Capítulo 1

Análisis léxico

Existe un lenguaje que va más allá de las palabras.

Paulo Coelho

Resumen: En este tema se explican las tareas del analizador léxico de un compilador, se recuerdan las nociones de expresiones regulares y de autómatas finitos necesarias para especificar y reconocer las unidades léxicas de un lenguaje, y se desarrollan técnicas de implementación de Analizadores Léxicos. Se pretende que el alumno sepa especificar los aspectos léxicos de un lenguaje informático, e implementar sistemáticamente el analizador léxico a partir de dicha especificación.

1.1. Introducción

- ⋆ El analizador léxico es la única parte del compilador que procesa **uno a uno** los caracteres del programa fuente.
- ⋆ Su tarea consiste en **agrupar** dichos caracteres en unidades más grandes llamadas **unidades léxicas** (*tokens*) que constituyen el **lenguaje de entrada** para el analizador sintáctico.
- ⋆ Puede considerarse una función de **secuencias en secuencias**:

$$Analizador\_lexico : Char^* \rightarrow Token^*$$

- ⋆ Las unidades léxicas (ULs) se agrupan en un número finito de **clases**. Ejemplos de clases de ULs:
  - identificadores
  - palabras reservadas (una clase por cada palabra)

- constantes literales (una clase por tipo de datos)
- símbolos (una clase por símbolo)
- operador infijo (e.g `<=`, `+`, ...)

⋆ Una **palabra reservada** es un identificador que tiene un significado predefinido en el lenguaje de programación, independientemente del lugar de la fuente en que aparezca (e.g. **do**). Ningún otro identificador puede tener ese nombre. Existen identificadores con un significado predefinido dentro de un contexto (e.g. *integer*), pero en otro contexto podría usarse ese nombre.

⋆ Los lenguajes formales utilizados para describir las unidades léxicas son los **lenguajes regulares**, cuyos reconocedores son los **autómatas finitos**.

⋆ Asociados a una UL, el analizador devuelve al menos su **clase** y cero o más **atributos**. La clase es utilizada por el a. sintáctico para discriminar entre ULs. Los atributos sólo serán necesarios en las siguientes fases del compilador. Ejemplos de atributos:

- nombre del identificador
- valor de la constante literal
- número de línea y de columna donde aparece la UL en la fuente

⋆ Para hacer eficiente el analizador sintáctico, debe haber una clase **distinta** por cada UL que sea tratada de forma distinta por las reglas gramáticales de la sintáxis (e.g. no tendría sentido agrupar las ULs correspondientes a las palabras reservadas **begin** y **end** en una sola clase).

⋆ El analizador puede procesar toda la fuente, o ser llamado por el a. sintáctico cada vez que se necesita la siguiente UL. En cualquier caso, el análisis de una UL empieza en el primer carácter que no formaba parte de la UL precedente.

⋆ El a. léxico **no** tiene información sobre la **estructura** de la frase. Esto es competencia del a. sintáctico (e.g. "**do do do**" sería aceptado por el a. léxico).

⋆ Otras tareas del a. léxico (o de su **discriminador** asociado) son:

- Convertir las constantes literales a su representación interna (e.g. $\langle$**LitInt**, 124$\rangle$ en lugar de $\langle$**LitInt**,"124"$\rangle$).
- Representar cada identificador por una clave numérica única.
- Eliminar los comentarios y los blancos redundantes una vez analizados.
- Detectar errores léxicos.

1.2. Características léxicas de los lenguajes

⋆ Es importante conocer el **juego de caracteres** y su **codificación** usados por nuestra instalación:

ASCII 7-bits, no soporta caracteres no ingleses.

ISO/IEC 8859 8-bits, los primeros 128 códigos coinciden con ASCII, los siguientes 128 son variables dependiendo de la codificación (p.e. ISO/IEC 8859-1, llamado Latin-1, codifica caracteres acentuados y nuestra ñ).

EBCDIC 8-bits, usado por antiguas máquinas IBM, obsoleto.

Unicode Soporta hasta 109.000 caracteres distintos en 93 grafías. Codificaciones a 8-bits mediante UTF-8, o a 16-bits mediante UTF-16.

ISO/IEC 10646 Define un *Universal Character Set* con espacio para más de un millón de caracteres. Codificaciones a longitud fija con UCS-2 (16-bits), UCS-4 (32-bits), y a longitud variable con UTF-1 (1 a 5 octetos).

Las codificaciones de Unicode y de ISO/IEC 10646 respetan la codificación Latin-1 en sus primeros 256 códigos.

⋆ Para el compilador serán relevantes algunas **propiedades**: si los números y las letras tienen códigos consecutivos, la relación entre la codificación de una minúscula y la de la correspondiente mayúscula, la posición del carácter blanco, etc.

⋆ La mayoría de los lenguajes admiten **minúsculas y mayúsculas**:

- Algunos las permiten pero no las distinguen: Pascal, Ada
- Otros si (i.e. `If` es distinto de `if`): C
- Otros requieren las palabras reservadas solo en mayúscula (Modula-2) o sólo en minúscula (C)

⋆ Los **identificadores** suelen comenzar por letra y admitir después letras, números y algunos caracteres especiales como {`_, -, ', $`} y otros. La **longitud** actualmente no es una limitación.

⋆ Muchos lenguajes admiten **formato libre**, en el que el número de blancos, tabuladores, fin de línea, etc, excepto el primero de ellos, **no son significativos**.

⋆ Recientemente, algunos lenguajes usan el **sangrado** del texto con un significado **sintáctico**: Occam, Haskell, Python, ...

⋆ Los **comentarios** pueden ser, de **línea** (e.g `//...`), de **bloque** (e.g. `/* ...*/`), y dentro de estos últimos permitir o no **anidamiento**.

1.3. Expresiones regulares

⋆ Un poco de notación:

- $\Sigma = \{a, b, c, \ldots\}$ **alfabeto** finito de símbolos
- Σ^* conjunto de todas las cadenas de cero o más símbolos de Σ. Usaremos $x, y, \ldots$ para denotar elementos de Σ^*
- ϵ la cadena **vacía**
- xy concatenación de las cadenas x e y
- x^n cadena formada concatenando n veces la cadena x

⋆ Un **lenguaje** sobre Σ es cualquier subconjunto $L \subseteq \Sigma^*$.

$L_1 \cup L_2$	Unión de lenguajes
$L_1 L_2 = \{xy \mid x \in L_1, y \in L_2\}$	Concatenación de lenguajes
$\overline{L} = \Sigma^* - L$	Lenguaje complementario
$\overline{A} = \Sigma - A \quad$ (con $A \subseteq \Sigma$)	Alfabeto complementario
$L^n = \{x_1 \cdots x_n \mid x_i \in L, 1 \leq i \leq n\}$	
$L^* = \bigcup_{n \geq 0} L^n$	Cierre de un lenguaje

⋆ Definimos inductivamente el conjunto de **expresiones regulares** sobre Σ, denotadas $r, s, \ldots$:

1. $\emptyset$, denota el **lenguaje vacío**
2. ϵ, denota el lenguaje no vacío $\{\epsilon\}$
3. a, con $a \in \Sigma$, denota el lenguaje $\{a\}$
4. Si r_1 y r_2 describen respectivamente los lenguajes L_1 y L_2:
   - $r_1 \mid r_2$ es una expresión regular que describe el lenguaje $L_1 \cup L_2$
   - $r_1 r_2$ es una expresión regular que describe el lenguaje $L_1 L_2$
   - r_1^* es una expresión regular que describe el lenguaje L_1^*

Para ahorrar paréntesis la precedencia será: * >> concatenación >> |

⋆ Llamamos **lenguaje regular** a todo lenguaje que puede ser definido por una expresión regular.

⋆ Ejemplos de expresiones regulares sobre $\Sigma = \{0, 1\}$:

- $(0 \mid 1)^*$ equivale a Σ^*
- $(01)^*$ cadenas de ceros y unos estrictamente alternados empezando en cero e incluyendo la cadena vacía.
- 01^* cadenas que consisten en un cero seguido de cualquier número de 1's, incluido ninguno.

- $(0 \mid 1)^*00(0 \mid 1)^*$ Cadenas de ceros y unos que incluyen al menos dos ceros consecutivos.

⋆ Abreviaturas:

- r^+ abrevia rr^*
- $[r]$ abrevia $r \mid \epsilon$

⋆ Si L es un lenguaje regular, $\overline{L}$ también es regular (no lo demostramos). Llamaremos $L(r)$ al lenguaje definido por la expresión regular r. Cuando r denote un subconjunto $A \subseteq \Sigma$, entonces escribiremos $\overline{r}$ como abreviatura del subconjunto $\overline{A} = \Sigma - A$.

