TEMA 2

ANÁLISIS LÉXICO

2.1 EL PAPEL DEL ANALIZADOR LÉXICO

El analizador léxico es la primera fase de un programa traductor. Es, por otra parte, el único que gestiona el fichero de entrada. Es la parte del compilador que lee los caracteres del programa fuente y que construye unos símbolos intermedios (*elementos léxicos* que llamaremos "***tokens***") que serán posteriormente utilizados por el analizador sintáctico como entradas.

El analizador sintáctico debe obtener una representación de la estructura (sintaxis) del programa fuente. Para realizar esta tarea debería concentrarse solamente en la estructura y no en otros aspectos menos importantes, como los espacios en blanco o tabuladores, los cambios de línea, los comentarios, etc. Además, los árboles sintácticos construidos con una gramática que genere los programas carácter a carácter no son útiles para construir una traducción.

¿Por qué separar el análisis léxico del sintáctico?

- El diseño de las partes posteriores dedicadas al análisis queda simplificado.
- Con fases separadas, se pueden aplicar técnicas específicas y diferenciadas para cada fase, que son más eficientes en sus respectivos dominios.
- Se facilita la portabilidad. Si se quiere cambiar alguna característica del alfabeto del lenguaje (por ejemplo para adaptarlo a determinados símbolos propios de máquinas distintas) sólo tenemos que cambiar el analizador léxico.

Si tomamos por ejemplo las expresiones "`6-2*30/7`" y "`6 - 2* 30/7`", podemos comprobar que la estructura de ambas expresiones es equivalente; sin embargo, los caracteres que componen ambas cadenas no son los mismos. Si tuvieramos que trabajar directamente con los caracteres estaríamos dificultando la tarea de obtener la misma representación para ambas cadenas. Si consideramos además la cadena "`8-2*3/5`", la estructura de esta cadena es de nuevo la misma que la de las cadenas anteriores, lo único que cambia son los valores concretos de los números. Por estos motivos (y también por eficiencia), el procesamiento de los caracteres se deja en manos del analizador léxico que entregará a las sucesivas etapas del compilador los componentes léxicos (*tokens*) significativos.

EJEMPLO:

Usando la cadena del ejemplo anterior, "`6-2*30/7`" (o la otra igual salvo el número de espacios), ambas serían representadas por el analizador léxico como la siguiente cadena de elementos léxicos:

<entero,6> <resta,–> <entero,2> <por,∗> <entero,30> <div,/> <entero,7>

donde cada *token* ha sido representado por un par en el que la primera componente de cada par es el tipo de *token* y la segunda componente es el *lexema* (el valor concreto de ese *token*). La tercera cadena del ejemplo anterior tendría la misma estructura que las otras dos, pero con distintos valores de los *lexemas*.

❑

En definitiva, el análisis léxico agrupará los caracteres de la entrada por categorías léxicas, establecidas por la especificación léxica del lenguaje fuente como veremos más adelante. Esta especificación también establecerá el alfabeto con el que se escriben los programas válidos en el lenguaje fuente y, por tanto, el analizador léxico también deberá rechazar cualquier texto en el que aparezcan caracteres ilegales (no recogidos en ese alfabeto) o combinaciones ilegales (no permitidas por las especificaciones léxicas) de caracteres del alfabeto.

Veremos que los componentes léxicos se especifican mediante *expresiones regulares* que generan lenguajes regulares, más sencillos de reconocer que los lenguajes independientes del contexto, y permiten hacer un análisis más rápido. Además, una gramática que represente la sintaxis de un lenguaje de alto nivel carácter a carácter sería mucho más compleja (para implementar un proceso de traducción a partir de ella) que otra que representase la misma sintaxis en función de sus componentes léxicos.

2.2 ERRORES LÉXICOS

Pocos son los errores característicos de esta etapa, pues el compilador tiene todavía una visión muy local del programa. Por ejemplo, si el analizador léxico encuentra y aísla la cadena "`wihle`" creerá que es un identificador, cuando posiblemente se tratara de un `while` mal escrito y no será él el que informe del error, sino que lo harán sucesivas etapas del análisis del texto.

Los errores que típicamente detecta el analizador léxico son:

- Utilizar caracteres que no pertenecen al alfabeto del lenguaje (p.ej.: 'ñ' o '±').
- Se encuentra una cadena que no coincide con ninguno de los patrones de los *tokens* posibles (p.ej.: en un lenguaje '`:=`' puede ser la asignación pero que no permita '`:`' solo).

