muy popularizado porque se distribuye gratuitamente con las versiones más comunes de Unix (es una utilidad estándar desde la séptima edición de Unix). `Lex` fue creado por Mike Lesk y Eric Schmidt de Bell Labs (AT&T) en los 70. El autor original de `flex` fue Jef Poskanzer, la versión actual es de Vern Paxson.

Usualmente, invocando a `lex` en una distribución estándar de Linux estaremos ejecutando a `flex`: el grado de compatibilidad entre ambos programas es muy alto.

Cuadro 2: Símbolos disponibles para las expresiones regulares en `flex`.

Símbolo	Significado
`.`	Cualquier carácter excepto la nueva línea (`\n`).
`*`	Cero o más copias de la expresión a su izquierda.
`[]`	Clase de caracteres. Se aplican las mismas reglas que hemos utilizado en este tema.
`+`	Una o más ocurrencias de la expresión a su izquierda.
`?`	Cero o una ocurrencia de la expresión a su izquierda.
`|`	Disyunción.
`()`	Altera el orden de aplicación de operadores y agrupa subexpresiones para poder referirse a ellas más adelante.
`{}`	Indica un rango de repeticiones para una expresión regular. Por ejemplo `a{1,3}` indica una, dos o tres apariciones del carácter `a`.
`^`	En una clase de caracteres, indica negación. Fuera de una clase de caracteres, describe el principio de una línea.
`$`	Final de línea.
`\`	"Escapa" metacaracteres.
`"..."`	Interpreta literalmente los caracteres entre comillas. Ejemplo: `"*+"` es un asterisco seguido de una suma.
`/`	Operador de anticipación. Acepta la expresión de la izquierda sólo si va seguida de la de la derecha. Por ejemplo, `([0-9][0-9])/am` reconoce dos dígitos si y sólo si van seguidos de una `a` y una `m`, pero estos caracteres no serán parte del lexema detectado.

7.1. Estructura de un programa flex

Una especificación (o programa) `flex` es un fichero (usualmente con la extensión `.l`) compuesto por tres partes separadas por `%%`:

Declaraciones
`%%`
Reglas
`%%`
Procedimientos auxiliares

7.1.1. La zona de reglas

Cada regla es de la forma

expresión regular {*acción*}

donde *acción* es un fragmento de código en C (con algunas extensiones que comentaremos más adelante). Las expresiones regulares pueden utilizar los metasímbolos que se muestran en el cuadro 2.

Los analizadores generados por `flex` funcionan de manera similar a los que hemos estudiado en el tema. Leen de la entrada el prefijo más largo que pertenece a la expresión regular de alguna regla. Entonces ejecutan la acción asociada a esa regla. Si la regla incluye una sentencia `return`, el control es devuelto a la función que invocó el analizador. En caso contrario, se busca el siguiente componente. Si la parte de la acción de una regla está vacía, la correspondiente entrada es simplemente desechada. La acción por defecto para aquellas partes de la entrada que no se pueden analizar es mostrarlas por la salida estándar.

Veamos un ejemplo de programa `flex`. En un fichero (`ejemplo.l`) escribimos:

```
%%
[ \t\n]+                { /*no hacer nada */ }
fin                     { printf("Adios.\n"); return; }
[a-zA-Z]([a-zA-Z0-9])*  { printf("Identificador %s\n", yytext); }
```

```
"+"|"-"|"*"|"/"           { printf("Operador %c\n", yytext[0]); }
[0-9]+                    { printf("Entero %d\n", atoi(yytext)); }
[0-9]+\.[0-9]+            { printf("Real %f\n", atof(yytext)); }
%%
```

El fichero describe un analizador léxico que detecta blancos (y los filtra), la palabra `fin`, identificadores y números (enteros y reales).

Aunque las expresiones regulares `fin` y `[a-zA-Z]([a-zA-Z0-9])*` no definen lenguajes disjuntos, no hay conflicto: la expresión que aparece primero tiene prioridad.

Para cada componente detectado (excepto los blancos), muestra por pantalla la categoría léxica y cierta información adicional: para identificadores, el lexema; y para los números, el valor (entero o real en función del tipo). La variable predefinida `yytext` es una cadena que contiene siempre el lexema del último componente detectado. Es un puntero a `char` que sigue la convención C de marcar el final con un carácter 0. Además, la variable `yyleng` contiene su longitud.