⋆ El lenguaje de las **unidades léxicas** se puede definir mediante expresiones regulares sobre $\Sigma = \{$juego de caracteres$\}$. Obsérvese que les damos nombres pero que no puede haber recursión mutua entre las definiciones (*definiciones regulares*):

$$
\begin{array}{lcl}
\textit{digito} & \longrightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\textit{letra} & \longrightarrow & a \mid b \mid \ldots \mid z \\
\textit{literalEntero} & \longrightarrow & [+|-]\ \textit{digito}^+ \\
\textit{id} & \longrightarrow & \textit{letra}\ (\textit{letra} \mid \textit{digito})^* \\
\textit{opRel} & \longrightarrow & <|<=|>|>=|==|\ /= \\
\textit{sep} & \longrightarrow & \text{SP} \mid \text{TAB} \mid \text{CR} \mid \text{NL} \\
\textit{seps} & \longrightarrow & \textit{sep}^+ \\
\textit{comentario} & \longrightarrow & //(\overline{CR})^* CR
\end{array}
$$

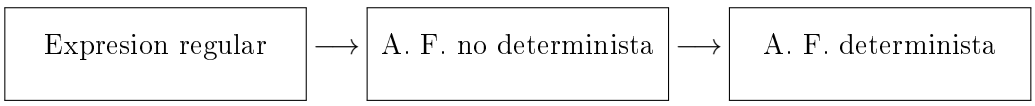
1.4. Autómatas finitos

⋆ Los autómatas finitos son los **reconocedores** de los lenguajes regulares. Dado un lenguaje regular siempre existe un autómata finito (normalmente más de uno) que lo reconoce, y viceversa.

⋆ En realidad, un mismo lenguaje regular queda determinado por varios formalismos equivalentes:

- una expresión regular
- un autómata finito no determinista con transiciones vacías
- un autómata finito determinista

⋆ Seguiremos el siguiente diagrama que permite pasar de unos a otros:



También es posible pasar de autómata finito determinista a expresión regular (no lo veremos).

1.4.1. Autómatas finitos no deterministas

⋆ Un **autómata finito no determinista** (AFN) es una tupla $M = (Q, \Sigma, \delta, q_0, F)$:

- Q un conjunto finito de **estados**
- $q_0 \in Q$ el **estado inicial**
- $F \subseteq Q$ conjunto de estados **finales**
- Σ un **alfabeto finito**
- $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ relación de **transición**

⋆ Relación de **evolución** entre configuraciones:

- (q, w) con $q \in Q, w \in \Sigma^*$ es una **configuración** de M.
- (q_0, w) es una **configuración inicial**.
- (q_f, ϵ) con $q_f \in F$ es una **configuración final**.
- $(q, aw) \vdash_M (p, w)$, con $a \in \Sigma \cup \{\epsilon\}$, (decimos que la primera configuración evoluciona a la segunda en un paso según M) sii $(q, a, p) \in \delta$.
- $(q, aw) \vdash_M^* (p, w)$, evolución en **cero o más** pasos, es el **cierre** reflexivo y transitivo de $\vdash_M$.

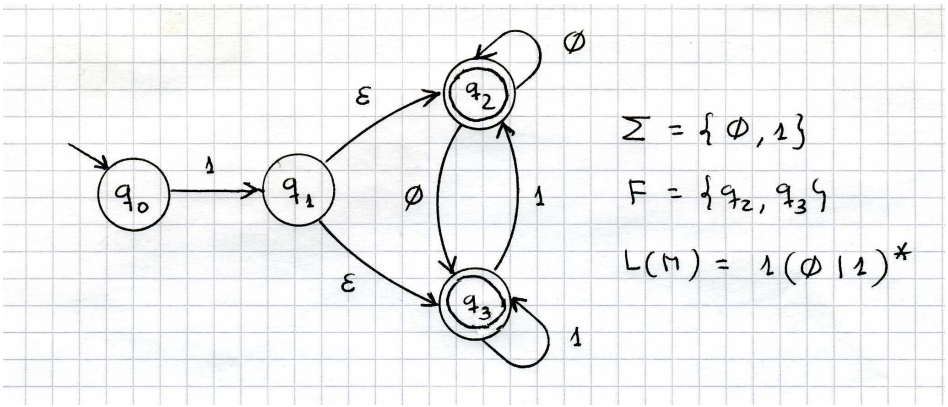
Como δ es una relación, fijados q y a, podrían existir distintos p_i tales que $(q, a, p_i) \in \delta$ para todo i. Ello expresa el **no determinismo** en la evolución del autómata.

⋆ **Lenguaje** reconocido por un AFN:

$$L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q_f, \epsilon),\ q_f \in F\}$$

Equivale a decir que **alguna** evolución (no necesariamente todas) del AFN conduce a un estado final tras haber consumido toda la cadena de entrada.

⋆ Usaremos frecuentemente la siguiente notación gráfica:



1.4.2. De expresión regular a AFN

⋆ Dada una expresión regular r, siempre existe un AFN M tal que $L(r) = L(M)$.

⋆ El autómata M se construye por inducción sobre la estructura de r. Exigiremos además que el autómata tenga **un** solo estado final.

Casos base Hay tres:

- Si $r = \epsilon$, entonces $M = (\{q_0, q_f\}, \Sigma, \{(q_0, \epsilon, q_f)\}, q_0, \{q_f\}))$
- Si $r = \emptyset$, entonces $M = (\{q_0, q_f\}, \Sigma, \emptyset, q_0, \{q_f\}))$
- Si $r = a$, entonces $M = (\{q_0, q_f\}, \Sigma, \{(q_0, a, q_f)\}, q_0, \{q_f\}))$

Casos inductivos Hay otros tres:

- Si $r = r_1 \mid r_2$, por h.i. existen dos AFNs M_1, M_2 que reconocen respectivamente los lenguajes de r_1 y r_2. Llamemos respectivamente q_1, q_2 y f_1, f_2 a los estados iniciales y finales de M_1, M_2. Entonces la construcción (a) de la Figura 1.1, nos da el autómata requerido.
- Si $r = r_1 r_2$, sean $M_1, M_2, q_1, q_2, f_1, f_2$ como en el apartado anterior. Entonces la construcción (b) de la Figura 1.1, nos da el autómata requerido.
- Si $r = r_1^*$, sea M_1 con q_1, f_1 el AFN asociado a r_1. Entonces la construcción (c) de la Figura 1.1, nos da el autómata requerido.

⋆ El siguiente AFN reconoce la expresión $r = 1(0 \mid 1)^*$ y ha sido construído aplicando dichas reglas:

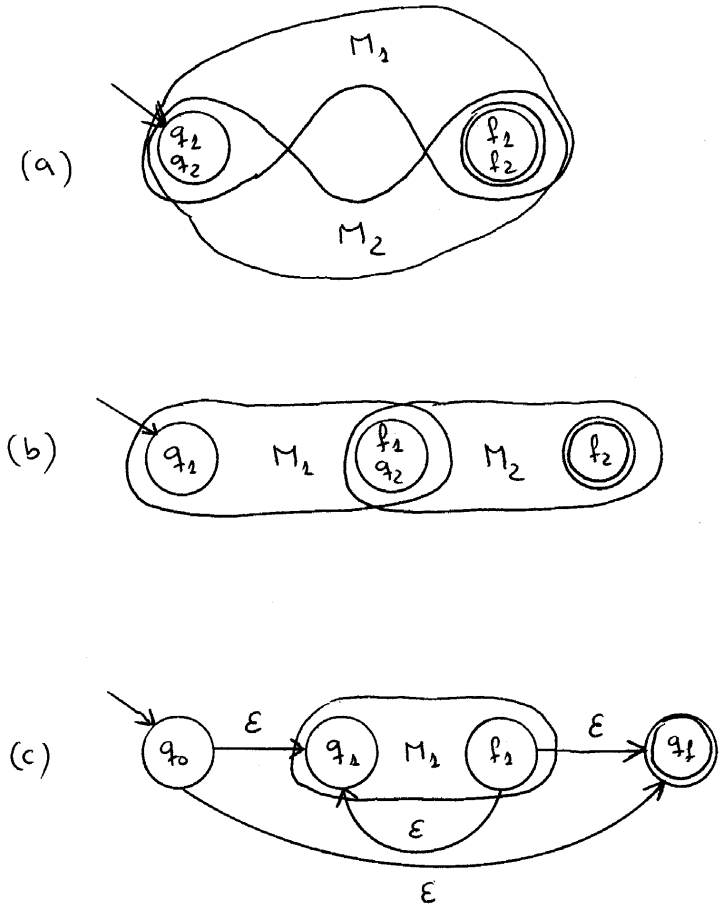
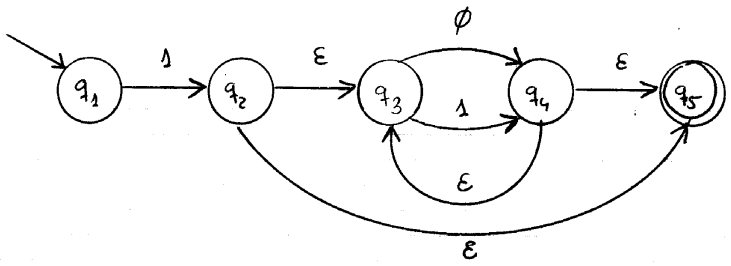


Figura 1.1: Reglas inductivas para construir el AFN asociado a una expresión regular



1.4.3. Autómatas finitos deterministas

⋆ Un **autómata finito determinista** (AFD) es un caso particular de autómata no determinista $M = (Q, \Sigma, \delta, q_0, F)$ en el que la relación de transición δ **no** incluye transiciones vacías ϵ y además es una **función parcial** $\delta : Q \times \Sigma \longrightarrow Q$.

