TEMA 2

ANÁLISIS LÉXICO

Índice

La primera fase de un compilador es el analizador léxico. Este tema se dedica al estudio de los fundamentos teóricos de esta parte del compilador, así como a su diseño e implementación. Las herramientas teóricas fundamentales del análisis léxico son las expresiones regulares y los autómatas finitos que las reconocen. Su implementación se puede realizar manualmente o mediante un generador automático de analizadores léxicos.

2.1. Objetivos del analizador léxico

Antes de describir los objetivos del analizador léxico, conviene introducir previamente tres términos esenciales para la comprensión de la función del analizador léxico:

- Un *token* es un par formado por un *nombre de token* y un *atributo opcional*. Un nombre de token es un símbolo abstracto que representa un tipo de unidad léxica, como por ejemplo una palabra clave concreta, o una secuencia de caracteres que denotan un identificador. Los nombres de token actúan como *símbolos terminales* de entrada de la etapa siguiente, es decir, del análisis sintáctico. A menudo nos referiremos a un token mediante su nombre escrito en negrita.
- Un *patrón* es una descripción de la forma que toman los lexemas de un token. En el caso de una palabra clave, el patrón es precisamente la secuencia de caracteres que forma dicha palabra clave. Los identificadores y otros tokens tienen un patrón más complejo, que pueden verificar muchas secuencias de caracteres distintas. Emplearemos *expresiones regulares* como notación formal para representar los patrones de tokens.
- Un *lexema* es una secuencia concreta de caracteres de un programa fuente que verifica el patrón de un token, y que es identificada por el analizador léxico como una *instancia* de dicho token.

Ejemplo 2.1. La siguiente es una sentencia en el lenguaje de programación C:

```
printf("Total = %d\n",puntos);
```

Tanto `printf` como `puntos` son lexemas que verifican el patrón del token **identificador** de acuerdo con la definición de la tabla 2.1, mientras que `"Total = %d\n"` es un lexema que verifica el patrón **literal**.

Token	Descripción informal	Ejemplo de lexemas
if	Caracteres `i`, `f`.	`if`
relación	Operadores binarios relacionales.	`<`, `<=`, `>=`, `!=`, `==`
identificador	Letra, seguida de letras y dígitos.	`puntos, pi, D2`
número	Cualquier constante numérica.	`46, 3.14, 10e3`
literal	Cualquier secuencia de caracteres que no sean `"`, rodeado por caracteres `"`	`"Hola"`

Tabla 2.1: Ejemplos de definición de tokens.

Dentro del contexto general de un compilador, el *analizador léxico* tiene como función principal agrupar los caracteres del programa fuente en *lexemas*, y generar una lista ordenada de *tokens* a partir de los caracteres de entrada. Esos tokens, o componentes léxicos, serán usados por el analizador sintáctico para construir el árbol sintáctico. Así pues, se debería considerar al analizador léxico como un módulo subordinado al analizador sintáctico, tal y como se indica en el esquema de interacción entre ambos, que se muestra en la figura 2.1.

Al ser la parte del compilador que tiene acceso al texto fuente, el analizador léxico puede realizar otras tareas adicionales a la identificación de lexemas, como por ejemplo:

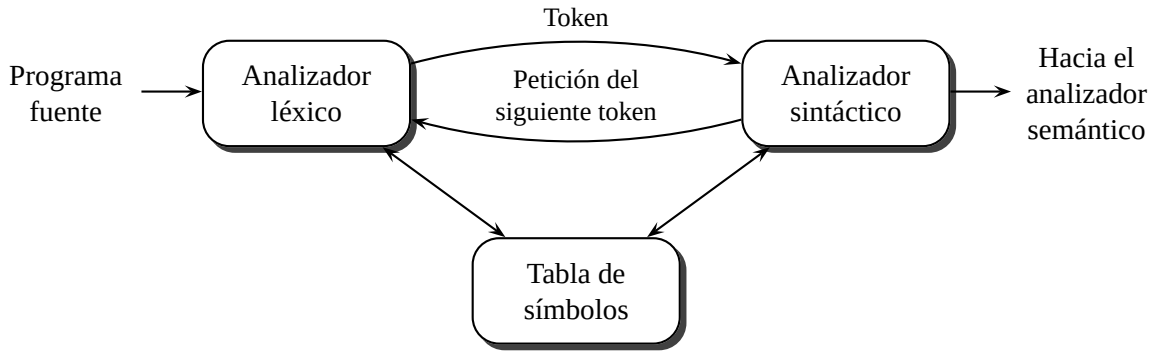


Figura 2.1: Esquema de interacción entre el analizador léxico y el sintáctico.