Ya hemos comentado que las partes de la entrada no aceptadas por ninguna de las expresiones son escritas en la salida estándar. Para tratar esta situación, es habitual añadir al final una regla con el punto como expresión regular:

```
%%
[ \t\n]+              { /*no hacer nada */ }
fin                   { printf("Adios.\n"); return; }
[a-zA-Z]([a-zA-Z0-9])* { printf("Identificador %s\n", yytext); }
"+"|"-"|"*"|"/"       { printf("Operador %c\n", yytext[0]); }
[0-9]+                { printf("Entero %d\n", atoi(yytext)); }
[0-9]+\.[0-9]+        { printf("Real %f\n", atof(yytext)); }
.                     { printf("Carácter no esperado %c\n", yytext[0]); }
%%
```

Si compilamos el analizador y tecleamos, por ejemplo, `1+bb10/yy a fin`, obtendríamos por pantalla lo siguiente:

```
Entero 1
Operador +
Identificador bb10
Operador /
Identificador yy
Identificador a
Adios.
```

7.1.2. La zona de funciones auxiliares

La zona de funciones auxiliares permite incluir código C que se copia literalmente en `lex.yy.c`. Si, por ejemplo, definimos una función `main`, crearemos un programa autónomo:

```
%%
[ \t\n]+                  { /*no hacer nada */ }
fin                       { printf("Adios.\n"); return; }
[a-zA-Z]([a-zA-Z0-9])*    { printf("Identificador %s\n", yytext); }
"+"|"-"|"*"|"/"           { printf("Operador %c\n", yytext[0]); }
[0-9]+                    { printf("Entero %d\n", atoi(yytext)); }
[0-9]+\.[0-9]+            { printf("Real %f\n", atof(yytext)); }
.                         { printf("Carácter no esperado %c\n", yytext[0]); }
%%
int main(int argc, char ** argv)
{
  yylex();
  exit(0);
}
```

7.1.3. La zona de declaraciones

En la zona de declaraciones podemos dar nombres a expresiones regulares. Estos nombres pueden usarse luego en la zona de reglas encerrando su identificador entre llaves.

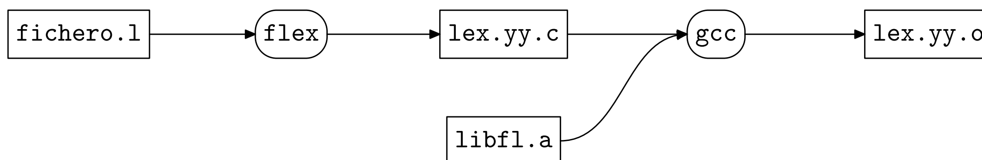
```
car       [a-zA-Z]
dig       [0-9]
%%
[ \t\n]+                 { /*no hacer nada */ }
fin                      { printf("Adios.\n"); return; }
{car}({car}|{dig})*      { printf("Identificador %s\n", yytext); }
"+"|"-"|"*"|"/"          { printf("Operador %c\n", yytext[0]); }
{dig}+                   { printf("Entero %d\n", atoi(yytext)); }
{dig}+\.{dig}+           { printf("Real %f\n", atof(yytext)); }
.                        { printf("Carácter no esperado %c\n", yytext[0]); }
%%
int main(int argc, char ** argv)
{
  yylex();
  exit(0);
}
```

Podemos declarar otros elementos en la zona de declaraciones:

- estados (que no veremos aquí),
- definiciones de variables o estructuras C(encerradas entre `%{` y `%}`) que usamos en las acciones asociadas a las reglas. Estas definiciones se copian literalmente al principio del fichero `lex.yy.c`.

7.2. Compilación de las especificaciones

Las especificaciones se pueden emplear para generar un fichero C, que puede actuar como módulo de un programa mayor o compilarse para dar directamente un programa ejecutable.



Una vez hemos escrito la especificación, debemos crear un fichero C que sea capaz de ejecutarla. La orden para crear el fichero es:

```
$ flex ejemplo.l
```

con lo que obtendremos un fichero `lex.yy.c`. Podemos compilar este fichero con

```
$ gcc lex.yy.c -c
```

El resultado es un fichero `lex.yy.o` que contiene una función `yylex()` que implementa el analizador léxico descrito. Si invocamos esta función desde un programa nuestro, el analizador leerá de la entrada estándar reconociendo componentes hasta encontrar alguno cuya acción asociada tenga `return`, momento en que nos devolverá el control.

Si nuestro programa ha sido compilado en el módulo `otro.o`, la orden para obtener el programa final es

```
$ gcc otro.o lex.yy.o -lfl
```

La opción `-lfl` le indica al enlazador que utilice la biblioteca de `flex`.