⋆ Es decir, fijado un estado $q \in Q$ y un símbolo $a \in \Sigma$, o bien queda bloqueado, o bien evoluciona a un **único** estado $\delta(q, a)$.

⋆ Relación de **evolución** y **lenguaje** reconocido:

- (q, w) con $q \in Q, w \in \Sigma^*$ es una **configuración** de M.
- (q_0, w) es una **configuración inicial**.
- (q_f, ϵ) con $q_f \in F$ es una **configuración final**.
- $(q, aw) \vdash_M (p, w)$, con $a \in \Sigma$, y decimos que la primera configuración evoluciona a la segunda en un paso según M, sii $\delta(q, a) = p$.
- lenguaje reconocido por un AFD:

$$L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q_f, \epsilon),\ q_f \in F\}$$

1.4.4. De AFN a AFD

⋆ Dado un AFN M, siempre existe un AFD M' tal que $L(M) = L(M')$.

⋆ Para construirlo explícitamente necesitamos primero el concepto de $cierre{-}\epsilon$ de un estado $q \in Q$ de un AFN $M = (Q, \Sigma, \delta, q_0, F)$:

$$cierre{-}\epsilon(q) = \{p \mid (q, \epsilon) \vdash_M^* (p, \epsilon)\}$$

⋆ Aplicamos esta definición a los estados del AFN de la Sección 1.4.2:

$$
\begin{aligned}
&\mathit{cierre}{-}\epsilon(q_1) = \{q_1\} \\
&\mathit{cierre}{-}\epsilon(q_2) = \{q_2, q_3, q_5\} \\
&\mathit{cierre}{-}\epsilon(q_3) = \{q_3\} \\
&\mathit{cierre}{-}\epsilon(q_4) = \{q_4, q_3, q_5\} \\
&\mathit{cierre}{-}\epsilon(q_5) = \{q_5\}
\end{aligned}
$$

⋆ Extendemos la definición a conjuntos de estados $S \subseteq Q$:

$$\mathit{cierre}{-}\epsilon(S) = \bigcup_{q \in S} \mathit{cierre}{-}\epsilon(q)$$

⋆ La construcción del AFD M' se basa en que sus estados van a ser **subconjuntos** de los estados de M, es decir, $Q' \subseteq \mathcal{P}(Q)$. La idea intuitiva es que dos estados p, q de M estarán en el mismo estado de M' si existe una palabra w tal que $(q_0, w) \vdash_M^* (p, \epsilon)$ y $(q_0, w) \vdash_M^* (q, \epsilon)$. La Figura 1.2 muestra el algoritmo de construcción. En ella $\epsilon{-}SS$ debe traducirse por $\mathit{cierre}{-}\epsilon$.

⋆ El **estado inicial** de M' es $q'_0 = \mathit{cierre}{-}\epsilon(q_0)$. Los **estados finales** de M' son todos aquellos subconjuntos de Q que contengan al menos un estado final de M.

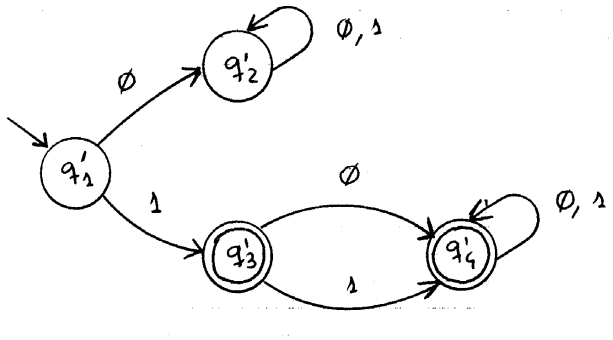
⋆ Si aplicamos el algoritmo al AFN de la Sección 1.4.2, obtenemos el siguiente AFD, donde $q'_1 = \{q_1\}$, $q'_2 = \{\}$, $q'_3 = \{q_2, q_3, q_5\}$, $q'_4 = \{q_3, q_4, q_5\}$:

Algorithm NFA → DFA
Input: NFA $M = (Q, \Sigma, \Delta, q_0, F)$.
Output: DFA $M' = (Q', \Sigma, \delta, q_0', F')$ according to Definition 7.8.

Method: States in M' are sets of states of M. The starting state of M' is ε-SS(q_0). The additional states generated for Q' are marked, as soon as their successor states or transitions under all the symbols of Σ have been generated. The marking of generated states is determined in the partial function *marked*: $\mathcal{P}(Q) \rightarrow$ **bool**.

$q_0' := \varepsilon\text{-SS}(q_0)$; $Q' := \{q_0'\}$; $marked(q_0') =$ **false**; $\delta := \emptyset$;
while there exists $S \in Q'$ **and** $marked(S) =$ **false do**
  $marked(S) :=$ **true**;
  foreach $a \in \Sigma$ **do**
    $T := \varepsilon\text{-SS}(\{p \in Q | (q, a, p) \in \Delta \text{ and } q \in S\})$;
    if $T \notin Q'$
    then $Q' := Q' \cup \{T\}$; (* new state *)
      $marked(T) :=$ **false**
    fi;
    $\delta := \delta \cup \{(S, a) \mapsto T\}$ (* new transition *)
  od
od;

Figura 1.2: Algoritmo de conversión de AFN a AFD



- ⋆ Generalmente el AFD resultante M' no es el más pequeño posible que reconoce el lenguaje del AFN.
- ⋆ Dado un AFD $M = (Q, \Sigma, \delta, q_0, F)$, siempre existe un AFD $M' = (Q', \Sigma, \delta', q'_0, F')$ tal que $L(M) = L(M')$ y $\mid Q' \mid$ es **mínimo**.
- ⋆ La construcción del **autómata mínimo** se basa en clasificar los estados de Q en **clases de equivalencia** que serán los estados de Q', de forma que los estados en una misma clase sean indistinguibles con respecto a las transiciones de M.
- ⋆ La Figura 1.3 muestra el **algoritmo de minimización**. Inicialmente se clasifican los estados de M en dos clases, F y $Q - F$, cuyo comportamiento difiere a efectos de aceptación de palabras. Después el algoritmo recorre **todas** las clases de la partición en curso y en cada iteración trata de descomponer una clase en varias, de forma que dentro de cada subclase todos los estados se comporten igual para todas sus transiciones. Si al recorrer una cierta partición ninguna clase es descompuesta, el algoritmo termina.
- ⋆ En la Figura 1.4 se muestra en dos fases la minimización del AFD que reconocía el lenguaje $1(0 \mid 1)^*$. En la fase (a) tenemos solo dos clases $\{q'_1, q'_2\}$ y $\{q'_3, q'_4\}$, pero q'_1 y q'_2 son distinguibles mediante el símbolo 1, lo que conduce a descomponer esa clase en dos. El resultado se muestra en la fase (b). En esta partición ninguna clase necesita ser descompuesta, luego ese es el autómata mínimo.

1.5. Construcción de analizadores léxicos

- ⋆ Una vez especificadas las unidades léxicas, hay en esencia dos posibilidades: (a) construir el analizador manualmente; (b) generarlo automáticamente.
- ⋆ La **construcción manual** parte del autómata finito determinista (idealmente mínimo) y consiste en programar explícitamente las transiciones válidas mediante instrucciones condicionales y bucles.

Algorithm MinDFA
Input: DFA $M = (Q, \Sigma, \delta, q_0, F)$.
Output: DFA $M_{\min} = (Q_{\min}, \Sigma, \delta_{\min}, q_{0,\min}, F_{\min})$ with $S(M) = S(M_{\min})$ and $Q_{\min}$ minimal.

Method: The set of states of M is divided into a partition which is gradually refined. We already know that two states in different classes of a partition exhibit different 'acceptance behaviour', that is, that there is at least one word w, under which a final state is reached from one of the states, but not from the other. Thus, we begin with the partition $\Pi = \{F, Q - F\}$. The algorithm stops when in some step the partition is not refined further. The procedure terminates since in each iteration step only classes of the current partition may be decomposed into unions of new classes, but Q and thus $\mathcal{P}(Q)$ are finite. The classes of the partition then found are the states of $M_{\min}$. There exists a transition between two new states P and R under a character $a \in \Sigma$, if there exists a transition $\delta(p, a) = r$ with $p \in P$ and $r \in R$ in M. This leads to the program.