- Eliminar comentarios. Una vez reconocido un lexema de un comentario, no se genera ningún token puesto que no aporta nada a la generación de código.
- Eliminar caracteres de espaciado. Al igual que con los comentarios, los lexemas de espacios en blanco se reconocen, pero no se generan tokens para ellos.
- Mantener contadores de números de línea y columna del texto fuente para generar la localización de los mensajes de error.

2.1.1. Atributos de los tokens

El analizador léxico guarda información sobre algunos tokens, necesaria en el proceso de análisis y síntesis, en forma de atributos de esos componentes léxicos. Por ejemplo, el patrón del token **número** describe secuencias como `321` o `1.12`, y es obviamente importante que el generador de código conozca cuál es el lexema asociado a cada instancia de este token en el código fuente. Por tanto, el analizador léxico no sólo devuelve el nombre del token, sino que en algunas ocasiones también devuelve el valor de un atributo mediante el cual se puede obtener el lexema encontrado.

La información almacenada en este atributo depende del token. Por ejemplo, un token **identificador**, usado para representar variables, puede tener asociada información como el lexema, el tipo de dato, la posición del fichero fuente en la que se declaró (para generar mensajes de error más útiles), etc. Esta información, actualizada y usada a lo largo de las diferentes fases del proceso de compilación, se suele mantener en la tabla de símbolos, de modo que el atributo asociado a un token **identificador** podría ser un puntero a una entrada en la tabla de símbolos.

2.1.2. Detección y reparación de errores léxicos

No son muchos los errores que puede detectar un analizador léxico sin ayuda de otras fases del compilador. Sin embargo, puede darse la situación de que, en el proceso de reconocimiento del siguiente lexema, se lea una cadena que no concuerde con ninguno de los patrones correspondientes a los tokens del lenguaje. En este caso, el analizador léxico debe informar del error y, en la medida de lo posible, debe intentar recuperarse para seguir analizando el programa fuente en busca de otros errores.

Por ejemplo, algunas versiones de Pascal requieren que la expresión de un número en punto flotante, menor que 0 y positivo, comience con un `0`. Por tanto, el lexema `0.5` se reconoce como un token válido mientras que el lexema `.5` no lo es. Otro tipo de error léxico que el analizador léxico podría detectar es el de exceder el número de caracteres máximo para un identificador.

2.1.3. Separación del análisis léxico y el sintáctico

Todo lo que puede ser descrito mediante expresiones regulares puede también ser descrito con gramáticas libres de contexto. Por tanto, cabría preguntarse por qué no incluir la definición de los tokens en las gramáticas libres de contexto de los lenguajes de programación. Existen varias justificaciones:

- El diseño es más sencillo y modular. Aunque el analizador sintáctico podría encargarse de reconocer los lexemas, el uso de un analizador léxico permite simplificar notablemente el analizador sintáctico. Por ejemplo, este último no tiene que encargarse de tratar espacios en blanco o comentarios.
- Mejora en la eficiencia del compilador: un analizador léxico que emplee técnicas de *buffering* para la lectura del programa fuente puede acelerar el análisis global de forma muy notable.
- Mejora en la portabilidad del compilador: las peculiaridades del interfaz de entrada (dispositivo, codificación de caracteres, etc.) quedan restringidas al analizador léxico.
- Las reglas léxicas de un lenguaje suelen ser bastante sencillas, y no se requiere la potencia de las gramáticas libres de contexto para describirlas.
- Las expresiones regulares generalmente proporcionan una notación más concisa y fácil de entender para la definición de los tokens de la gramática.
- Se pueden desarrollar de forma automática analizadores léxicos si se parte de las expresiones regulares, mientras que si se intenta desde gramáticas con formatos más arbitrarios es más complicado.