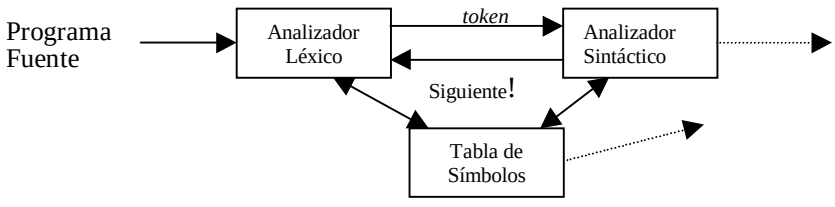
Cuando el analizador léxico encuentra un error, lo habitual es parar su ejecución e informar, pero hay una serie de posibles acciones por su parte para anotar los errores, recuperarse de ellos y seguir trabajando:

- Ignorar los caracteres no válidos hasta formar un *token* según los patrones dados;
- Borrar los caracteres extraños;
- Insertar un carácter que pudiera faltar;
- Reemplazar un carácter presuntamente incorrecto por uno correcto;
- Conmutar las posiciones de dos caracteres adyacentes.

Estas transformaciones se realizan sobre el prefijo de entrada que no concuerda con el patrón de ningún *token*, intentando conseguir un lexema válido. No obstante, todas son complicadas de llevar a cabo y peligrosas por lo equivocadas que pueden resultar para el resto del análisis.

2.3 FUNCIONAMIENTO DEL ANALIZADOR LÉXICO

La principal función del analizador léxico es procesar la cadena de caracteres y devolver pares (*token* , lexema). Debe funcionar como una subrutina del analizador sintáctico (ver figura).



Operaciones que realiza el analizador léxico:

- Procesado léxico del programa fuente: identificación de *tokens* y de sus lexemas que deberá entregar al analizador sintáctico y (puede que) interaccionar con la tabla de símbolos.
- Maneja el fichero del programa fuente; es decir: abrirlo, leer sus caracteres y cerrarlo. También debería ser capaz de gestionar posibles errores de lectura.
- Ignora los comentarios y, en los lenguajes de formato libre, ignora los separadores (espacios en blanco, tabuladores, retornos de carro, etc.).
- Cuando se produzca una situación de error será el analizador léxico el que sitúe el error en el programa fuente (tal línea, tal posición). Lleva la cuenta de las líneas procesadas.
- Preproceso de macros, definiciones, constantes y órdenes de inclusión de otros ficheros.

Cada vez que el analizador sintáctico llame al léxico éste debe leer caracteres desde donde se quedó en la anterior llamada hasta conseguir completar un nuevo *token*, y en ese momento debe devolver el par (*token* , lexema). Cuando el analizador léxico intenta reconocer algunos tipos de *tokens* como los identificadores o los números se produce una circunstancia especial: el analizador léxico debe leer caracteres hasta que lea uno que no pertenece a la categoría del *token* que está leyendo; ese último carácter (que no tiene por qué ser un espacio en blanco) no puede perderse, y debe devolverse al buffer de entrada para ser leído en primer lugar la próxima vez que se llame al analizador léxico.

EJEMPLO:

En la cadena "`Grande / 307>=`" marcamos las posiciones a las que tiene que llegar el analizador léxico para decidir qué *tokens* ha reconocido.

```
Grande / 307>=
      ↑↑    ↑↑
```

Como se puede observar, para el caso del identificador "`Grande`", ha tenido que ir una posición más allá del final del mismo, pues sólo allí, al encontrar el espacio en blanco puede saber que el nombre del identificador ha concluido. Para encontrar el símbolo "`/`" basta con ponerse sobre él y *verlo* si consideramos que no es prefijo de ninguno otro (para este lenguaje imaginario). Para el número "`307`" sucede lo mismo que con el identificador: hay que llegar hasta un carácter que no sea un número para saber que el número ha terminado. Ese símbolo en este caso es el signo "`>`" de "`>=`" (mayor o igual que). Después, para reconocer este nuevo *token* el léxico avanzará para ver si es el "`=`" lo que sigue al "`>`". Como sí que lo es y suponemos que "`>=`" no es prefijo de ningún otro, el analizador devolverá el *token* "mayor o igual", si no hubiera aparecido el igual al avanzar, hubiera tenido que retroceder una posición y devolver el *token* "mayor".

❑

El analizador léxico debe intentar leer siempre el *token* más largo posible. Puede ocurrir que haya leído ya un *token* correcto y al intentar leer un *token* más largo no sea posible; en este caso no se debe producir un error, sino que el analizador léxico debe devolver el *token* correcto y debe retroceder en el *buffer* de entrada hasta el final de ese *token*.