$\Pi := \{F, Q - F\}$;
do changed := **false**;
  $\Pi' := \Pi$;
  foreach $K \in \Pi$ **do**
    $\Pi' := (\Pi' - \{K\}) \cup \{\{K_i\}_{1 \le i \le n}\}$, where the K_i are maximal with $K = \bigcup_{1 \le i \le n} K_i$; and $\forall a \in \Sigma : \exists K'_i \in \Pi : \forall q \in K_i : \delta(q, a) \in K'_i$
    if $n > 1$ **then** changed := **true** **fi** (* K was split up *)
  od;
$\Pi := \Pi'$
until not changed ;
$Q_{\min} = \Pi -$ (Dead $\cup$ Unreachable);

$q_{0,\min}$	the class in Π containing q_0.
$F_{\min}$	the classes, containing an element of F.
$\delta_{\min}(K, a) = K'$,	if $\delta(q, a) = p$ with $q \in K$ and $p \in K'$ for one and thus for all $a \in \Sigma$.
$K \in$ Dead,	if K is not a final state and only contains transitions to itself.
$K \in$ Unreachable,	if there is no path from the initial state to K.

Figura 1.3: Algoritmo de minimización de un AFD

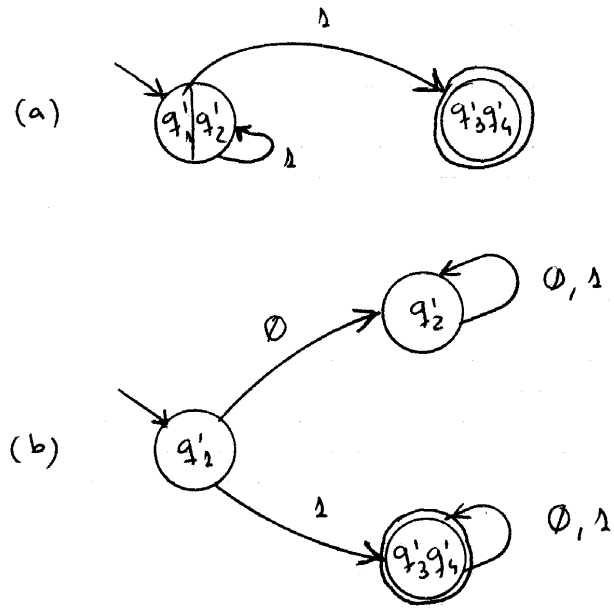


Figura 1.4: Ejemplo de minimización de un AFD

- ⋆ También puede consistir en un simple bucle que procesa un carácter en cada iteración y una **tabla** que implementa la función parcial $\delta : Q \times \Sigma \to Q$.
- ⋆ Los **generadores automáticos** como LEX toman como entrada la expresión regular y producen directamente un analizador dirigido por tabla y la tabla misma. Permiten asociar acciones C al reconocimiento de cada unidad léxica, que serán útiles para implementar un **discriminador** asociado al reconocedor.
- ⋆ En cualquiera de las dos opciones será necesario prever el tratamiento de los **errores léxicos**. Además de reportar los errores, hay que prever **recuperarse** de ellos, dejando el analizador en un estado en que pueda seguir reconociendo unidades léxicas.

1.6. Construcción manual de analizadores léxicos

Antes de comenzar la codificación debe especificarse el analizador léxico mediante un conjunto de definiciones regulares. En dicha especificación conviene distinguir tres tipos diferentes de definiciones:

- *Definiciones léxicas.* Se corresponden con las clases léxicas del lenguaje (v.g., identificador, paréntesis de apertura, punto y coma, palabra reservada **do**, etc.).

- *Definiciones auxiliares.* Son definiciones intermedias que se utilizan para mejorar la legibilidad de las especificaciones, pero que no se corresponden, por sí mismas, con clases léxicas (v.g., letra, dígito, separador, ...).

- *Definiciones de cadenas ignorables.* Caracterizan cadenas en la entrada que, al no afectar al significado del programa, deben ser ignoradas por el analizador léxico (v.g., espacios en blanco, comentarios, ...). Dichas cadenas se denominan *cadenas ignorables*

Una vez finalizada la especificación, esta debe transformarse en un AFD que reconozca las unidades léxicas. Debido a que, para especificaciones de un tamaño relativamente grande, la aplicación manual de los algoritmos puede ser impracticable, puede adoptarse la estrategia siguiente:

- A partir de las definiciones regulares se obtiene una expresión regular para cada clase léxica, así como una expresión regular para cada tipo de cadena ignorable.

- Se construye un AFD equivalente a cada expresión regular asociada con cada clase léxica, así como un AFD equivalente a cada expresión regular asociada con cadenas ignorables. En este último caso, se hace que el estado inicial coincida con el final.

- Se unen los estados iniciales de todos estos AFDs y se *determiniza* el resultado agrupando estados en conjuntos según sea necesario. En dicho proceso, los estados finales de los AFDs para las clases léxicas deben considerarse **distinguibles**.

Una vez disponible el AFD, puede procederse a codificar el analizador léxico. Normalmente dicho analizador se codifica como un componente que mantiene internamente:

- El estado actual del autómata (conviene utilizar enumerados o constantes con nombres significativos para codificar dicho estado).

- El flujo de entrada asociado a la fuente.

- Un *buffer* de entrada con el siguiente (o los siguientes) carácter(es) a procesar (típicamente un único carácter, aunque, dependiendo del lenguaje, podrían precisarse más).

- La última cadena reconocida.

- La posición de inicio (fila, columna) de dicha cadena en la fuente.

- La posición (fila, columna) actual.

Así mismo, típicamente el analizador publica una función para leer la siguiente unidad léxica (v.g., `sigToken`). Aparte de un código (un número, el valor de un enumerado) que identifique la clase léxica, la unidad léxica contendrá los distintos atributos necesarios para el proceso posterior (alternativamente, la función puede devolver el código de clase léxica;

en este caso, el analizador léxico podrá incluir funciones adicionales para recuperar los atributos, que podrán guardarse como variables internas del mismo). En particular, en este método de implementación manual, se consideran los siguientes atributos:

- Las coordenadas de inicio de la unidad léxica
- Opcionalmente, la cadena reconocida: el *lexema* de la unidad léxica. Dicho lexema será necesario para las unidades de clases *multivaluadas* (aquellas para las que existen unidades léxicas con distintos léxemas: v.g., identificadores, números ...), aunque podrá obviarse para las unidades de clases *univaluadas* (aquellas que siempre tienen el mismo lexema: v.g., símbolos de puntuación, palabras reservadas ...)

La estructura típica de esta función es como sigue:

- Una fase de inicialización, en la que se inicializan las variables internas del analizador (estado actual, posición de inicio, ...) para permitir reconocer la siguiente unidad.
- Una fase de reconocimiento, en la que se reconoce la siguiente unidad en la entrada.

Una estructura típica para la fase de reconocimiento es:

- Un bucle `while(true)`.
- En el cuerpo de dicho bucle, un `switch` sobre los posibles estados del AFD
- En cada caso de dicho `switch` la codificación de las acciones de transición para el correspondiente estado. Típicamente, dichas acciones dependen del carácter actual y del tipo de estado (final o no final):
  - Si el carácter actual provoca una transición válida, se actualiza el estado del reconocedor en consecuencia.
  - Si este no es el caso (el carácter actual no está asociado a una transición válida): (i) si el estado actual es final, se construye una unidad léxica adecuada, y se devuelve el control (`return`), (ii) en otro caso, se está ante un error léxico: se activa la función de tratamiento de errores.

De esta forma, ante varias alternativas posibles, el analizador siempre reconocerá la unidad léxica correspondiente con el lexema de mayor longitud. Para finalizar, es conveniente considerar con un poco más de detalle el recoocimiento de las palabras reservadas:

- Una posibilidad es tratarlas como cualquier otro tipo de clase léxica. Sin embargo, dado que la estructura de estas palabras reservadas normalmente se solapa con la estructura de los identificadores, se obtienen diagramas de transiciones muy enrevesados.
- Otra posibilidad (la más habitual) es reconocer únicamente identificadores. Una vez reconocido un identificador, puede consultarse en una tabla si el identificador reconocido se corresponde o no con una palabra reservada, ajustando en consecuencia la clase de la unidad léxica reconocida.

