Introducción

En primer lugar, veremos un algoritmo que nos permite encontrar una expresión regular que describe el lenguaje de una autómata indeterminista sin transiciones con la palabra vacía.

Después de eso, estudiaremos una de las aplicaciones básicas de la teoría de autómatas, como es el diseño de programas eficientes de búsqueda.

A continuación, mostraremos cómo se puede diseñar el analizador léxico de un compilador.

Por último, empezaremos a estudiar el tipo de autómatas que se utilizan en el diseño del analizador sintáctico de un compilador.

La equivalencia viene dada por el siguiente teorema.

Teorema

Para todo lenguaje L, existe un autómata indeterminista M tal que $L(M) = L$ si y sólo si existe una expresión regular α tal que $L = L(\alpha)$.

Este teorema expresa entonces la equivalencia entre el concepto de expresión regular y el concepto de autómata indeterminista.

Observación

Obsérvese que el concepto de autómata determinista es un caso particular del concepto de autómata indeterminista, ya que si $M = (K, \Sigma, \delta, q_0, F)$ es un autómata determinista, se tiene que δ es una función de $K \times \Sigma$ en K, y por tanto δ es un subconjunto de $K \times \Sigma \times K$, y por consiguiente δ es un subconjunto de $K \times (\Sigma \cup \{\lambda\} \times K$.

A continuación, vamos a ver un algoritmo que nos permite encontrar una expresión regular que describe el lenguaje de una autómata indeterminista que no tenga transiciones con la palabra vacía. Por tanto, este algoritmo sirve para encontrar una expresión regular que describe el lenguaje de una autómata determinista. Si el autómata indeterminista tiene transiciones con la palabra vacía, deberemos transformar previamente dicho autómata en un autómata determinista equivalente, utilizando el algoritmo que vimos en la última clase.

El Lema de Arden

El algoritmo que veremos está basado en el llamado Lema de Arden. Para poder enunciar dicho lema, necesitamos definir el siguiente concepto.

Una ecuación lineal sobre un alfabeto Σ es una expresión

$$X = AX \cup B$$

donde A y B son lenguajes sobre el alfabeto Σ y X es una incógnita.

Lema de Arden. (a) A^*B es solución de la ecuación $X = AX \cup B$.

(b) Si $\lambda \notin A$, entonces A^*B es la única solución de la ecuación $X = AX \cup B$.

En el algoritmo que vamos a ver ahora se utiliza la parte (a) del Lema de Arden.

Algoritmo para encontrar una expresión regular asociada a un autómata

La entrada del algoritmo es un autómata indeterminista $M = (K, \Sigma, \Delta, q_0, F)$ sin transiciones con la palabra vacía. Y la salida es una expresión regular α tal que $L(\alpha) = L(M)$.

Sean $q_0, \ldots, q_n$ los estados de K, donde q_0 es el estado inicial. Para todo $i \leq n$ definimos

$$X_i = L(q_i) = \{x \in \Sigma^* : \exists q \in F(q_i x \vdash_M^* q)\}.$$

Por tanto, $L(M) = X_0$.

Algoritmo para encontrar una expresión regular asociada a un autómata

Entonces, para todo estado $q_i \in F$ ponemos la ecuación

$$X_i = \bigcup \{aX_j : j \leq n, (q_i, a, q_j) \in \Delta\} \cup \lambda$$

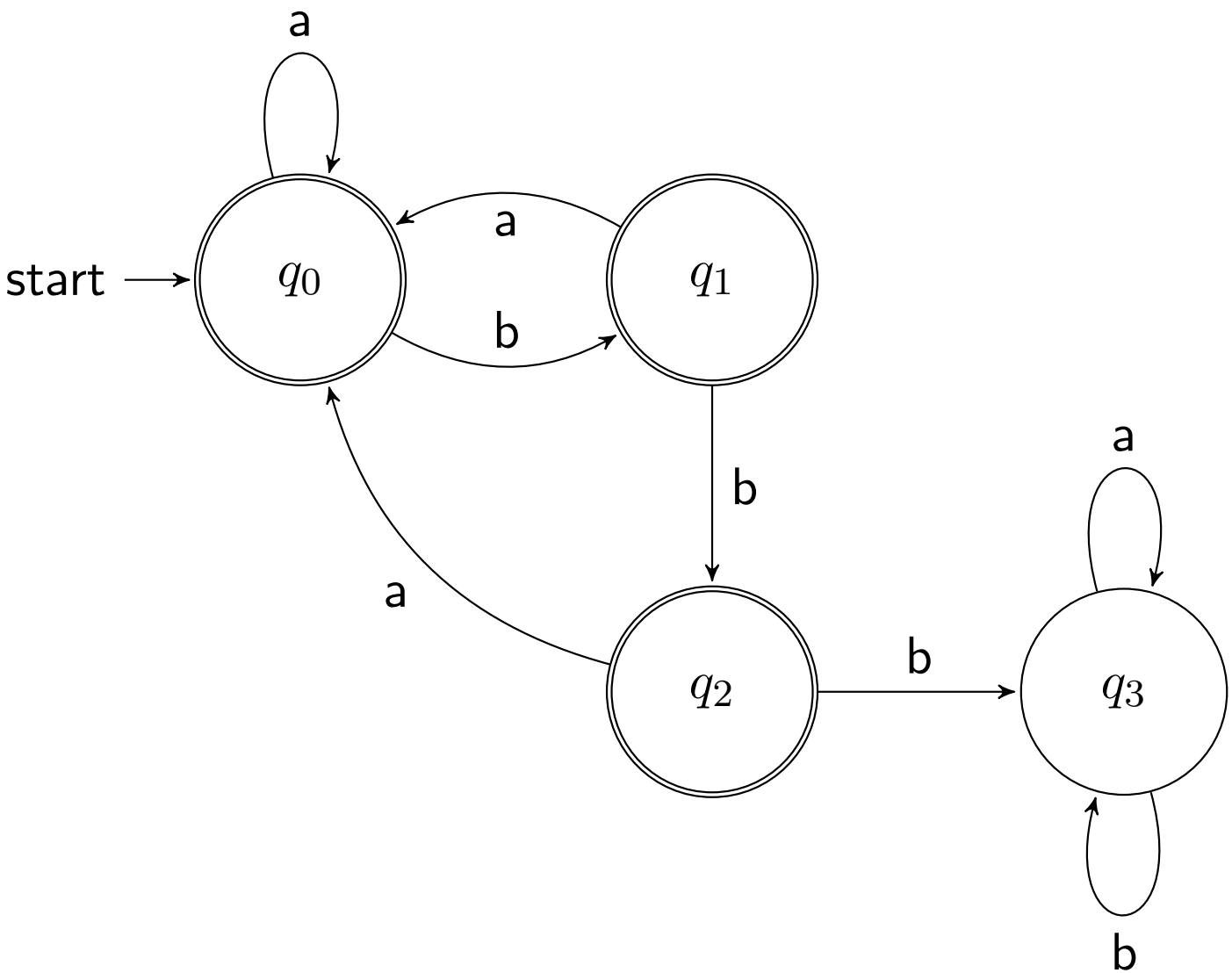
y para todo estado $q_i \notin F$ ponemos la ecuación