EJEMPLO:

Si el operador "`!=`" (*distinto*) pertenece al lenguaje pero el carácter "`!`" no, cuando aparezca en la entrada este carácter, el analizador debe leer el siguiente carácter; si es un "`=`", devolverá el *token* correspondiente al operador *distinto*, pero si no es un "`=`", debe producir un error léxico si el carácter "`!`" por sí solo no perteneciera al lenguaje.

❑

2.4 ESPECIFICACIÓN DE UN ANALIZADOR LÉXICO

2.4.1 Definiciones de términos comunes en esta fase

- ***Tokens***: desde el punto de vista léxico son los elementos léxicos del lenguaje mientras que para el resto de las fases de un compilador son los símbolos terminales de la gramática (por ejemplo: palabras reservadas, identificadores, signos de puntuación, constantes numéricas, operadores, cadenas de caracteres, etc.). Es posible, dependiendo del lenguaje, que varios signos formen un solo *token* (“`:=`”, “`==`”, “`+=`”, “`||`”, etc.).
- **Patrón**: expresión regular que define el conjunto de cadenas que puede representar a cada uno de los *tokens.*
- **Lexema**: secuencia de caracteres del código fuente que concuerda con el patrón de un *token.* Es decir, cuando analizamos el texto fuente y encontramos una cadena de caracteres que representa un *token* determinado diremos que esa cadena es su lexema.
- **Atributos**: El análisis léxico debe proporcionar información adicional sobre los *tokens* en sus atributos asociados. El número de atributos depende de cada *token*. En la práctica, se puede considerar que los *tokens* tienen un único atributo, un registro que contiene toda la información propia de cada caso (por ejemplo, lexema, tipo de *token* y línea y columna en la que fue encontrado). Lo normal es que toda esa información se entregue a los analizadores sintáctico y semántico para que la usen como convenga.

EJEMPLO:

Token	**Lexemas**	**Patrón (E.R.)**
Identificador	`Pepe, cons1, ...`	Letra • (Letra | Dígito)*
Num_Entero	`10, –105, +24`	(+|–|ε) • Dígito$^+$
PR_IF	`If , if , IF , iF`	(i | I) • (F | f)

Cuando el analizador léxico encuentra un lexema devuelve como información a qué *token* pertenece y todo lo que sabe de él, incluido el propio lexema. En el último caso, se supone que esa palabra pertenece a un lenguaje en el que mayúsculas y minúsculas son equivalentes.

❑

Para especificar correctamente el funcionamiento de un analizador léxico se debe utilizar una máquina de estados, llamada ***diagrama de transiciones*** (DT), muy parecida a un autómata finito determinista, con las siguientes diferencias:

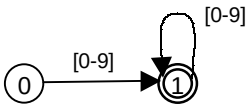
- Un AFD sólo dice si la cadena de caracteres pertenece al lenguaje o no; un DT debe funcionar como un analizador léxico; es decir, debe leer caracteres hasta que complete un *token*, y en ese momento debe retornar (en los estados de aceptación) el *token* que ha leído y dejar el *buffer* de entrada preparado para la siguiente llamada.
- Un DT no puede tener estados de absorción (para cadenas incorrectas en AFDs) ni de error (se considerará que las entradas para las que no hay una transición desde cada estado son error).
- De los estados de aceptación de un DT no deben salir transiciones.
- En el caso de las tiras no específicas, necesitamos otro estado al que ir cuando se lea un carácter que no pueda formar parte del patrón. En este último estado (al que se llega con la transición especial **otro**) se debe devolver al *buffer* de entrada el carácter leído (que puede ser parte del siguiente *token*), lo cual se indica marcando el estado con un asterisco, y se debe retornar el *token* correspondiente a ese estado de aceptación. Por ejemplo, para reconocer números enteros, con un AFD son necesarios solamente dos estados; con un DT necesitamos ese otro estado al que ir cuando se lea un carácter que no pueda formar parte del número.

En el caso más general, se suelen utilizar estos diagramas de transiciones para reconocer los *tokens* de entrada, construidos a partir de sus patrones correspondientes, expresados mediante las respectivas expresiones regulares. Estos autómatas se combinan en una máquina única que, partiendo de un único estado inicial, sigue un recorrido u otro por los estados hasta llegar a alguno de los estados de aceptación. En función de en cuál se detenga devolverá un *token* u otro. Si no llega a un estado de aceptación o recibe una entrada que no le permite hacer una transición a otro estado, entonces dará error.