1.6.1. Ejemplo

Se considera un sencillo lenguaje de programación en el que los programas:

- Comienzan por la palabra reservada **evalua**
- Van seguidos por una expresión aritmética. Dicha expresión puede involucrar: (i) valores básicos (números enteros, números reales con decimales, e identificadores), (ii) operaciones aritméticas sencillas (suma: `+`, resta: `-`, producto: `*` y división: `/`). En las expresiones pueden utilizarse paréntesis para alterar las prioridades y asociatividades de los operadores.
- Terminan, opcionalmente, con la palabra reservada **donde** seguida de una lista de definiciones de la forma $identificador = valor$ separadas por coma.

Un ejemplo de programa en este lenguaje:

```
evalua
   166.386 * euros + 1.66386 * (centimos1 + centimos2)
donde
   euros = 567,
   centimos1 = 456,
   centimos2 = 10
```

Vamos a ver cómo desarrollar manualmente un analizador léxico para este lenguaje.

Especificación del léxico del lenguaje

Lo primero que tenemos que hacer es especificar los aspectos léxicos del lenguaje. Para ello:

- Tenemos que saber aislar qué detalles son relevantes para llevar a cabo dicha especificación, y qué detalles quedan fuera del nivel léxico. Desde la perspectiva léxica, nos tenemos que fijar en los componentes básicos del lenguaje (las clases léxicas: identificadores, números, palabras reservadas, símbolos de puntuación ...). Otros aspectos, tales como la estructura que tienen las expresiones, asegurar que los identificadores que se utilizan en las expresiones hayan sido declarados, etc., **no son aspectos relevantes desde el punto de vista léxico** (estos aspectos se tratarán en fases posteriores del desarrollo del procesador de lenguaje). En este ejemplo, podemos distinguir las siguientes clases léxicas: *identificadores*, *números enteros*, *números reales*, operadores *suma*, *resta*, *multiplicación* y *división*, *paréntesis de apertura*, *paréntesis de cierre*, *igual*,*coma*,palabra reservada *evalua* y palabra reservada *donde* (como patrones usuales, cada símbolo de puntuación y cada operador da lugar a una clase léxica; así mismo, cada palabra reservada está asociada con una clase léxica diferente, como ya se ha indicado anteriormente).

- Una vez delimitadas las clases léxicas, tenemos que decidir cómo vamos a separar las unidades léxicas en nuestro lenguaje (normalmente se utilizan blancos, tabuladores, retornos de carro y/o saltos de línea), así como si vamos a incluir o no comentarios. Esto da lugar a clases léxicas ignorables (el analizador léxico *se saltará* las correspondientes unidades, ya que no juegan papel alguno en el procesamiento subsiguiente). En el ejemplo, adoptaremos el convenio habitual para los separadores, y consideraremos comentarios *de línea*: comenzarán por `#` y se extenderán hasta el final de la línea (o el fin de fichero, si no hay tal fin de línea).
- El último paso es describir formalmente todas las clases identificadas utilizando expresiones regulares. Para aumentar la legibilidad de dichas expresiones podrán incluirse clases auxiliares, así como referir clases ya definidas (como ya se indicó anteriormente, no se admiten definiciones recursivas).

Una posible definición léxica para el lenguaje ejemplo es:

- **Definiciones auxiliares**
  $letra \longrightarrow \mathbf{A}|\mathbf{B}|\ldots|\mathbf{Z}|\mathbf{a}|\mathbf{b}|\ldots|\mathbf{z}$
  $digitoPositivo \longrightarrow \mathbf{1}|\ldots|\mathbf{9}$
  $digito \longrightarrow digitoPositivo|\mathbf{0}$
  $parteEntera \longrightarrow digitoPositivo\ \ digito*$
  $parteDecimal \longrightarrow digito*\ \ digitoPositivo$
- **Definiciones de cadenas ignorables**
  $separador \longrightarrow \mathbf{SP}|\mathbf{TAB}|\mathbf{NL}$
  $comentario \longrightarrow \#(\overline{\mathbf{NL}|\mathbf{EOF}})*$
- **Definiciones léxicas**
  $identificador \longrightarrow letra(letra|digito)*$
  $numeroEntero \longrightarrow [+|-]parteEntera$
  $numeroReal \longrightarrow [+|-]parteEntera.parteDecimal$
  $evalua \longrightarrow \mathbf{evalua}$
  $donde \longrightarrow \mathbf{donde}$
  $operadorSuma \longrightarrow \backslash+$
  $operadorResta \longrightarrow -$
  $operadorMultiplicacion \longrightarrow \backslash*$
  $operadorDivision \longrightarrow /$
  $parentesisApertura \longrightarrow \backslash($
  $parentesisCierre \longrightarrow \backslash)$
  $igual \longrightarrow =$
  $coma \longrightarrow ,$

Obsérvese que, dado que un lenguaje puede incluir, en su juego de caracteres, caracteres que se utilizan para escribir expresiones regulares (v.g., $+$,$*$, etc.), es necesario distinguir cuándo dichos caracteres ocurren como caracteres ordinarios en las expresiones regulares, y

cuando se utilizan para construir las expresiones en sí. Para ello puede usarse un mecanismo de *acotación*: así, por ejemplo, $\ast$ denota el carácter "$*$" en lugar del *metacarácter* para el cierre de Kleene. Por su parte, \\ denota el carácter "\".

Diagrama de transición del AFD para el analizador léxico

Aplicando los convenios descritos anteriormente (o incluso utilizando la especificación únicamente como guía) es posible obtener el diagrama de transición para un AFD que reconoce las unidades léxicas del lenguaje (y que consume las unidades ignorables). La Figura 1.5 muestra el diagrama de transición para el lenguaje de ejemplo.

Obsérvese que, aparte de reconocer las clases contempladas en la especificación, se reconoce una clase adicional que se corresponde con el *fin de fichero*. Así mismo, siguiendo la estrategia introducida anteriormente, las palabras reservadas no se reconocen explícitamente, sino que se reconocen como identificadores, relegándose a la implementación el discriminar cuándo el identificador reconocido se corresponde o no con una palabra reservada. Obsérvese, por último, que los estados se nombran con mnemotécnicos significativos (v.g., **recIDec** para el estado al que se llega tras haber reconocido la porción inicial de la parte decimal de un número real). Este convenio facilitará el posterior mantenimiento de la implementación en el caso (habitual) de que el lenguaje se vaya ampliando incrementalmente.

Implementación

Vamos a esbozar la implementación en Java del analizador léxico para este lenguaje (la implementación en otro lenguaje será similar).

Comenzamos eligiendo representaciones adecuadas para las unidades léxicas producidas por el analizador. Dichas unidades se representarán mediante la clase `UnidadLexica`:

```
public abstract class UnidadLexica {
   private ClaseLexica clase;
   private int fila;
   private int columna;
   public UnidadLexica(int fila, int columna, ClaseLexica clase) {
     this.fila = fila;
     this.columna = columna;
     this.clase = clase;
   }
   public ClaseLexica clase () {return clase;}
   public abstract String lexema();
   public int fila() {return fila;}
   public int columna() {return columna;}
}
```

Dicha clase es abstracta, ya que la presencia o no de un atributo *lexema* dependerá de si