No existen reglas firmes que determinen qué poner en las reglas léxicas y qué incluir en las sintácticas. Se puede decir que las expresiones regulares son más útiles en la descripción de la estructura de los identificadores, las constantes, las palabras claves, los espacios en blanco, etc. Por su parte, las gramáticas son más útiles para la descripción de estructuras anidadas tales como los paréntesis balanceados, los pares `begin-end`, los `if-then-else`, y otras estructuras similares cuya recursividad no puede ser tratada con las expresiones regulares.

2.2. Fundamentos teóricos

Como se ha indicado anteriormente, las *expresiones regulares* sirven para especificar la sintaxis de los símbolos básicos de un lenguaje de programación. Por tanto, podemos utilizar los *autómatas finitos* como reconocedores de esos símbolos. A continuación recordamos las definiciones básicas de dichos conceptos.

2.2.1. Expresiones regulares

El concepto de expresión regular, introducido por el matemático norteamericano Simon Cole Kleene en 1956, se desarrolla después que el de autómata finito, como respuesta a la necesidad de expresar de un modo formal los lenguajes reconocidos por estos últimos.

Definición 2.1. *Dado un alfabeto V, y los símbolos $\emptyset$ (conjunto vacío), λ (palabra vacía), $|$ (unión), $\circ$ (concatenación), $*$ (cierre o clausura), se dice que:*

1. *El símbolo $\emptyset$ es una expresión regular.*
2. *El símbolo λ es una expresión regular.*
3. *Cualquier símbolo $a \in V$ es una expresión regular.*
4. *Si α y β son dos expresiones regulares, $\alpha \mid \beta$ y $\alpha \circ \beta$ son expresiones regulares.*

5. *Si α es una expresión regular, α^* es una expresión regular. Se define:*

$$\alpha^* = \bigcup_{i=0}^{\infty} \alpha^i$$

en donde α^i es igual a la concatenación de α consigo misma i veces, siendo $\alpha^0 = \lambda$.

6. *Sólo son expresiones regulares las que se obtienen aplicando las reglas anteriores un número finito de veces sobre los símbolos de V, λ y $\emptyset$.*

Debe asumirse que :

- El operador unario $*$ tiene la mayor precedencia, y es asociativo por la izquierda.
- La concatenación $\circ$ tiene la segunda mayor precedencia, y es asociativa por la izquierda.
- La unión $|$ es el operador de menor precedencia y también es asociativo por la izquierda.

En cualquier caso, la precedencia se puede modificar usando paréntesis.

Definición 2.2. *A cada expresión regular, α, podemos asociarle un subconjunto de V^*, $L(\alpha)$, que es el lenguaje regular que describe α, según las siguientes reglas:*

- *Si $\alpha = \emptyset$, $L(\alpha) = \emptyset$.*
- *Si $\alpha = \lambda$, $L(\alpha) = \{\lambda\}$.*
- *Si $\alpha = a$, $L(\alpha) = \{a\}$.*
- *Si α y β son dos expresiones regulares, entonces $L(\alpha \mid \beta) = L(\alpha) \cup L(\beta)$.*
- *Si α y β son dos expresiones regulares, entonces $L(\alpha \circ \beta) = L(\alpha)L(\beta)$.*
- *Si α es una expresión regular, entonces $L(\alpha^*) = (L(\alpha))^*$.*

Definición 2.3. *Dos expresiones regulares α y β son equivalentes (se escribe $\alpha = \beta$) si describen el mismo lenguaje, $L(\alpha) = L(\beta)$.*

2.2.2. Autómatas finitos

Como se indicó anteriormente, los autómatas finitos constituyen el modelo de cálculo que reconoce los lenguajes generados por las gramáticas regulares. Por tanto, el proceso de análisis léxico puede realizarse implementando uno de estos autómatas. Éste debe reconocer el lenguaje formal descrito mediante una expresión regular, que a su vez describirá la sintaxis de los símbolos básicos del lenguaje de programación.

2.2.2.1. Autómatas finitos deterministas (AFD)

Un AFD tiene, por cada estado y por cada símbolo de su alfabeto de entrada, exactamente una transición hacia otro estado.