EJEMPLO:

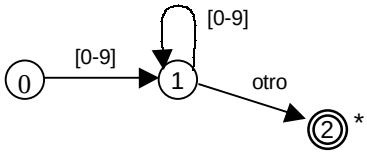
A continuación se muestra un ejemplo de reconocedor de números enteros sin signo mediante la expresión regular $[0\text{-}9]^+$.

El AFD sería:



El estado ((1)) reconoce números enteros.

y el DT:



El estado ((2)) devuelve el *token* Num_entero

Como se observa, en el DT surge un nuevo estado, que es realmente el de aceptación y que está marcado con un asterisco que indica que se llega a él leyendo un carácter más de los necesarios para reconocer ese *token*, y por tanto hay que devolver ese carácter a la entrada. La transición a ese estado se hace mediante la entrada **otro** que significa cualquier otro carácter del alfabeto del lenguaje que no esté en el rango [0-9].

❑

Si durante el recorrido del autómata se produce una transición no autorizada o la tira de entrada finaliza en un estado no de aceptación, el analizador informará del error. Este tipo de máquinas es útil para lenguajes con grandes conjuntos de elementos léxicos distintos y las matrices de transición resultantes tienen grandes zonas vacías que conviene comprimir y resumir mediante algoritmos adecuados. Cuando los lenguajes son poco extensos es mejor redactar los analizadores "a mano", tratando de tomar decisiones adecuadas en función de los caracteres que van apareciendo en la entrada (ver apartado de implementación, más adelante).

El analizador suele tener unos subprogramas auxiliares encargados de gestionar el *buffer* (técnicas de doble *buffer*, saltos de línea, <EOF>, etc.) y de ir devolviendo caracteres al *buffer* de entrada cada vez que el procedimiento de reconocimiento y aislamiento de *tokens* lo requiera.

2.4.2 Identificación de palabras reservadas

Las palabras reservadas son aquellas que los lenguajes de programación reservan para usos particulares. El problema que surge es: ¿cómo reconocer las palabras reservadas si responden al mismo patrón que los identificadores, pero son *tokens* diferentes al *token* "identificador"?.

Existen dos enfoques para resolver este problema: 1) **resolución implícita**: considerar que todas son identificadores y buscarlas en una tabla. Implica saltarse el formalismo para buscar una solución práctica (factible si se implementa el A.Léx. "a mano" y preferible si el lenguaje tiene muchas palabras reservadas); 2) **resolución explícita**: se indican todas las expresiones regulares de todas las palabras reservadas y se integran los DT resultantes de sus especificaciones léxicas en la máquina reconocedora (los analizadores resultan mucho más

complejos, pero es necesario si usamos programas de generación de automática de analizadores a partir de especificaciones).

La primera solución citada consiste en considerar que las palabras reservadas son en principio identificadores, y entonces el analizador leerá letras y dígitos hasta completar un identificador, e inmediatamente antes de retornar el *token* "identificador", comparar el lexema leído con una lista de las palabras reservadas, para ver si coincide con alguna de ellas.

En definitiva, se procede normalmente tratando las palabras reservadas como lexemas particulares del patrón del identificador, y cuando se encuentra una cadena que responde a dicho patrón, se analiza si es una palabra reservada o un identificador.

Una posible solución para ello es, en el A.L.:

- Primero inicializar la tabla de símbolos con todas las palabras reservadas (lo normal es hacerlo por orden alfabético para facilitar la posterior búsqueda y acceso).
- Cuando encuentre un identificador se irá a mirar la tabla de símbolos
  ⇒ SI lo encuentra en la zona reservada para ellas ENTONCES es una palabra reservada
  ⇒ SI NO, será un identificador, que, como tal, será añadido a la tabla de símbolos.

EJEMPLO:
Si se encuentra el identificador "`Cont`" en la entrada, antes de que el A.L. devuelva el token identificador deberá comprobar si se trata de una palabra reservada. Si el número de palabras reservadas es muy grande lo mejor es tenerlas almacenadas desde el principio de la compilación en la tabla de símbolos, para ver si allí ya se encuentra definida esa cadena como tal. Aquí hemos supuesto que no es así y "`Cont`" queda registrado como un identificador.

Tabla de Símbolos

`do`	Pal.Reservada	↑
`end`	Pal.Reservada	Zona de
`for`	Pal.Reservada	palabras
`while`	Pal.Reservada	reservadas
`...`	....	↓
`Cont`	Identificador	↓ Zona de identificadores
`...`	....	