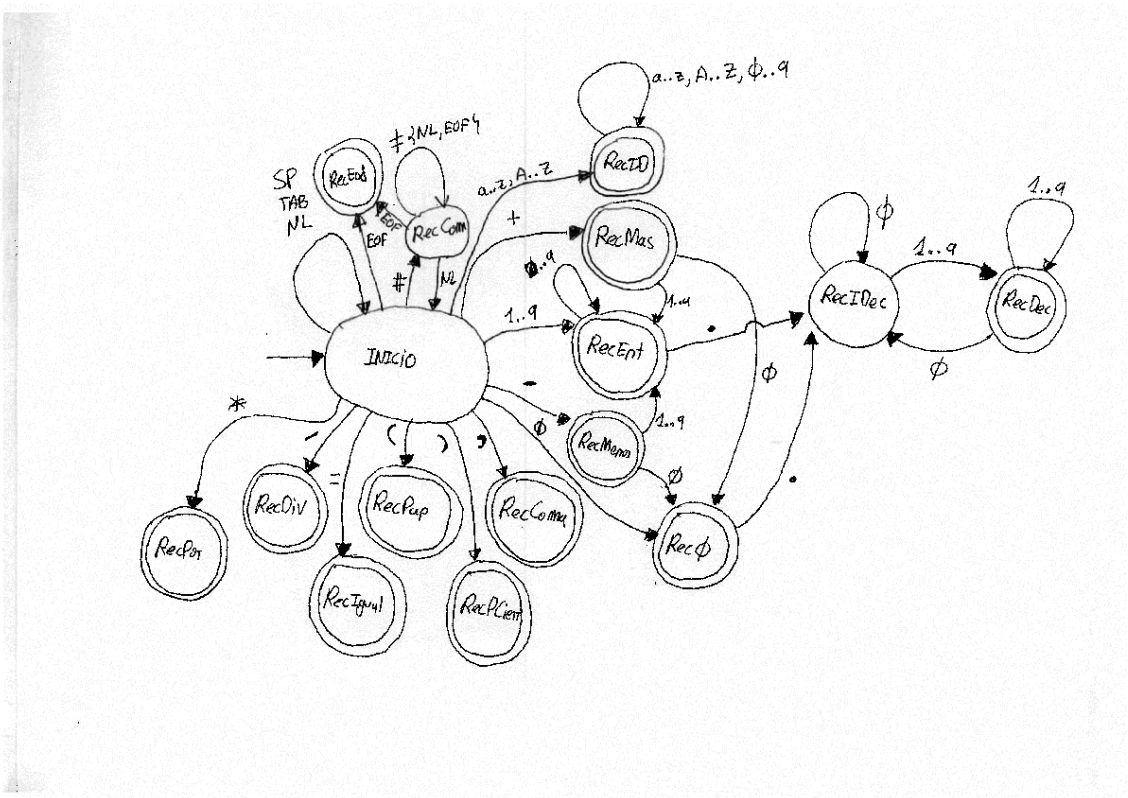


Figura 1.5: AFD para el lenguaje ejemplo

la clase léxica correspondiente es uni o multivaluda. Por tanto, dicha clase se especializará como:

```
public class UnidadLexicaUnivaluada extends UnidadLexica {
   public String lexema() {throw new UnsupportedOperationException();}

   public UnidadLexicaUnivaluada(int fila, int columna, ClaseLexica clase){
     super(fila,columna,clase);
   }
  public String toString() {
    return "[clase:"+clase()+",fila:"+fila()+",col:"+columna()+"]";
  }
}

public class UnidadLexicaMultivaluada extends UnidadLexica {
  private String lexema;
  public UnidadLexicaMultivaluada(int fila, int columna,
                                  ClaseLexica clase, String lexema) {
     super(fila,columna,clase);
     this.lexema = lexema;
   }
  public String lexema() {return lexema;}
  public String toString() {
    return "[clase:"+clase()+",fila:"+fila()+",col:"+columna()+",
            lexema:"+lexema()+"]";
  }
}
```

La codificación de la clase léxica de las unidades puede llevarse a cabo mediante un tipo enumerado:

```
public enum ClaseLexica {
  IDEN, ENT, REAL, PAP, PCIERRE, IGUAL, COMA,
  MAS, MENOS, POR, DIV, EVALUA, DONDE, EOF
}
```

Una vez decidida la representación de las unidades léxicas, el analizador en sí se implementa naturalmente como una clase, que declarará las variables de estado necesarias para llevar a cabo el proceso de análisis:

```
public class AnalizadorLexicoEjemplo {

   private Reader input;        // flujo de entrada
```

```
  private StringBuffer lex;     // cadena para almacenar el lexema de la
                                // u. léxica
  private int sigCar;           // siguiente carácter a procesar

  private int filaInicio;       // fila inicio de la u. léxica
  private int columnaInicio;    // col. inicio de la u. léxica
  private int filaActual;
  private int columnaActual;
  private Estado estado;        // estado del autómata

  ...
}
```

El estado del autómata puede codificarse mediante un tipo enumerado:

```
enum Estado {
       INICIO, REC_POR, REC_DIV, REC_PAP, REC_PCIERR, REC_COMA, REC_IGUAL,
       REC_MAS, REC_MENOS, REC_ID, REC_ENT, REC_0, REC_IDEC, REC_DEC,
       REC_COM,REC_EOF
   }
```

Obsérvese que hay un valor enumerado por cada uno de los estados del diagrama de la Figura 1.5.

La inicialización del analizador supone inicializar adecuadamente los campos del reconocedor: establecer el flujo de entrada, crear el almacén para el lexema, leer el primer carácter y fijar la fila y columna actual:

```
public AnalizadorLexicoEjemplo(Reader input) throws IOException {
    this.input = input;
    lex = new StringBuffer();
    sigCar = input.read();
    filaActual=1;
    columnaActual=1;
   }
```

Por su parte, la función `sigToken` puede proporcionarse como un método público, y programarse siguiendo las guías dadas anteriormente:

```
 public UnidadLexica sigToken() throws IOException {
       // Fase de inicialización
     estado = Estado.INICIO;
     filaInicio = filaActual;
     columnaInicio = columnaActual;
     lex.delete(0,lex.length());
```

```
 // Fase de reconocimiento
while(true) {
    switch(estado) {
      case INICIO:
         if(hayLetra())  transita(Estado.REC_ID);
         else if (hayDigitoPos()) transita(Estado.REC_ENT);
         else if (hayCero()) transita(Estado.REC_0);
         else if (haySuma()) transita(Estado.REC_MAS);
         ...
         else if (hayAlmohadilla()) transitaIgnorando(Estado.REC_COM);
         else if (haySep()) transitaIgnorando(Estado.INICIO);
         else if (hayEOF()) transita(Estado.REC_EOF);
         else error();
         break;
      case REC_ID:
         if (hayLetra() || hayDigito()) transita(Estado.REC_ID);
         else return unidadId();
         break;
      case REC_ENT:
          if (hayDigito()) transita(Estado.REC_ENT);
          else if (hayPunto()) transita(Estado.REC_IDEC);
          else return unidadEnt();
          break;
      case REC_0:
          if (hayPunto()) transita(Estado.REC_IDEC);
          else return unidadEnt();
          break;
      case REC_MAS:
          if (hayDigitoPos()) transita(Estado.REC_ENT);
          else if(hayCero()) transita(Estado.REC_0);
          else return unidadMas();
          break;
      ...

      case REC_POR: return unidadPor();
      ...
    }
  }
}
```

Para aumentar la legibilidad del código se han utilizado diferentes funciones auxiliares (dichas funciones deben implementarse como métodos privados):

- Discriminantes de clases de carácteres (v.g., `hayLetra`,`hayDigitoPos`, etc.). Por ejemplo, `hayLetra` puede implementarse como:

```
private boolean hayLetra() {
    return sigCar >= 'a' && sigCar <= 'z' ||
           sigCar >= 'A' && sigCar <= 'z';
}
```

  (la implementación del resto de las funciones discriminantes es análoga)

- Constructoras de unidades léxicas (v.g., `unidadEnt`, `unidadSuma`, etc.). Ejemplos de implementaciones de estas constructoras:

```
private UnidadLexica unidadEnt() {
   return new UnidadLexicaMultivaluada(filaInicio,columnaInicio,
                                ClaseLexica.ENT,lex.toString());
 }
 private UnidadLexica unidadMas() {
   return new UnidadLexicaUnivaluada(filaInicio,columnaInicio,
                                     ClaseLexica.MAS);
 }
```

- Funciones de transición: `transita`, para transitar a otro estado mientras se está reconociendo una unidad léxica, y `transitaIgnorando`, para cambiar de estado mientras se está reconociendo una cadena ignorable.

- Función de tratamiento de errores: `error`.

Es interesante analizar con un poco más de detalle la determinación de la unidad léxica asociada al reconocimiento de un identificador. Dado que, a fin de simplificar el diagrama de transiciones, las palabras reservadas se reconocen como si fueran identificadores, en la constructora de la unidad léxica asociada a un identificador (`unidadId`) debe determinarse si la cadena reconocida se corresponde con un identificador, o, si por el contrario, se corresponde con una palabra reservada:

```
private UnidadLexica unidadId() {
  switch(lex.toString()) {
      case "evalua":
         return new UnidadLexicaUnivaluada(filaInicio,columnaInicio,
                                           ClaseLexica.EVALUA);
      case "donde":
```

```
            return new UnidadLexicaUnivaluada(filaInicio,columnaInicio,
                                              ClaseLexica.DONDE);
         default:
            return new UnidadLexicaMultivaluada(filaInicio,columnaInicio,
                                                ClaseLexica.IDEN,lex.toString());

      }
   }
```

En este caso sencillo, la discriminación se lleva a cabo mediante programa. En casos más complejos, puede utilizarse una tabla que mapeé lexemas de palabras reservadas en las respectivas clases léxicas.