Definición 2.4. *Un AFD es una 5-tupla $\{Q, V, \delta, q_0, F\}$ en donde:*

- *Q es un conjunto finito de estados posibles del autómata.*
- *V es el alfabeto de entrada al autómata.*

- q_0 *es el estado inicial.*
- $F \subseteq Q$ *es el conjunto de estados finales.*
- $\delta : Q \times V \rightarrow Q$ *es la función de transición del autómata.*

2.2.2.2. Autómatas finitos no deterministas (AFND)

Los AFD presentan una única transición para cada carácter y estado. El siguiente tipo de autómata posibilita el que haya varias transiciones posibles para un mismo estado y carácter de entrada.

Definición 2.5. *Un AFND es una 5-tupla* $\{Q, V, \Delta, q_0, F\}$ *donde todos los componentes coinciden con los de la definición de AFD excepto la función de transición, que se define de la siguiente forma:*

- $\Delta : Q \times (V \cup \{\lambda\}) \rightarrow \mathcal{P}(Q)$

donde $\mathcal{P}(Q)$ *representa el conjunto de las partes de* Q*.*

2.3. Diseño de un analizador léxico

El diseño de un analizador léxico no debe plantearse como una cuestión trivial. Es necesario seguir un proceso metodológico para implementar esta fase de forma adecuada. El proceso se puede resumir en los siguientes puntos:

1. Identificar los tokens del lenguaje, y definir sus patrones utilizando expresiones regulares.
2. Obtener el autómata finito de cada expresión regular, y combinar todos los autómatas en uno global, minimizando el número de estados. Por combinación entendemos la unión de autómatas, necesaria porque la expresión regular que maneja el analizador léxico es $R_1|R_2|...|R_n$, donde Ri es la expresión regular del i-ésimo componente léxico (incluidos espacios, comentarios y tratamiento de error).
3. Programar el autómata, simulando su ejecución con la técnica que se considere oportuna.
4. Diseñar y programar el interfaz entre la entrada estándar y el autómata.
5. Diseñar y programar el interfaz entre el analizador sintáctico y el autómata que simula el analizador léxico. Normalmente se programa el analizador léxico como una función llamada por el analizador sintáctico, devolviéndole un token cada vez que lo requiera, así como la información asociada a él.
6. Diseñar y programar el tipo abstracto de dato que representa la *tabla de símbolos*, en algunos casos necesaria en el proceso de interacción entre el analizador sintáctico y el léxico. Generalmente, es el analizador sintáctico el encargado de inicializar la tabla de símbolos, que se actualizará y usará a lo largo de todo el proceso de traducción.
7. Especificar y programar el manejo de errores del analizador léxico.

Todas estas tareas son importantes y necesarias. Sin embargo, nosotros usaremos la herramienta *Flex* para realizar el analizador léxico, y esto implica que delegaremos gran parte de ellas en dicha herramienta. En particular, *Flex* genera una implementación en C basándose en un fichero de especificación de expresiones regulares. Es decir, las tareas 2. y 3. se realizan de forma automática, mientras que la 4. y la 5. se reducirán al uso de ciertas funciones y variables que *Flex* proporciona al usuario para interaccionar con la entrada y el analizador sintáctico (si lo hubiera).

2.3.1. Especificación y reconocimiento de componentes léxicos

Como ya se ha comentado, los tokens se pueden describir mediante expresiones regulares y se pueden reconocer mediante autómatas finitos.

La expresión regular de un token describe todos los lexemas que dicho token puede tener asociados. Los autómatas finitos se construyen a partir de las expresiones regulares que describen los tokens, y guían el desarrollo de los analizadores léxicos, encargados del reconocimiento de los lexemas.

Ejemplo 2.2. Los identificadores en Java se describen mediante la siguiente expresión regular:

$$(L \mid \_ \mid \$)(L \mid \_ \mid \$ \mid D)^*$$

en donde $L = a|b|\ldots|z|A|\ldots|Z$ y $D = 0|1|\ldots|9$.

Para un buen reconocimiento léxico, se deben diseñar con cuidado los posibles tipos de tokens. La tabla 2.1 recoge algunos ejemplos. Habitualmente se consideran componentes léxicos:

- *Palabras clave*: son cadenas que forman parte del lenguaje de programación.
- *Operadores*.
- *Identificadores*.
- *Constantes*: reales, enteras y de tipo carácter.
- *Cadenas de caracteres*.
- *Signos de puntuación*.