Si hemos utilizado la zona de funciones auxiliares para definir `main` y queremos crear un programa autónomo, podemos utilizar la secuencia:

```
$ flex ejemplo.l
$ gcc lex.yy.c -lfl
```

Lógicamente, podemos utilizar cualesquiera otras opciones del compilador para optimizar, cambiar el nombre del ejecutable o lo que nos interese.

7.3. Otro ejemplo

El siguiente programa `flex` cuenta el número de caracteres, palabras y líneas que se proporcionen por la entrada estándar.

```
%{
unsigned c = 0, p = 0, l = 0;
%}
palabra [^ \t\n]+
eol     \n
%%
{palabra}    { p++; c+=yyleng; }
{eol}        { c++; l++; }
.            { c++; }
%%
main() {
  yylex();
  printf("%d %d %d\n", c, p, l);
}
```

Las variables `c`, `p` y `l` cuentan, respectivamente, el número de caracteres, palabras y líneas.

7.4. Cómo utilizar diferentes flujos de entrada

¿Y si queremos que se lea la entrada de un fichero y no de `stdin`? Basta con redefinir la función `main` y, dentro de ella, asignar a `yyin` el fichero en cuestión. La variable `yyin` es un *stream* del que `yylex` toma la entrada.

```
...
%%
int main(int argc, char * argv[]) {
  FILE * f;
  if (argc > 1) {
    f = fopen(argv[1], "r");
    if (!f) { fprintf(stderr,"Error abriendo %s\n", argv[1]); exit(1); }
    yyin = f;
  }  /* Si argc=1, yyin vale stdin */
  yylex();
  printf("%d %d %d\n", c, p, l);
  return 0;
}
```

Otras posibles opciones para la lectura son:

- **Lectura de varios ficheros.** Cuando `yylex()` alcanza el final de fichero, llama automáticamente a la función `yywrap()`, que devuelve 0 o 1. Si devuelve un 1, el programa ha finalizado. Si devuelve un 0, el analizador asume que `yywrap()` ha asignado a `yyin` un nuevo fichero para lectura, con lo que sigue leyendo. De este modo se pueden procesar varios ficheros en una única tanda de análisis.

- **Lectura de una cadena.** Es posible analizar una cadena en lugar de un fichero. Para esto, se puede redefinir la macro `YY_INPUT` o utilizar las funciones `yy_scan_string`, `yy_scan_bytes` o `yy_scan_buffer`.

8. Algunas aplicaciones de los analizadores léxicos

Además de para construir compiladores e intérpretes, los analizadores léxicos se pueden emplear para muchos programas "convencionales". Los ejemplos más claros son aquellos programas que tienen algún tipo de entrada de texto donde hay un formato razonablemente libre en cuanto espacios y comentarios. En estos casos es bastante engorroso controlar dónde empieza y termina cada componente y es fácil liarse con los punteros a `char`. Un analizador léxico simplifica notablemente la interfaz y si se dispone de un generador automático, el problema se resuelve en pocas líneas de código.

8.1. Ficheros organizados por columnas

Un ejemplo sencillo de uso de analizadores léxicos sería el trabajo con ficheros organizados en forma de columnas separadas por tabuladores. Supongamos que tenemos un fichero de texto en el que aparecen secuencias de dígitos separadas por tabuladores que definen columnas. Queremos sumar la columna n de la tabla.

Podemos emplear la siguiente especificación léxica:

categoría	expresión regular	acciones	atributos
numero	$[0–9]^+$	calcular valor emitir	valor
separación	\t	emitir	
linea	\n	emitir	
blanco	$␣^+$	omitir	

Pasar esto a `flex` es trivial:

```
%{
#include "columnas.h"
#include <stdio.h>
#include <string.h>
int nlinea;
int yyval;
%}
%%

" "+ ; /* Eliminamos los espacios. */
[0-9]+ { yyval= atoi(yytext); return NUMERO; }
\t { return SEPARACION; }
\n { nlinea++; return LINEA; }
. { fprintf(stderr, "Aviso: carácter %c inesperado, línea %d.\n", yytext[0], nlinea); }
```

Las definiciones auxiliares necesarias estarían en el fichero `columnas.h`:

```
#define NUMERO 256
#define SEPARACION 257
#define LINEA 258

extern int yyval;
extern int nlinea;
int yylex();
```

```
int suma (int ncol) /* Suma los datos de la columna ncol. */
  {
    int token, col, res;

    res= 0; /* resultado */
    nlinea= 0;
    while ( (token= yylex())!= 0)
      {
        col= 1; /* columna actual */
        while (1)
          {
            if ( token!= NUMERO )
              error("La línea debe comenzar por un número");
            if ( col== ncol )
              res+= yyval;
            token= yylex();
            if ( token!= SEPARACION && token!= LINEA )
              error ("Detrás de un número debe haber un tabulador o un fin de línea.");
            if ( token== LINEA )
              break;
            token= yylex();
            col++;
          }
        if ( col< ncol )
          error ("No hay suficientes campos en la línea.");
      }
    return res;
  }
```

Figura 7: Función para sumar los datos de una columna.