$$X_i = \bigcup \{aX_j : j \leq n, (q_i, a, q_j) \in \Delta\}$$

El algoritmo consiste entonces en resolver el sistema de ecuaciones, utilizando la parte (a) del Lema de Arden.

Ejemplo

Consideremos el autómata determinista M', que vimos anteriormente:



Ejemplo

Tenemos las siguientes ecuaciones asociadas al autómata M:

(1) $X_0 = aX_0 \cup bX_1 \cup \lambda$.

(2) $X_1 = aX_0 \cup bX_2 \cup \lambda$.

(3) $X_2 = aX_0 \cup bX_3 \cup \lambda = aX_0 \cup \lambda$.

(4) $X_3 = aX_3 \cup bX_3$.

Obsérvese que en la ecuación para X_2, tenemos que $aX_0 \cup bX_3 \cup \lambda = aX_0 \cup \lambda$, porque $X_3 = L(q_3) = \emptyset$, y por tanto $bX_3 = \emptyset$.

Ejemplo

Sustituyendo (3) en (2), obtenemos:

$(5) X_1 = aX_0 \cup baX_0 \cup b \cup \lambda = (a \cup ba)X_0 \cup b \cup \lambda.$

Ahora, sustituyendo (5) en (1), obtenemos:

$(6) X_0 = aX_0 \cup b(a \cup ba)X_0 \cup bb \cup b \cup \lambda = (a \cup b(a \cup ba))X_0 \cup bb \cup b \cup \lambda.$

Aplicando entonces el Lema de Arden a (6), obtenemos:

$(7) X_0 = (a \cup b(a \cup ba))^* \cdot (bb \cup b \cup \lambda).$

Por tanto, $L(M) = X_0 = L(\alpha)$ donde

$\alpha = (a \cup b(a \cup ba))^* \cdot (bb \cup b \cup \lambda).$

Búsqueda de un patrón en un texto

Estudiamos una aplicación básica de los autómatas, y es su uso para diseñar programas eficientes de búsqueda.

Consideremos el problema de buscar en un texto una secuencia de caracteres, a la cual denominamos "patrón". Tenemos entonces un patrón p y un texto t, que suponemos largo en relación al patrón, y queremos buscar el patrón p en el texto t. Un programa en JAVA que hace esta función sería el siguiente.

```
int buscar (String texto, String patron int n, int m)
//n y m son las longitudes del texto y patrón respectivamente
{ int i =0,j; boolean trobat = false;
  while ((i <= n-m) && (trobat == false)
        { j=0;
        while(texto.charAt(i+j)== patron.charAt(j) && j < m -1)
                                                    j = j+1;
        if (texto.charAt(i+j) == patron.charAt(j) ) trobat = true;
        i = i + 1;}
return trobat;}
```