Debe procurarse que los tokens sean sencillos y que los lexemas sean independientes. Aún así, podemos encontrar algunas dificultades a la hora de reconocer tokens, descritas a continuación.

2.3.1.1. Expresiones regulares en *Flex*

- Se forman combinando **símbolos** y **operadores**.
- Si alguno de los operadores se quiere utilizar como otro carácter cualquiera debe ir precedido por \.
- Todo lo que vaya entre “ se toma como una cadena de caracteres literal.
- El operador [ ] indica unión o clases de caracteres. Dentro de los corchetes son significativos:
  - El operador - , que indica rangos.
  - El operador ^, que debe ir al principio, y complementa al conjunto de caracteres entre corchetes.
  - El operador \ que indica caracteres de escape.
  - También pueden aparecer expresiones de clases de caracteres entre la siguente estructura [: :] (ej: [:digit:], [:alnum:], etc.)
- El operador . se refiere a todo carácter excepto new-line.
- El símbolo ? indica elemento opcional.
- Los símbolos $*$ y + son los operadores de clausura.
- El operador | indica alternativas.
- Los paréntesis ( ) se usan para construir expresiones regulares más complejas.

- {} indica repeticiones y también se usa para expandir definiciones.
- Si ^ aparece fuera de corchetes indica que la expresión regular a su derecha se reconoce sólo después de new-line.
- $ indica que la expresión regular a su izquierda se reconoce solo antes de new-line.
- / sirve para reconocer una expresión regular sólo si va seguida de otra.
- <<EOF>> se usa para realizar acciones al encontrar un final de fichero. No puede usarse con otras expresiones regulares.

Precedencia de Operadores

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

$*$? +
concatenación
repetición {}
$ ^
|
/

2.3.1.2. Caracteres de anticipación

A veces es necesario leer uno o más caracteres extra de la entrada, denominados *caracteres de anticipación*, para decidir el código de token reconocido. Por ejemplo, si el lenguaje de programación admite los operadores < y <=, es necesario, una vez que se lee de la entrada el símbolo <, comprobar si el siguiente es o no el símbolo =. En caso de que lo sea, el lexema encontrado es <=. Si no lo es, dicho carácter de anticipación se devuelve a la entrada, y el lexema encontrado es <.

Flex maneja de forma automática el problema de "**look_ahead**", reconociendo cuándo se necesita leer uno o más caracteres extra para determinar el final de un token (reglas que acaban en +, $*$, ? , $, /, y siempre que un lexema de una expresión regular pueda ser prefijo de otro), y devolviendo a la entrada los caracteres no usados *(push_back)*.

2.3.1.3. Comentarios y espacios en blanco

En la mayoría de los casos, el analizador léxico debe encargarse de reconocer los comentarios y espacios en blanco, y eliminarlos. De esta forma, el analizador sintáctico no tiene que preocuparse por ellos. Existe la alternativa de modificar la gramática para incorporar los espacios en blanco y los comentarios en la sintaxis del lenguaje, pero esta opción no es tan fácil de implementar.

Por otra parte, dependiendo del lenguaje de programación, los espacios en blanco pueden actuar como *delimitadores* o no. Se dice que un espacio en blanco es delimitador si su aparición implica que los caracteres no blancos anterior y posterior pertenecen a lexemas distintos.

En el caso de tener espacios en blanco delimitadores, el analizador debe eliminarlos sin más. Sin embargo, en caso de que los espacios en blanco no sean delimitadores, además de eliminarlos, el analizador léxico debe agrupar los lexemas anterior y posterior a ellos.

Por otro lado, en algunos lenguajes con formato de línea, como FORTRAN, se exige que ciertas construcciones aparezcan en posiciones fijas de la línea de entrada (por ejemplo, que se comience en la séptima columna). Así, la alineación de un lexema puede ser importante para determinar si es correcto o no. En este caso, la definición de un token cuyo lexema está formado por seis blancos, podría facilitar esta labor. Hoy en día se tiende a diseñar lenguajes de programación independientes del formato.