Por su parte, las funciones de transición capturan el comportamiento genérico del reconocedor asociado con una transición. La transición durante el reconocimiento de una unidad léxica implica fijar el nuevo estado, aumentar el lexema, y leer el siguiente carácter:

```
private void transita(Estado sig) throws IOException {
  lex.append((char)sigCar);
  sigCar();
  estado = sig;
}
```

En el caso de que se esté ignorando una cadena, debe omitirse el añadir el carácter actual al lexema, así como actualizar la fila y columna de inicio de la siguiente unidad léxica:

```
private void transitaIgnorando(Estado sig) throws IOException {
  sigCar();
  filaInicio = filaActual;
  columnaInicio = columnaActual;
  estado = sig;
}
```

Los detalles de bajo nivel relativos a la lectura del siguiente carácter se relegan al método `sigCar` y a la maquinaria asociada al mismo (en particular, dicha maquinaria lleva a cabo el tratamiento del fin de línea de forma independiente del SO subyacente):

```
 // Secuencia de caracteres que representa el fin de línea en
 // la plataforma (en Unix será LF, en DOS CR+LF, en MAC CR, ...)
private static String NL = System.getproperty("line.separator");

private void sigCar() throws IOException {
  sigCar = input.read();
```

```
      // Si es el comienzo del fin de línea,
      // reconocerlo... Como resultado, sigCar se fijará a '\n'
   if (sigCar == NL.charAt(0)) saltaFinDeLinea();
   if (sigCar == '\n') {
      filaActual++;
      columnaActual=0;
   }
   else {
     columnaActual++;
   }
  }
  private void saltaFinDeLinea() throws IOException {
     for (int i=1; i < NL.length(); i++) {
         sigCar = input.read();
         if (sigCar != NL.charAt(i)) error();
     }
     sigCar = '\n';
  }
```

Por último, el método `error` trata los errores léxicos (encontrar un carácter inexperado). En este caso nos conformaremos con un tratamiento simple (aunque no es complicado realizar un tratamiento más sofisticado, siguiendo las guías explicadas anteriormente):

```
private void error() {
     System.err.println("("+filaActual+','+columnaActual+
                        "):Caracter inexperado");
     System.exit(1);
}
```

1.7. Herramientas de Generación de Analizadores Léxicos

La construcción manual de analizadores léxicos, aunque simple y sistemática una vez que se dispone de la especificación de los aspectos léxicos del lenguaje, es bastante laboriosa. Este hecho dificulta también el mantenimiento del componente (aspecto muy importante en el desarrollo de un procesador de lenguaje, desarrollo que suele llevarse a cabo de forma incremental a fin de manejar su complejidad). Afortundamente, los aspectos más farragosos del desarrollo son automatizables aplicando los algoritmos de la teoría de lenguajes formales revisados anteriormente. Efectivamente:

- La especificación basada en expresiones regulares puede transformarse en un AFN aplicando una construcción como la descrita anteriormente (construcción de Thompson)

- La construcción mediante subconjuntos permite obtener un AFD equivalente al AFN obtenido en el paso anterior.
- Por último, dicho AFD puede minimizarse (en este caso, como en el caso de la construcción manual, hay que tener en cuenta que aquellos estados finales que están asociados con el reconocimiento de unidades léxicas diferentes son distinguibles entre sí)

De esta forma, existen varias herramientas que permiten generar analizadores léxicos a partir de su especificación y que, a grandes rasgos, se basan en este proceso.

Un generador de analizadores léxicos clásicos es LEX. La versión original generaba analizadores léxicos escritos en C, aunque posteriormente se implementaron versiones de la herramienta que generaban código en otros muchos lenguajes. Flex es una versión de GNU de LEX, mientras que Flex++ es una versión de Flex que permite generar código en C++ (así como encapsula los analizadores generados en clases C++). JLex y JFlex son generadores de analizadores léxicos para Java (ambos generan analizadores encapsulados como clases Java). Así mismo, existen generadores para otros muchos lenguajes (v.g., ALex es un generador de analizadores léxicos para el lenguaje funcional Haskell).

El uso de un generador de analizadores léxicos facilita substancialmente el desarrollo, al liberar al desarrollador de la codificación manual del DFA. Efectivamente, el desarrollo se reduce a:

- Codificar la especificación en el lenguaje de entrada del generador.
- Configurar las opciones de generación (normalmente esto se realiza también utilizando el lenguaje del generador)
- Proporcionar el código auxiliar necesario para obtener un analizador funcional (normalmente los generadores permitirán intercalar acciones escritas en el código del lenguaje de implementación: dado que los errores en dicho código no serán detectados por el generador, sino que serán detectados más tarde, cuando se compile el analizador, es conveniente mantener dicho código lo más simple posible, utilizando servicios proporcionados por código complementario escrito y depurado independientemente)

A continuación describiremos un ejemplo de un generador de analizadores léxicos sencillo: JLex (el generador, junto con su documentación, puede obtenerse en
`http://http://www.cs.princeton.edu/~appel/modern/java/JLex/`)

1.7.1. JLex

JLex es una implementación en Java de un subconjunto de LEX. La herramienta genera, así mismo, implementaciones de analizadores léxicos con Java.

JLex toma como entrada una especificación JLex, y genera como salida una implementación de un analizador léxico, implementado como una clase Java. La estructura de una especificación JLex es la siguiente:

```
 <<Código adicional>>
%%
 <<Configuración del proceso de generación>>
 <<Definiciones regulares:
    Nombre = Expresión
    Nombre = Expresión
    ... >>
 %%
  <<Descripción del analizador:
    Patrón Acción
    Patrón Acción
    ... >>
```

En esta estructura:

- El *código adicional* se refiere al código Java que se desea incorporar al comienzo del archivo generado: cláusula **package**, cláusulas de importación, clases auxiliares, etc... **IMPORTANTE**: hay que tener en cuenta lo que se ha comentado antes sobre el código escrito en el lenguaje de implementación que se intercala en la especificación...
- La *configuración del proceso de generación* consiste en una serie de cláusulas de configuración. Algunas de las más relevantes son las siguientes:
  - `%line`: Indica al generador que incluya un campo privado `yyline`, en el que se mantendrá la cuenta del número de líneas en el programa procesado (**NOTA**: el analizador generado no informa sobre el número de columna)
  - `%class`: Permite indicar el nombre de la clase que implementa al analizador léxico -por defecto la clase se llama `Yylex`
  - `%type`: Permite especificar el tipo de la unidad léxica (por defecto, se devuelve el código de la clase léxica)
  - `%unicode`: Especifica que el juego de caracteres a utilizar debe ser **unicode**.
  - `%{...%}`: El código introducido entre estas marcas será copiado literalmente en la clase. Permite añadir nuevos campos, nuevos métodos, etc. a la clase generada.
  - `%init{... %}`: El código introducido será añadido literalmente a los constructores de la clase generada (hay un constructor que permite indicar el flujo de entrada como un `InputStream`, y otro que permite indicarlo como un `Reader`).
  - `%eofval{...%eofval}`: Permite especificar el valor que debe devolverse al llegar al fin de fichero.

  (En la documentación aparecen otras opciones de configuración)
- En lo que respecta a la especificación léxica, JLex implementa un subconjunto del lenguaje de especificación de LEX. En particular, las construcciones más habituales utilizadas en las expresiones regulares son:

- *Caracteres individuales.* Como regla general, se pone el carácter literalmente. \ se utiliza como carácter de escape. `$` denota el fin de línea, y `.` cualquier carácter menos el fin de línea. Así mismo, `\n`, `\r`, `\b` y `\t` tienen el mismo significado que en Java, pudiéndose especificar también caracteres mediante su código, como en Java (v.b., `\u5678`).
- *Conjuntos de caracteres.* Expresiones de la forma $[E_0, ..., E_k]$ donde cada E_i denota, bien un carácter, bien un rango de caracteres $c_0 - c_1$.
- *Complemento de conjuntos de caracteres.* Expresiones de la forma $[\hat{}E_0, ..., E_k]$: el conjunto de los caracteres *no* incluidos en $[E_0, ..., E_k]$.
- Referencia a una definición anterior: $\{nombre\}$ (el nombre de la definición entre llaves)
- Expresiones compuestas: la concatenación se escribe poniendo una expresión detrás de otra; la unión se escribe con |, el cierre con *, el cierre positivo con +, la opcionalidad con ?. Las prioridades coinciden con las descritas para expresiones regulares (+ y ? tienen la misma prioridad que *).

(Ver la documentación de la herramienta para otros tipos de expresiones)

- Por último, la descripción del analizador se lleva a cabo indicando *patrones* de cadenas de texto que tienen que aparecer en la entrada (estos patrones son expresiones regulares que, en general, harán referencia a las definiciones), así como las *acciones* que han de ejecutarse una vez que se reconozcan cadenas que satisfacen dichos patrones (dichas acciones son código Java arbitrario). Como regla general:

  - Las acciones para los patrones correspondientes con las definiciones ignorables serán vacías: `{}`
  - Las acciones correspondientes con el reconocimiento de unidades léxicas serán de la forma `{return <<unidad>>;}`

El analizador generado reconoce siempre la cadena más larga posible (esto facilita expresar directamente el reconocimiento de las palabras reservadas, anteponiendo sus patrones al del identificador: no obstante, debe tenerse en cuenta también el incremento en el número de estados del AFD que ello supone). Así mismo, una regla general para tratar los errores léxicos es añadir, como último patrón, uno de la forma `[^]`, y asociarle una acción que permita tratar el error.

En cuanto al analizador generado, como ya se ha indicado puede utilizarse instanciando la clase correspondiente con el flujo de entrada. El método que devuelve la siguiente unidad léxica es `yylex`.