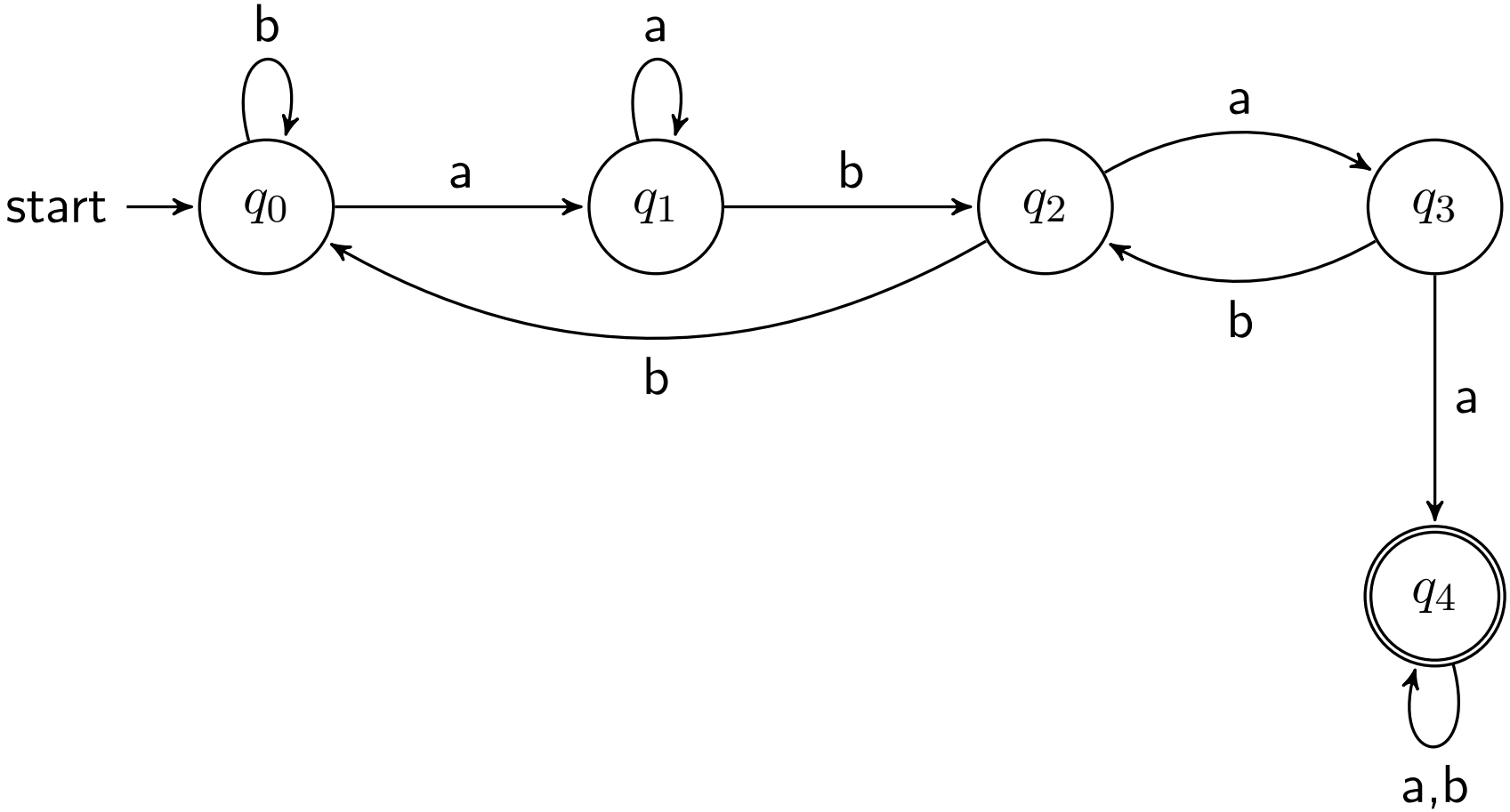
Búsqueda de un patrón en un texto

Observamos que el programa se sitúa en una posición del texto, y entonces compara los símbolos del patrón con los del texto a partir de la posición considerada. Si se produce una discrepancia, el programa se sitúa en la siguiente posición del texto y vuelve a comparar los caracteres del patrón con los del texto. Si se encuentra el patrón, el programa asigna a la variable trobat el valor true y sale del bucle "while" externo.

Búsqueda de un patrón en un texto

Sin embargo, este programa se puede mejorar, ya que vemos que no saca partido de la información que va recogiendo a medida que explora el texto. Para poner un ejemplo de este hecho, supongamos que trabajamos con el alfabeto $\Sigma = \{a, b\}$, que el patrón es la palabra $abaa$ y que el texto tiene la forma $ababaaabba$..... Entonces, el programa anterior se sitúa inicialmente en la primera posición del texto, y compara $abaa$ con $abab$, Como no son iguales, el programa pasa a la segunda posición del texto y compara $abaa$ con $baba$. Sin embargo, esto es ineficiente, ya que al comparar el patrón $abaa$ con $abab$, tenemos que las posiciones tercera y cuarta del texto coinciden con las dos primeras posiciones del patrón. Por tanto, un programa eficiente debe situarse en la quinta posición del texto y no en la segunda. Para conseguir hacer esto, hay que considerar un autómata determinista que reconozca el patrón $abaa$, y a continuación programar dicho autómata utilizando el algoritmo que ya conocemos. El autómata determinista que reconoce el patrón $abaa$ es el siguiente:

Búsqueda de un patrón en un texto



Búsqueda de un patrón en un texto

Se tiene entonces que si n es la longitud del texto, el tiempo de ejecución del programa asociado al anterior autómata determinista está en $O(n)$, es decir, tiene complejidad lineal, por lo que es un programa que mejora sustancialmente al primer programa que mostramos.

Es posible incluso, dados un texto y un patrón, escribir un programa que en primer lugar construye el autómata determinista que reconoce el patrón y, a continuación, simula el programa asociado al autómata que ha construido.

Fases en el diseño de un compilador

El diseño de compiladores es la principal aplicación de la teoría de autómatas. Un compilador es un programa interno del ordenador, que recibe como entrada un programa escrito en un lenguaje de programación de alto nivel (como Java o C), y entonces determina si el programa está correctamente escrito, y si es así traduce dicho programa a código máquina. Por tanto, los compiladores crean archivos ejecutables cuyo código entiende el ordenador. Los compiladores son esenciales en programación, ya que sin ellos no sería posible programar en lenguajes de alto nivel.