❑

La disposición de una tabla ordenada con las palabras reservadas es útil cuando el número de éstas es grande. Cuando el lenguaje tiene sólo unas pocas puede ser más práctico el realizar la identificación "directamente", mediante una serie de `if`s que comparen con las cadenas correspondientes a esas palabras.

Cuando la detección de palabras reservadas se hace, en cambio, explícitamente, entonces los patrones de la especificación léxica del lenguaje tendrán su correspondencia en el diagrama de transiciones global a partir del cual se implementará el analizador léxico. Las especificaciones léxicas de las palabras reservadas, como tiras específicas que son, constarán de concatenaciones de caracteres y pueden ser siempre prefijos de identificadores (como por ejemplo "do" –palabra reservada– y "dos" –identificador–). Aparecerán en estos casos los problemas de prefijos y tiras no específicas que se describen en los ejemplos que se ofrecen más adelante.

EJEMPLO:
Si el lenguaje se define como *sensible al tamaño*:

```
d·o
e·n·d
f·o·r
w·h·i·l·e
```

Si no lo es (mayúsculas y minúsculas equivalentes):

```
(d|D)·(o|O)
(e|E)·(n|N)·(d|D)
(f|F)·(o|O)·(r|R)
(w|W)·(h|H)·(i|I)·(l|L)·(e|E)
```

❑

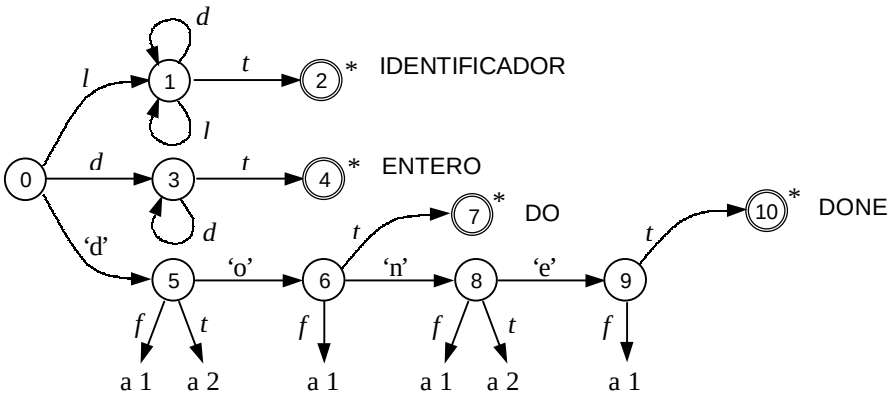
Cuando en la especificación léxica del lenguaje coexisten expresiones regulares de tiras no específicas como los identificadores con las de específicas como las palabras reservadas, hay que llevar más cuidado porque cualquiera de las palabras reservadas puede ser un prefijo de un identificador válido. Esto motiva que los subautómatas que reconocen las palabras deben estar comunicados con el de los identificadores (ver ejemplo más abajo).

Por otra parte, cuando un elemento léxico es prefijo de otro y ambos son tiras específicas, aparecerán estados de aceptación que partirán de estados intermedios (ver ejemplo de la página siguiente).

EJEMPLO:

Constrúyase un diagrama de transiciones para el reconocimiento de identificadores, números enteros sin signo y las palabras reservadas “do” y “done”.

Notación: *d* = dígito; *l* = letra; *t* = otro; *f* = otro alfanumérico (dígito o letra); a *n* = ir al estado *n*.



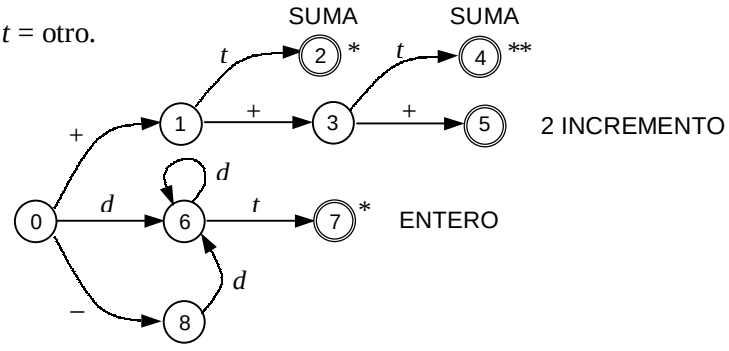
Como se observa en este diagrama de transiciones, todos los estados de aceptación están marcados con asterisco, por lo que siempre habrá que devolver el último carácter leído al *buffer* de entrada. Esto es debido, en este ejemplo, a que todos los *tokens* son bien unos prefijos de otros (do → done → identificador) o bien son tiras no específicas (entero e identificador).