Algunos comentarios: hemos utilizado valores por encima de 255 para no mezclar categorías y caracteres (esto es un convenio debido al interfaz con `bison` y es bueno seguirlo); declaramos `nlinea` e `yyval` para poder acceder a ellos en el programa principal. El método que tiene `flex` para pasar atributos no es muy elegante (y también es debido a `bison`), por eso usamos `yyval` para el valor del número.

La función que suma una columna determinada la puedes ver en la figura 7. La función `error` es simplemente:

```
void error (char *mensaje)
  {
    fprintf(stderr, "Error en línea %d: %s\n", nlinea, mensaje);
    exit(1);
  } // error
```

8.2. Procesamiento de ficheros de configuración

Otro ejemplo típico de uso de analizadores léxicos sería la lectura de un fichero de configuración. Supongamos que en este fichero se tienen líneas con el formato: *variable* = *valor*. Los nombres de variable están son secuencias de letras y dígitos que comienzan por una letra. Los valores pueden ser cadenas entre comillas (sin comillas en su interior) o números enteros. Se permiten comentarios que comienzan por el carácter `#` y terminan al final de la línea. Además se permiten líneas vacías.

También ahora, la especificación léxica es sencilla:

categoría	expresión regular	acciones	atributos
variable	[a–zA–Z][a–zA–Z0–9]*	copiar lexema emitir	lexema
igual	=	emitir	
valor	[0–9][0–9]*|"[^"\n]*"	copiar lexema emitir	lexema
linea	(#[^\n]*)?\n	emitir	
blanco	[␣\t]$^+$	omitir	

Como antes, codificar el analizador con `flex` es fácil:

```
%{
#include "config.h"
#include <stdio.h>
int nlinea;
%}
%%

[ \t]+ ;
[a-zA-Z][a-zA-Z0-9]* { return VARIABLE; }
= { return IGUAL; }
[0-9][0-9]*|\"[^"\n]*\" { return VALOR; }
(#.*)?\n { nlinea++; return LINEA; }
. { fprintf(stderr, "Aviso: carácter %c inesperado, línea %d.\n", yytext[0], nlinea); }
```

Los identificadores `VARIABLE`, `IGUAL`, `VALOR` y `LINEA` son constantes definidas en `config.h`:

```
#define VARIABLE 256
#define IGUAL 257
#define VALOR 258
#define LINEA 259

extern int nlinea;
extern char *yytext;
int yylex();
```

El procesamiento del fichero es ahora bastante sencillo, como puedes ver en la figura 8.

9. Resumen del tema

- El analizador léxico divide la entrada en componentes léxicos.
- Los componentes se agrupan en categorías léxicas.
- Asociamos atributos a las categorías léxicas.
- Especificamos las categorías mediante expresiones regulares.
- Para reconocer los lenguajes asociados a las expresiones regulares empleamos autómatas de estados finitos.
- Se pueden construir los AFD directamente a partir de la expresión regular.
- El analizador léxico utiliza la máquina discriminadora determinista.
- El tratamiento de errores en el nivel léxico es muy simple.
- Podemos implementar la MDD de dos formas: con tablas, mediante control de flujo.
- Hay generadores automáticos de analizadores léxicos, por ejemplo `flex`.
- Se pueden emplear las ideas de los analizadores léxicos para facilitar el tratamiento de ficheros de texto.

© Universitat Jaume I 2008-2009

```
void procesaConfiguracion()
  {
    int token;
    char *var= 0, *valor= 0;

    nlinea= 1;
    while ( (token= yylex())!= 0 )
      {
        if ( token== LINEA )
          continue;

        if ( token!= VARIABLE )
          error("la línea debe empezar por un nombre de variable.");
        free(var);
        var= strdup(yytext);

        token= yylex();
        if ( token!= IGUAL )
          error("la línea debe tener un igual tras la variable.");

        token= yylex();
        if ( token!= VALOR )
          error("la línea debe tener un valor tras el igual.");
        free(valor);
        valor= strdup(yytext);

        token= yylex();
        if ( token!= LINEA )
          error ("la línea debe terminar en fin de línea.");

        asignar(var, valor);
      }
    free(var); free(valor);
  } // procesaConfiguracion
```

Figura 8: Función para procesar ficheros de configuración.