En el diseño de un compilador, se distinguen las tres siguientes fases:

(1) Análisis léxico.

(2) Análisis sintáctico.

(3) Análisis semántico.

Fases en el diseño de un compilador

El programa-fuente, es decir, el programa escrito en lenguaje de alto nivel que se va a traducir a código máquina, se almacena en un fichero de caracteres, al que se le llama fichero de entrada. El analizador léxico agrupa entonces los símbolos que va leyendo del fichero de entrada en categorías sintácticas, es decir, identifica las categorías sintácticas de los símbolos que va leyendo, y a continuación pasa esa información al analizador sintáctico. Las categorías sintácticas (o tokens) de un lenguaje de programación son la clase de los identificadores, cada tipo básico de datos (tipo integer, tipo float, tipo char, etc), cada símbolo relacional, cada operador aritmético, cada símbolo de puntuación y cada palabra reservada.

Recordemos que los identificadores son los nombres que damos a los tipos, literales, variables, clases, métodos, paquetes y sentencias de un programa.

En el lenguaje C, un identificador es una palabra formada por letras, dígitos y el carácter de subrayado, de manera que el primer carácter de la palabra no es un dígito.

Fases en el diseño de un compilador

En la segunda fase del diseño del compilador, en la fase de análisis sintáctico, se determina si el programa está escrito correctamente según las reglas del lenguaje de programación. El analizador sintáctico de un compilador trabaja siempre con las categorías sintácticas de los símbolos del programa que le suministra el analizador léxico. Y esto es así, porque sería mucho más complicado diseñar un analizador sintáctico que trabajase con los símbolos del programa en lugar de con las categorías. La utilización de las categorías sintácticas simplifica, por tanto, el diseño del analizador sintáctico.

Si no se detectan errores en la fase del análisis sintáctico, se pasa entonces a la fase del análisis semántico. En dicha fase, se controlan los aspectos semánticos como la correspondencia de tipos en las asignaciones o en los parámetros de llamadas a funciones, y si no hay tales errores semánticos se procede a efectuar la traducción.

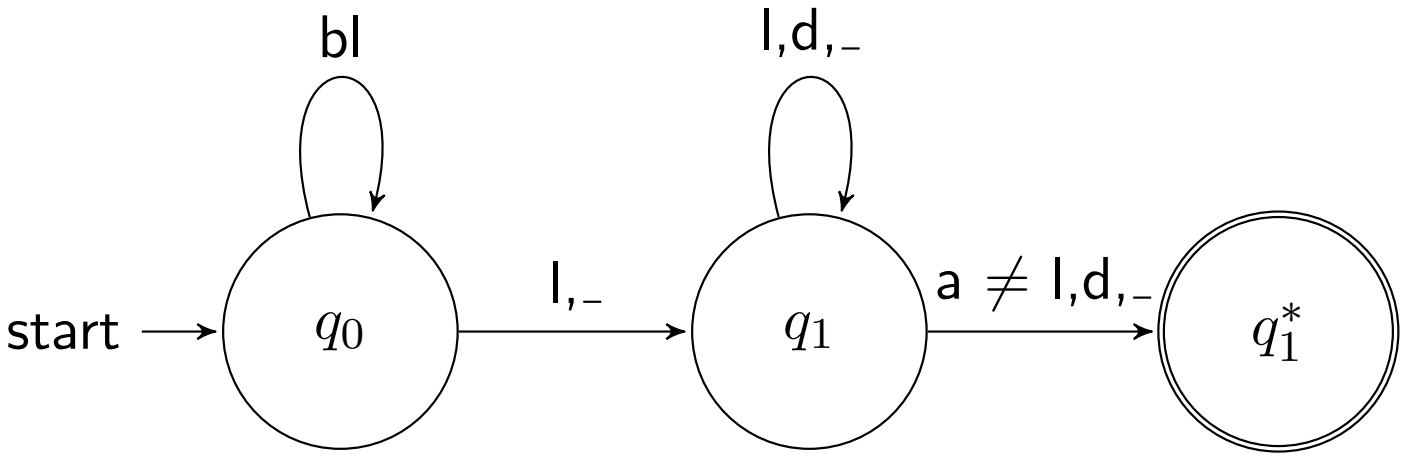
Diseño de analizadores léxicos

Para diseñar el analizador léxico de un compilador, se ha de definir un autómata indeterminista que reconozca cada una de las categorías sintácticas del lenguaje de programación que estemos considerando.

Para simplificar la construcción, denotamos por l a cualquier letra y por d a cualquier dígito. Y denotamos por bl al carácter blanco.