❑

EJEMPLO:

Constrúyase un diagrama de transiciones para el reconocimiento de números enteros con signo negativo o sin signo (ER: $(-|\varepsilon)\bullet d^{+}$) y los operadores suma (“+”) y doble incremento (“+++”).

Notación: *d* = dígito; *t* = otro.



Obsérvese que el estado de aceptación del *token* "doble incremento" no lleva asterisco por ser tira específica y no ser prefijo de ninguna otra, y por tanto no necesita leer el siguiente carácter y retroceder. Sí que lo llevan los estados de aceptación del *token* "suma" a pesar de ser específicas, por ser prefijos del "doble incremento". Además, uno de ellos lleva dos asteriscos, indicando que si se llega a ese estado hay que retornar el *token* "suma" y devolver dos caracteres al *buffer* de entrada.

❑

2.5 IMPLEMENTACIÓN DE ANALIZADORES LÉXICOS

Existen distintas posibilidades para de crear un analizador léxico, las tres más generales son:

1.- Usar un generador automático de analizadores léxicos, como el LEX: su entrada es un código fuente con la especificación de las expresiones regulares de los patrones que representan a los *tokens* del lenguaje, y las acciones a tomar cuando los detecte.

* Ventaja: comodidad y rapidez en el desarrollo.
* Inconveniente: ineficiencia del analizador resultante y complicado mantenimiento del código generado.

2.- Escribir el AL en un lenguaje de alto nivel de uso general utilizando sus funciones de E/S.

* Ventaja: más eficiente y compacto.
* Inconveniente: hay que hacerlo todo a mano.

3.- Hacerlo en lenguaje ensamblador.

* Ventaja: máxima eficiencia y compacidad.
* Inconveniente: muy complicado de desarrollar.

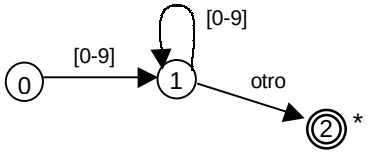
Como se ha indicado, la forma más cómoda de implementar un analizador léxico es con un generador automático de analizadores léxicos, como `lex`, si bien no es la forma más eficiente. Si se opta por hacerlo "a mano", se puede hacer de varias maneras: implementando el diagrama de transiciones simulando las transiciones entre estados o bien se pueden implementar "directamente", usando estructuras de selección múltiple (`switch` en C, `case` en Pascal, etc.) para, según cual sea el primer carácter del *token*, leer caracteres hasta completar el *token*. Por supuesto, con esta técnica también es necesario devolver caracteres al *buffer* de entrada.

La opción intermedia es utilizar un enfoque mixto en el que se mezcle el análisis manual con el análisis mediante máquinas reconocedoras. Se optará así por analizar mediante estructuras de selección múltiple los elementos léxicos de estructura más sencilla (usualmente los operadores) y dejar para el análisis mediante diagramas de transiciones de los elementos léxicos definidos como tiras no específicas, prefijos comunes, etc. Luego todo ello se empaquetará dentro de una única función que se encargará del análisis léxico.

La forma de implementar el diagrama de transiciones es mediante la construcción de su *tabla de transiciones*. Para ello se etiquetan las filas como los estados del DT y las columnas como las distintas posibles entradas a las que hay que añadir el *token* que se reconoce y el número de caracteres que hay que devolver a la entrada después del reconocimiento.

EJEMPLO:

A partir del DT construido antes para los números enteros:



La tabla de transiciones correspondiente será:

	Entradas			
estado	0-9	Otro	***token***	***Retroceso***
0	1	Error	-	-
1	1	2	-	-
2	-	-	Num_entero	1

❑

Esta es la forma de trabajo de cualquier construcción de un analizador léxico: a partir de las especificaciones léxicas en forma de expresiones regulares se construye la máquina reconocedora (DT) y se representa mediante la tabla de transiciones. Una vez que se tiene ésta, el analizador léxico la recorrerá cada vez mediante un bucle con la sentencia:

```
Estado := TablaTransiciones [ Estado , Entrada ];
```

que intentará llegar a un estado de aceptación en el que restaurará la entrada según lo que diga el campo "Retroceso" para ese número de estado, y devolverá el lexema y *token* encontrados.

El recorrido se inicializa con la variable Estado en el valor del estado inicial (0 en el ejemplo anterior) e itera hasta llegar a un estado de aceptación (2 en el ejemplo anterior).