2.3.1.4. Final de fichero

La definición del token `EOF` (End Of File) puede facilitar el análisis sintáctico posterior, pues permitiría comprobar si después del final del programa aparecen más símbolos, o bien si el fichero termina antes de que termine la escritura de un programa completo.

2.3.1.5. Palabras clave

Las palabras clave pueden estar *reservadas* o no, dependiendo del lenguaje de programación. Si están reservadas, su significado está predefinido y el usuario no puede modificarlo usándolas, por ejemplo, como identificadores. En este caso, el analizador léxico debe reconocerlas directamente (a través del autómata finito) o bien usando una tabla de palabras reservadas. Si las palabras clave no están reservadas, entonces el analizador léxico las reconoce como identificadores, y la tarea de distinguirlas de estos queda relegada al analizador sintáctico. Por ejemplo, en el lenguaje `PL/1` las palabras clave no son reservadas, y por lo tanto una sentencia de este tipo tiene sentido:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

En el caso de que las palabras clave sean reservadas, debemos de asegurarnos de que nuestro analizador léxico no las confunda con los identificadores. Cuando tenemos un lexema que podemos asociar a varias expresiones regulares, *Flex* resuelve la *ambigüedad* con los siguientes criterios:

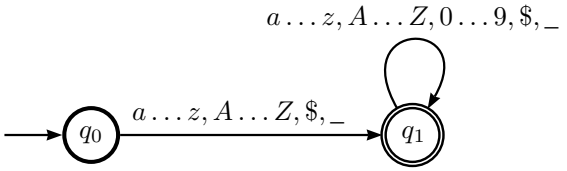
- Se reconoce el lexema más largo posible que pueda estar asociado a la expresión regular.
- Si el más largo puede corresponder a varias expresiones regulares, se asocia a la que aparezca antes.

Por tanto, *colocando las palabras reservadas antes que la expresión regular que reconoce los identificadores, resolvemos el problema asociado a ellas*.

2.3.2. Construcción del autómata finito

Una vez que hemos definido los patrones de cada token en forma de expresiones regulares, es necesario construir los autómatas finitos que reconocen los lenguajes descritos por éstas.

Ejemplo 2.3. Partiendo de la expresión regular de los identificadores de Java, indicada en el ejemplo 2.2, se puede construir el siguiente autómata finito:



Las transiciones que no se representan conducen a un estado erróneo.

Para llegar hasta el autómata que reconoce las expresiones regulares, existen varias alternativas:

- Se puede obtener el AFD correspondiente a la expresión regular que se desea reconocer.
- También se puede obtener el AFND, aunque su simulación es más costosa que la del AFD.
- Una última alternativa es obtener primero el AFND y, seguidamente, construir un AFD equivalente.

Por otra parte, existen algoritmos para minimizar el número de estados de un AFD y para producir representaciones rápidas y más compactas de la tabla de transiciones del autómata. De forma general, para simular un autómata finito con un programa suele interesar que el autómata sea determinista y mínimo.

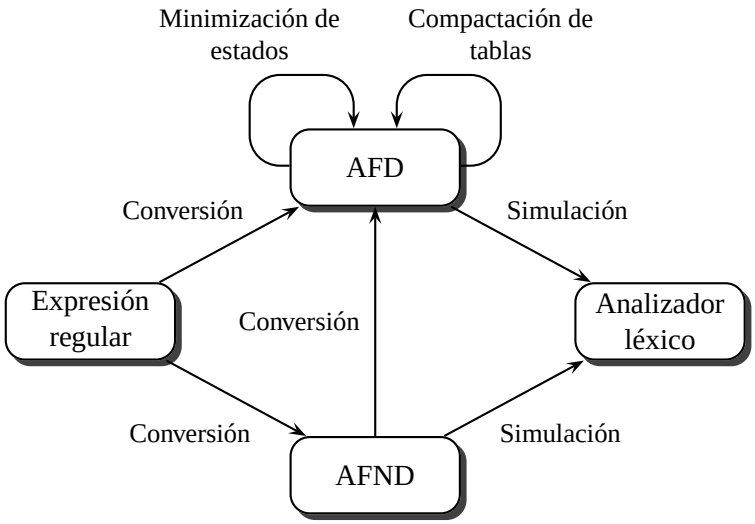


Figura 2.2: Construcción de un analizador léxico a partir de una expresión regular.