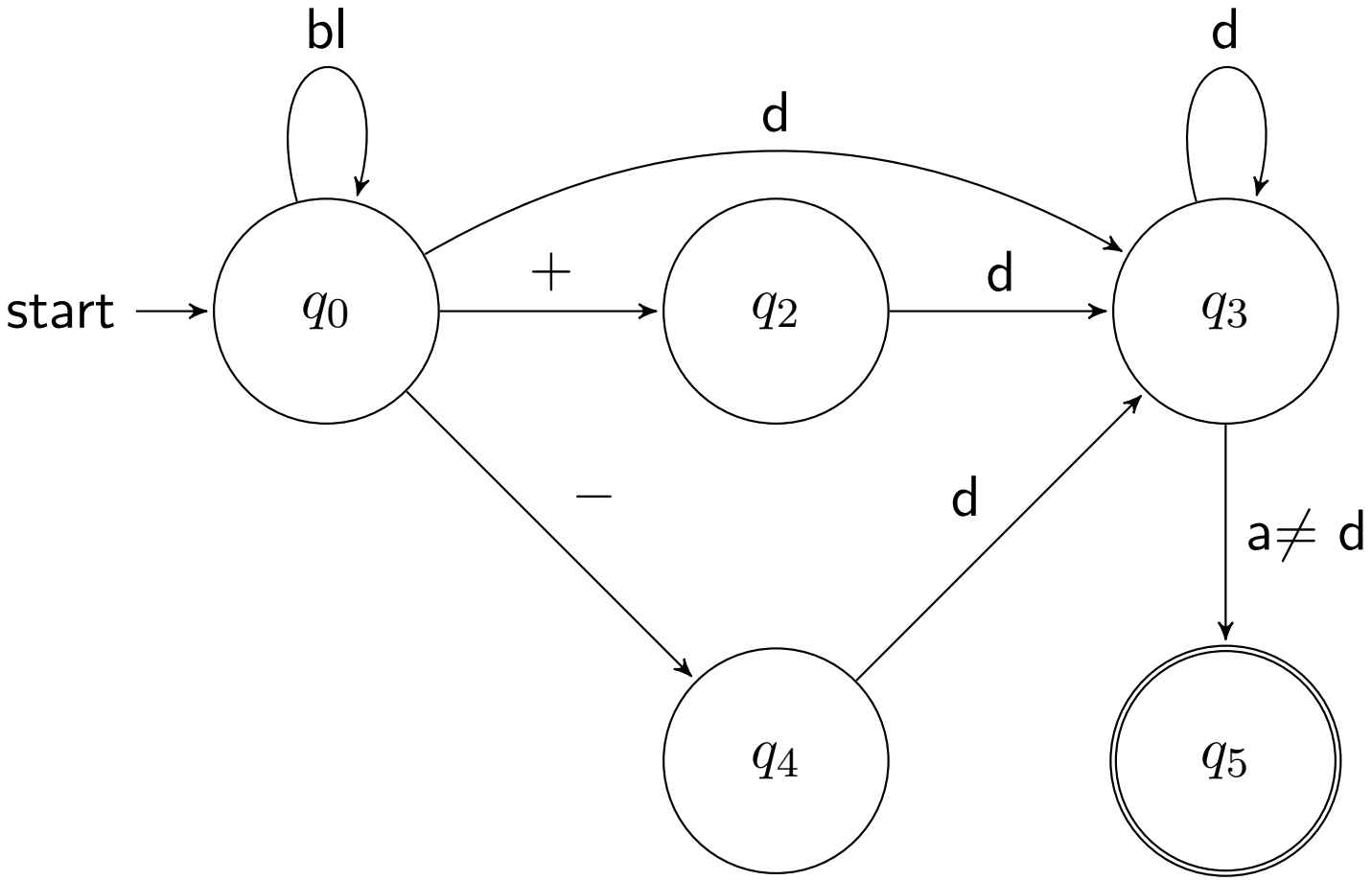
Diseño de analizadores léxicos

Tenemos el siguiente autómata para reconocer los identificadores de C:



Diseño de analizadores léxicos

Para el tipo “integer”, tenemos el siguiente autómata:

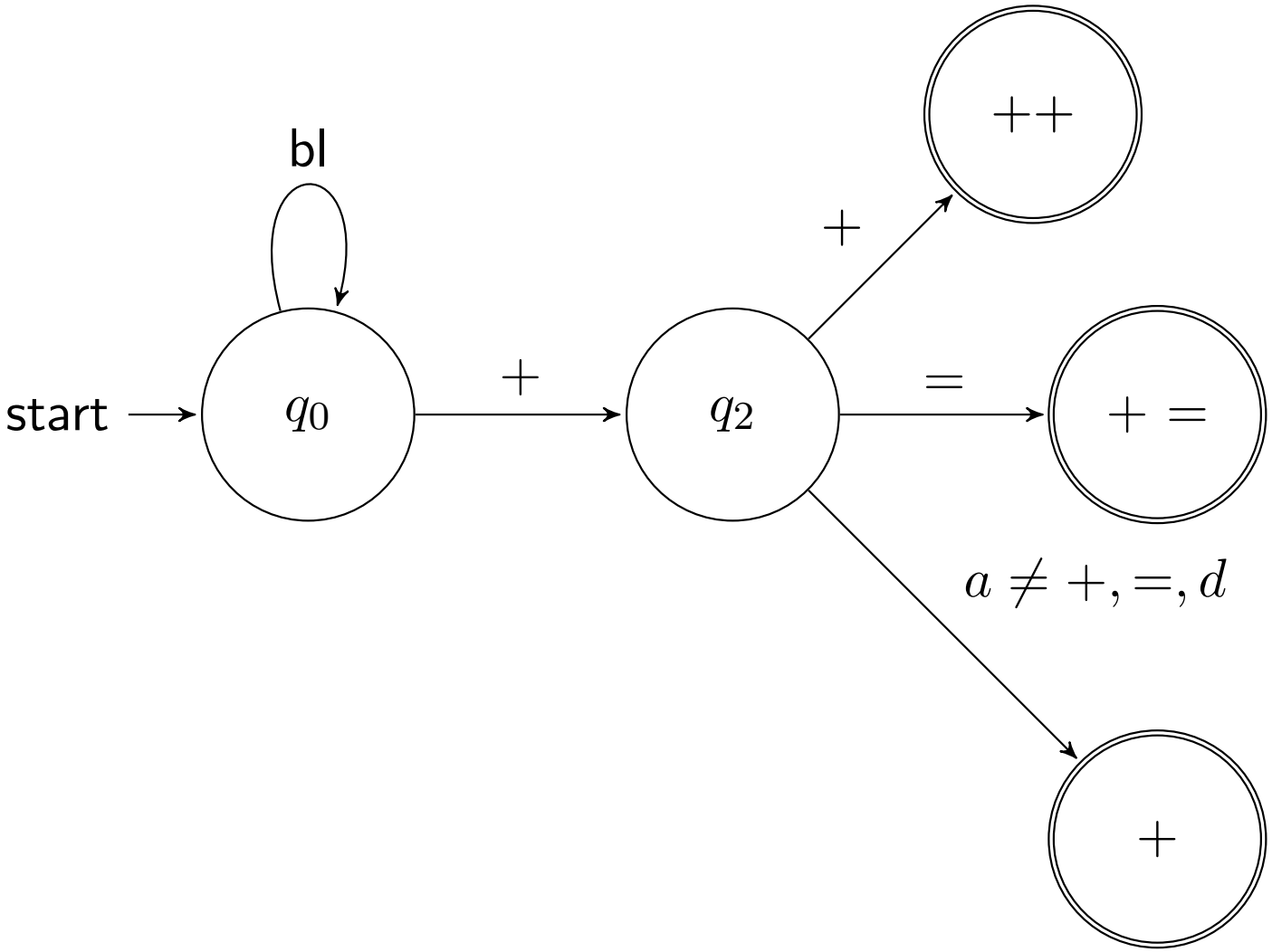


Diseño de analizadores léxicos

Ahora, a partir de los estados q_2 y q_4, podemos reconocer las categorías sintácticas $+$, $-$, $++$, $--$, $+=$ y $-=$.

En concreto, para reconocer las categorías sintácticas $+$, $+=$ y $++$, tenemos el siguiente autómata:

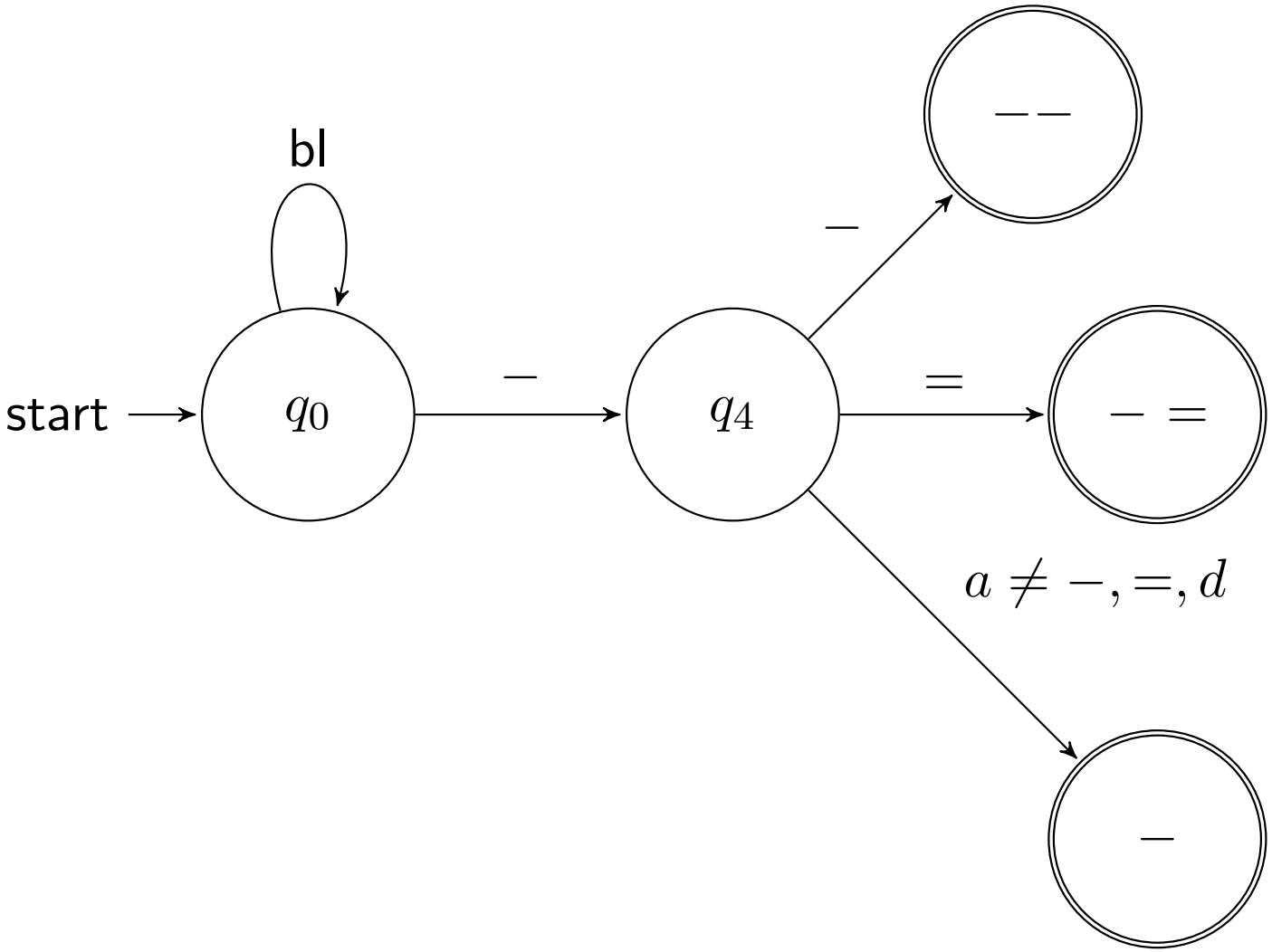
Diseño de analizadores léxicos



Diseño de analizadores léxicos

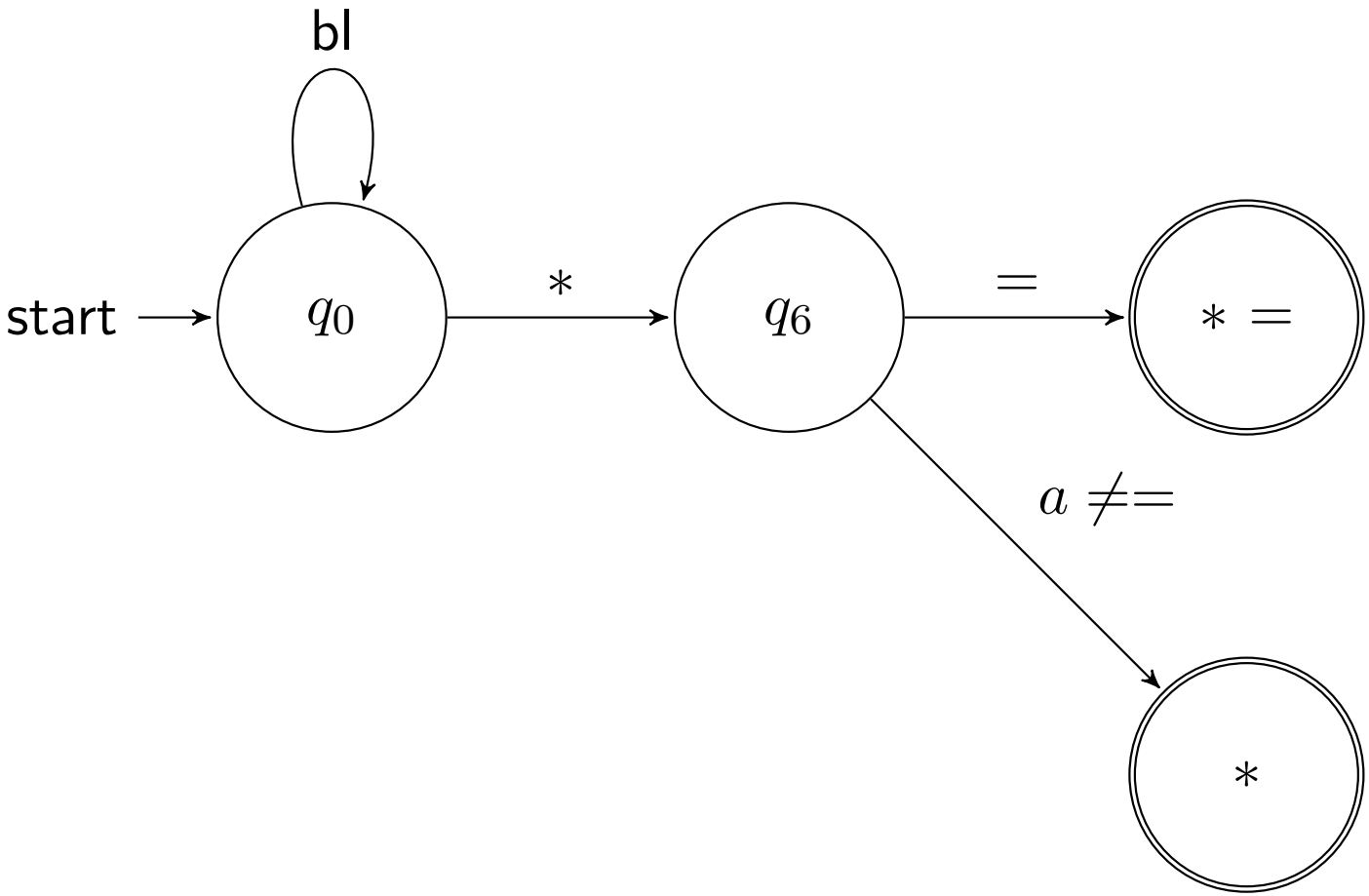
Y para reconocer las categorías sintácticas $-$, $- =$ y $--$, tenemos el siguiente autómata:

Diseño de analizadores léxicos



Diseño de analizadores léxicos

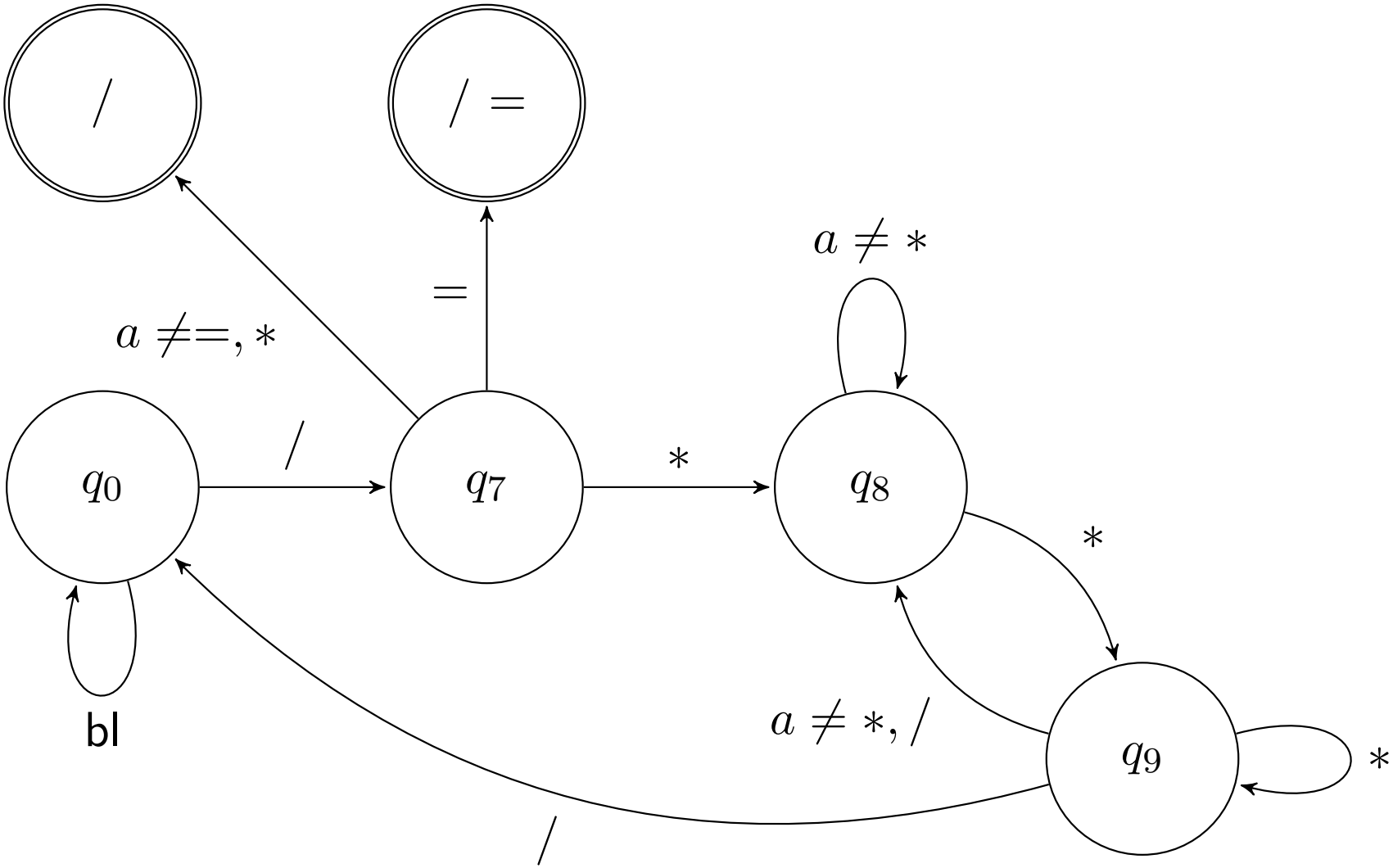
Ahora, para reconocer las categorías sintácticas $*$ y $* =$, tenemos el siguiente autómata:



Diseño de analizadores léxicos

Si entra el símbolo /, es posible que entre el símbolo para la división o la categoría sintáctica $/=$ o es posible que tengamos el comienzo de un comentario. El analizador léxico debe saltar los comentarios, ya que éstos no tienen interés para el análisis sintáctico. En el siguiente autómata, eliminamos entonces los comentarios que comienzan por los símbolos $/*$ y terminan con los símbolos $*/$. Análogamente, se eliminarían los comentarios de una sola línea.

Diseño de analizadores léxicos



De esta forma, vamos reconociendo en el autómata las categorías sintácticas del lenguaje de programación. A continuación, transformamos este autómata indeterminista en un autómata determinista M equivalente, añadiendo un estado de error, es decir, añadiendo un estado no aceptador, del cual nunca saldremos y al cual accederemos desde cualquier otro estado cuando entre un símbolo que no interesa. Por tanto, los estados del autómata determinista M serán los mismos estados del autómata indeterminista más el estado de error.

En dicho autómata M, se observa que hay categorías sintácticas que son reconocidas al leer el último símbolo de la categoría, como es el caso de las categorías $++$ y $--$. Sin embargo, hay otras categorías sintácticas que son reconocidas al leer el símbolo siguiente de la categoría, como es el caso de los identificadores y del tipo entero, y de los operadores $+$ y $-$. Se dice entonces que estos estados aceptadores van adelantados un carácter, ya que reconocen la categoría cuando leen el símbolo siguiente.

Al escribir posteriormente el programa asociado al autómata, cuando lleguemos a un estado aceptador que vaya adelantado un carácter, no deberemos leer el siguiente símbolo de la entrada, porque dicho símbolo ya está leído. Para programar entonces el analizador léxico, debemos programar el autómata determinista M con las siguientes modificaciones:

Diseño de analizadores léxicos

(1) Cuando se llegue a un estado aceptador, se ha de imprimir la categoría sintáctica reconocida y se ha de volver al estado inicial.

(2) Cuando se llegue a un estado aceptador que vaya adelantado un carácter, no se ha de leer el siguiente carácter de la entrada.

(3) Cuando se llegue al estado q_1^*, se ha de explorar la tabla de las palabras reservadas para determinar si la palabra leída está en la tabla. Si no lo está, se da como salida "identificador". Y si lo está, se da como salida la palabra reservada.

Bibliografía para el tema 3

- Teoría de autómatas y lenguajes formales (D. Kelley)

  Capítulos 1 y 2.
- Compiladores: teoría e implementación (J. Ruiz Catalán)

  Capítulos 1 y 2.