EJEMPLO:

Vamos a construir la tabla de transiciones del ejemplo anterior en el que se reconocían números enteros con signo negativo o sin signo (ER: $(-|\varepsilon)\bullet d^+$) y los operadores suma ("+") y doble incremento ("+++"). La tabla de transiciones correspondiente al DT dibujado allí sería en este caso la siguiente:

	Entradas				
estado	+	–	D	***token***	***Retroceso***
0	1	8	6	-	-
1	3	2	2	-	-
2	-	-	-	SUMA	1
3	5	4	4	-	-
4	-	-	-	SUMA	2
5	-	-	-	2INCR	0
6	7	7	6	-	-
7	-	-	-	ENTERO	1
8	ERROR	ERROR	-	-	-

Obsérvese que las casillas de las filas correspondientes a estados de aceptación nunca tienen valores porque, por definición de los DT, de esos estados no se puede ir a ningún otro. El analizador léxico debe detenerse al llegar a cualquiera de ellos. En cambio las casillas vacías de filas que corresponden a estados de no aceptación están etiquetadas como error.

❑

2.5.1 Prioridad de *tokens*

Por otro lado, lo normal cuando se construye un A.L. es establecer criterios para dar más prioridad a unos *tokens* que a otros. Criterios:

- Dar prioridad al *token* para el que encontramos el lexema más largo. P.ej: "`DO`" / "`DOT`", el generador se quedaría con el más largo ("`DOT`") como identificador (otro ejemplo: "`>`" y "`>=`" se debe quedar con el segundo).

- Si el lexema es el mismo que se puede asociar a dos *tokens* (patrones), estos patrones estarán definidos en un orden, así se asociará al que esté primero.

EJEMPLO:

Si en la especificación léxica aparecen (entre otras) las expresiones regulares

`w•h•i•l•e` → Palabra reservada `while`
`l•(l|d)*` → Identificador

Si en la entrada aparece el lexema "`while`", este puede ser generado por ambas expresiones regulares, pero como está primero la de la palabra reservada, el A.L. debe devolver dicho *token*, no el de identificador. Si estas especificaciones aparecieran en el orden inverso:

`l•(l|d)*` → Identificador
`w•h•i•l•e` → Palabra reservada `while`

Entonces, el A.L. debería devolver siempre el *token* identificador y no devolver nunca la palabra reservada "while" (lo cual no suele ser práctico).

❑

En el siguiente ejemplo vamos a ver cómo se suelen estructurar los analizadores léxicos construidos con lenguajes de programación de alto nivel (C en este caso), utilizando la técnica de la diferenciación manual de los distintos *tokens* mediante estructuras de selección múltiple.

EJEMPLO:

El lexema puede ser una variable global, aunque si se organizan los *tokens* como estructuras que almacenan todos los atributos rel ativos a cada *token* devuelto, puede ser un campo de ellas.

```
int analex(void)
{
  c = obtenercaracter();
  switch (c)
  {
    case ' ' :
    case '\t':
    case '\n':/* y para los demás separadores, no hacer nada */ break;
    case '+' :
    case '-' : return(ADDOP);
    case '*' :
    case '/' : return(MULOP);
    /* ... resto de operadores y elementos de puntuación ... */
    default: if (ESNUMERO(c))
               { /* leer caracteres mientras sean números */
                 /* devolver al buffer de entrada ultimo carácter leído */
                 /* almacenar el lexema leído */
                 return (NUMINT);
               }
             else if (ESLETRA(c))
               { /* leer caracteres mientras sean letras o números */
                 /* devolver al buffer de entrada ultimo carácter leído */
                 /* comprobar si es una palabra reservada */
                 /* almacenar el lexema leído */
                 return(token); /* que será palab.reservada o ident.*/
               }
  }  /* del switch */
}  / de analex */
```

❑

EJERCICIOS RESUELTOS Y PROPUESTOS:

Ejercicio 1. Diseñar un analizador léxico que utilice un DT construido a partir de las expresiones regulares de los patrones de los tokens involucrados en expresiones algebraicas (los operadores son "∗", "+", "–", "/", "(" y ")") en las que intervengan números enteros y reales (en notación no exponencial) sin signo y variables expresadas mediante identificadores.