Ejemplo

A modo de ejemplo, se muestra la especificación JFlex para el analizador desarrollado en la sección anterior:

```
package alex;

%%
%line
%class AnalizadorLexicoTiny
%type  UnidadLexica
%unicode

%{
  private ALexOperations ops;
  public String lexema() {return yytext();}
  public int fila() {return yyline+1;}
%}

%eofval{
  return ops.unidadEof();
%eofval}

%init{
  ops = new ALexOperations(this);
%init}

letra  = ([A-Z]|[a-z])
digitoPositivo = [1-9]
digito = ({digitoPositivo}|0)
parteEntera = {digitoPositivo}{digito}*
parteDecimal = {digito}* {digitoPositivo}
separador = [ \t\r\b\n]
comentario = #[^\n]*
evalua = evalua
donde = donde
identificador = {letra}({letra}|{digito})*
numeroEntero = [\+,\-]?{parteEntera}
numeroReal = [\+,\-]?{parteEntera}\.{parteDecimal}
operadorSuma = \+
operadorResta = \-
operadorMultiplicacion = \*
operadorDivision = /
parentesisApertura = \(
parentesisCierre = \)
igual = \=
coma  = \,
%%
{separador}               {}
```

```
{comentario}              {}
{evalua}                  {return ops.unidadEvalua();}
{donde}                   {return ops.unidadDonde();}
{identificador}           {return ops.unidadId();}
{numeroEntero}            {return ops.unidadEnt();}
{numeroReal}              {return ops.unidadReal();}
{operadorSuma}            {return ops.unidadSuma();}
{operadorResta}           {return ops.unidadResta();}
{operadorMultiplicacion}  {return ops.unidadMul();}
{operadorDivision}        {return ops.unidadDiv();}
{parentesisApertura}      {return ops.unidadPAp();}
{parentesisCierre}        {return ops.unidadPCierre();}
{igual}                   {return ops.unidadIgual();}
{coma}                    {return ops.unidadComa();}
[^]                       {ops.error();}
```

Obsérvese que, en todo momento, se trata de mantener el código Java incluido en esta especificación lo más simple prosible. Para ello, se emplea una clase auxiliar `ALexOperations` que implementa operaciones auxiliares análogas a las utilizadas en la implementación manual para devolver las unidades léxicas, y para tratar los errores léxicos (los métodos `lexema` y `fila` que aparecen en la especificación permiten acceder al lexema y al número de fila desde dichas operaciones auxiliares). Obsérvese también que este ejemplo ilustra la estructura general de una especificación JLex: *código adicional* (en este caso, el paquete en el que estará el analizador generado), *configuración del proceso de generación*, *especificación* (la parte más importante), y *descripción del analizador* (como una secuencia de pares *patrón - acción*). Evidentemente, la producción y mantenimiento de esta especificación son bastante más simples que en el caso de la implementación manual.

Notas bibliográficas

Se debe ampliar el contenido de estas notas en (Scott, 2009, Capítulo 2), (Wilhelm y Maurer, 1995, Capítulo 7), (Aho et al., 2006, Capítulo 3) y (Louden, 1997, Capítulo 2) en los cuales están parcialmente basadas. También debe consultarse la documentación de JLex en http://www.cs.princeton.edu/ appel/modern/java/JLex/ -la herramienta también se describe brevemente en (Appel y Palsberg, 2002, Capítulo 2)

Ejercicios

1. Demostrar las siguientes propiedades de las expresiones regulares:

   - la concatenación es asociativa con elemento neutro ϵ.
   - '|' es conmutativo, asociativo, e idempotente.

- la concatenación es distributiva con respecto a '|', tanto por la izquierda como por la derecha.
- $(r^*)^* = r^*$.

2. Especificar mediante una expresión regular una unidad léxica *identificador* que comienza por letra, sigue por letras o números en longitud variable y puede contener el carácter '_', pero nunca como primero, último, ni dos de ellos consecutivos.

3. Especificar mediante una expresión regular los comentarios de bloque de Pascal, encerrados entre dos llaves, y de C++, encerrados entre las cadenas `/*` y `*/`.

4. Especificar mediante una expresión regular un comentario de bloque encerrado entre dos cadenas `##`, que puede contener internamente el carácter `#`, pero nunca dos de ellos consecutivos.

5. Especificar mediante una expresión regular un literal en coma flotante en Fortran, que admite las siguientes posibilidades:

   - no tiene parte decimal
   - tiene una parte entera seguida de '.'
   - tiene un '.' seguido de una parte decimal
   - tiene una parte entera, un '.' y una parte decimal

6. Especificar mediante una expresión regular una constante literal *cadena de caracteres* que comienza y termina con " y puede contener dentro pares "" que no indican el final de la cadena.

7. Especificar mediante una expresión regular todas las cadenas de $\Sigma = \{0, 1\}$ que no contengan dos ceros consecutivos.

8. Dada la expresión regular (0 | 1)*00(0 | 1)*, realizar los siguientes apartados:

   - Siguiendo el esquema dado en este tema, dibujar el AFN que reconoce dicho lenguaje.
   - Usar el algorimo dado para convertir este último a AFD.
   - Usar el algoritmo dado para encontrar el AFD mínimo equivalente.

9. Realizar los mismos apartados con la expresión regular resultado del Ejercicio 5.

10. Realizar los mismos apartados con la expresión regular resultado del Ejercicio 6.

11. Realizar los mismos apartados con la expresión regular resultado del Ejercicio 7.

12. Identificar las unidades léxicas presentes en el siguiente programa Pascal. Indicar qué atributos es adecuado devolver para cada una de ellas.

```
function max (i, j: integer) : integer ;
    { Devuelve el maximo de dos enteros }
begin
    if i > j then max := i else max := j
end
```

13. Una constante de tipo carácter de un cierto lenguaje puede tomar las siguientes formas:

`'c'`	Carácter imprimible entre comillas simples
`\126`	Código ASCII del carácter en decimal
`\o133`	Código ASCII del carácter en octal
`\xA7`	Código ASCII del carácter en hexadecimal
`\c`	Código especial $c \in \{n, r, t, ", ', \backslash\}$

    Escribir una definición regular para la constante carácter, y realizar todos los pasos necesarios para llegar al autómata finito mínimo que reconoce dicha unidad.

14. Considera un lenguaje de programación sencillo en el que los programas están formados por: (i) una sección de *declaraciones*, y (ii) una sección de *instrucciones*. Ambas secciones están separadas por `&&`. La sección de declaraciones, por su parte, está compuesta por una o más declaraciones, separadas por *puntos y coma*. Cada declaración consta de: (i) un *nombre de tipo*, (ii) un *nombre de variable*. Los nombres de tipo pueden ser **int** o **bool**. Por su parte, los nombres de variable comienzan necesariamente por una letra, seguida de una secuencia de cero o más letras, dígitos, o subrayado (`_`). Por su parte, la sección de instrucciones consta de una o más instrucciones separadas por *puntos y coma*. El lenguaje únicamente considera instrucciones de *asignación*. Dichas instrucciones constan de una *variable*, seguida de `=`, seguida de una expresión. Las expresiones básicas consideradas son números enteros con y sin signo (un signo `+` o `-` opcional, seguido de uno o más dígitos), **true** y **false**. Los operadores que pueden utilizarse en las expresiones son los operadores aritméticos binarios usuales (`+`, `-`, `*` y `/`), el *menos* unario (`-`), los operadores lógicos **and**, **or** y **not** y los operadores relacionales (`<`,`>`, `<=`, `>=`,`==`,`!=`). También es posible utilizar paréntesis para alterar las precedencias y asociatividades de los operadores.

    Ejemplo de programa

```
int peso;
bool pesado
&&
peso = (45 * 12) / 2;
pesado = (peso >  10) or (peso / 2 <= 4)
```

(a) Determina las clases léxicas de este lenguaje, (b) especifica formalmente el léxico del mismo, (c) diseña manualmente un analizador léxico para este lenguaje y esboza su implementación en Java, (d) implementa un analizador léxico para este lenguaje utilizando JLex.

Bibliografía

Durante mucho tiempo se creyó que esos libros impenetrables correspondían a lenguas pretéritas o remotas.

Jorge Luis Borges

Aho, A., Lam, M. S., Sethi, R. y Ullman, J. *Compilers: Principles, Techniques and Tool (2nd Edition)*. Addison-Wesley, 2006.

Appel, A. W. y Palsberg, J. *Modern Compiler Implementation in Java (2nd Edition)*. Cambridge University Press, 2002.

Louden, K. C. *Compiler Construction: Principles and Practice*. Course Technology, 1997.

Scott, M. L. *Programming Language Pragmatics*. 3rd edition. Morgan Kaufmann, 2009.

Wilhelm, R. y Maurer, D. *Compiler Design*. Addison-Wesley, 1995.