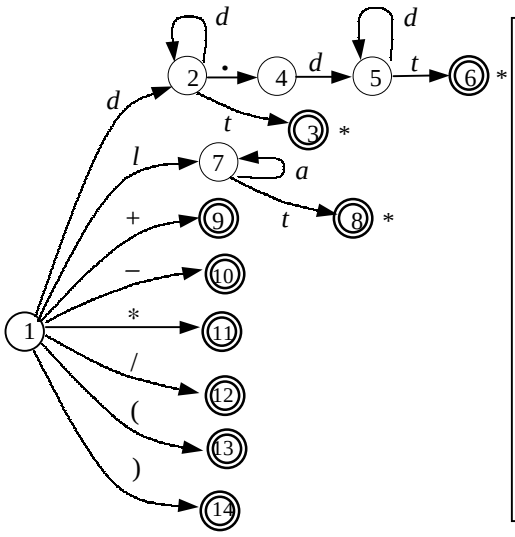
Las expresiones regulares no triviales son:

Números enteros: $dígito^+$
Números reales: $dígito^+$ (**.** $dígito^+$)?
Identificadores: letra (letra | dígito)*

SOLUCIÓN:

Se pueden calcular los AFD a partir de las definiciones de los operadores (tiras específicas) y de las expresiones regulares (tiras no específicas) y luego se combinan todos, para que puedan compartir tabla de estados, a partir de un mismo estado inicial, resultando como sigue:

Notación: *d* = dígito; *l* = letra; *a* = alfanumérico (dígito | letra); *t* = otro.



	díg.	.	+	–	∗	/	letr.	(	)	otro	Token	Dev.
1	**2**	0	**9**	**10**	**11**	**12**	**7**	**13**	**14**	0	-	-
2	**2**	**4**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	-	-
3	-	-	-	-	-	-	-	-	-	-	NENT	1
4	**5**	0	0	0	0	0	0	0	0	0	-	-
5	**5**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	-	-
6	-	-	-	-	-	-	-	-	-	-	NREA	1
7	**7**	**8**	**8**	**8**	**8**	**8**	**7**	**8**	**8**	**8**	-	-
8	-	-	-	-	-	-	-	-	-	-	ID	1
9	-	-	-	-	-	-	-	-	-	-	SUM	0
10	-	-	-	-	-	-	-	-	-	-	RES	0
11	-	-	-	-	-	-	-	-	-	-	MUL	0
12	-	-	-	-	-	-	-	-	-	-	DIV	0
13	-	-	-	-	-	-	-	-	-	-	PARI	0
14	-	-	-	-	-	-	-	-	-	-	PARD	0

Tabla de transiciones
(los 0 son error)

Se observa que el patrón de los enteros es un prefijo del patrón de los reales, por lo que se puede construir para ambos una misma rama del autómata en la que aparecerá un estado de aceptación que surge de un estado intermedio para los enteros (3) y otro final para los reales (6). En este caso, todos los estados de aceptación llevan asociado el reconocimiento de un *token* distinto. Aquellos que están marcados con "*" se debe a que el reconocimiento para esos *tokens* se produce cuando se ha leído uno o más caracteres más allá del final del lexema correspondiente a dicho patrón, por lo que llevan aparejada como acción asociada el retroceso de la marca de análisis sobre el *buffer* de entrada.

En la tabla de transiciones, las celdas marcadas con "0" son transiciones que dan error. Se puede observar la poca eficiencia del almacenamiento en la tabla.

Ejercicio 2. Hacer lo mismo pero implementándolo "a mano" en C.

Ejercicio 3. Diseñar un diagrama de transiciones determinista para reconocer los siguientes componentes léxicos:

while	la palabra reservada "`while`" (en minúsculas)
when	la palabra reservada "`when`" (en minúsculas)
ident	cualquier secuencia de letras (mayúsculas y minúsculas) y dígitos que empiece por una letra, y que no coincida con ninguna de las palabras reservadas
opersum	el símbolo "+"
opermul	el símbolo "∗"
operinc	el símbolo "++"

Ejercicio 4. Diseñar un diagrama de transiciones determinista para reconocer los siguientes componentes léxicos:

letras	cualquier secuencia de una o más letras (mayúsculas y minúsculas)
entero	cualquier secuencia de uno o más dígitos
explos1	la palabra reservada "`bang`"
explos2	la palabra reservada "`boom`"
true	la secuencia "`:-)`"
false	la secuencia "`:-(`"
asignar	la secuencia "`:=`"

Ejercicio 5. Diseñar un diagrama de transiciones determinista para reconocer los siguientes componentes léxicos:

read	la palabra reservada "`read`"
print	la palabra reservada "`print`"
pradir	la palabra reservada "`pradir`"
redir	la palabra reservada "`redir`"
ident	cualquier secuencia de letras o dígitos que comience por una letra y no sea ninguna de las palabras reservadas
raya	el símbolo "`-`"
punto	el símbolo "`.`"
uno	la secuencia "`.----`"
dos	la secuencia "`..---`"

E indíquese como separa este analizador la secuencia de entrada "`pradir6dire..--.-`"