La figura 2.2 resume los algoritmos aplicables a los autómatas finitos que son de interés en el análisis léxico. Dado que el algoritmo de conversión de una expresión regular a un AFD es más complejo que el de conversión a un AFND, habitualmente se suele escoger la segunda opción empleando el algoritmo de *construcción de Thompson*. Una vez obtenido el AFND, se suele aplicar un algoritmo de conversión del AFND a un AFD equivalente (*construcción de subconjuntos*). Finalmente se pueden aplicar algoritmos de minimización de estados y compactación de tablas sobre el AFD para llegar a una versión optimizada del mismo.

Como hemos comentado antes, la herramienta *Flex* se encarga de la tarea de generar un autómata finito determinista en forma de tabla, a partir del conjunto de expresiones regulares.

2.4. Implementación de un autómata finito determinista

Partiendo de la base de que el tipo de autómata a implementar va a ser un AFD, podemos escribir programas que simulen su comportamiento de varias formas:

- Simular directamente la tabla de transiciones de estados del autómata.
- Simular el diagrama de transiciones mediante sentencias `if-then-else`, `case`, etc.
- Usar la herramienta *Flex*, mediante la cual se puede generar automáticamente el código para simular el autómata.

Los dos primeros enfoques se analizaron en la asignatura *Autómatas y lenguajes formales* del primer cuatrimestre. Aquí nos centraremos, por tanto, en la tercera solución.

Como ya hemos comentado, *Flex* es una herramienta para generar analizadores léxicos a partir de un fichero que contenga especificaciones de expresiones regulares y acciones definidas por el usuario.

Flex funciona bajo entorno Linux y Windows. Su predecesora, *Lex*, bajo entornos UNIX.

2.4.1. Formato de un fichero *Flex*

El formato de un `fichero.l` es el siguiente:

```
{definiciones}
%%
{reglas}
%%
{subrutinas de usuario}
```

Las tres secciones son opcionales:

- **Definiciones**: Se especifican, entre otras cosas, macros que dan nombre a expresiones regulares auxiliares que serán utilizadas en la sección de reglas. Se puede incluir código C, condiciones de contexto, tablas, etc.
- **Reglas**: En esta sección hay secuencias del tipo $r_i \ accion_i$; dónde r_i es una expresión regular y $accion_i$ es un bloque de sentencias en C que se ejecutan cuando se reconoce el lexema asociado a esa expresión regular.
- **Subrutinas de usuario**: Procedimientos auxiliares necesarios.

2.4.1.1. Acciones de Flex

Para que Flex *devuelva un código de token* después de reconocer un lexema mediante una expresión regular, debemos poner, después de ésta, la acción correspondiente, que en este caso será:

```
return código_token;
```

Para que *ignore una secuencia de caracteres* sin devolver ningún código de token, ponemos al lado de la expresión regular correspondiente la sentencia nula:

```
;
```

Si lo que lee de la entrada no lo reconoce, por defecto lo escribe en la salida. Por tanto, si queremos filtrar la entrada completamente, debemos especificar expresiones regulares que reconozcan cualquier secuencia de caracteres de entrada.

2.4.2. Uso de la herramienta *Flex*

La creación de un analizador léxico usando *Flex* se realizaría según el esquema de la figura 2.3:

A partir de un fichero con esa estructura, que suele tener extensión `.l`, se genera otro, llamado `lex.yy.c`, que una vez compilado (con `gcc`) funciona como analizador léxico. Es decir, ejecutamos *Flex*, escribiendo:

```
flex fichero.l
```

La ejecución de este comando produce el fichero `lex.yy.c`, que contiene la función `yylex()`, con la que se reconoce, cada vez que es llamada, un lexema y se ejecuta una acción. Es decir, esta función implementaría el analizador léxico. A continuación compilamos con C para obtener el ejecutable:

```
gcc [fichero.c] lex.yy.c -lfl -o fichero
```

Los corchetes indican que, opcionalmente, se puede compilar un `fichero.c` junto con el fichero generado con *Flex*, `lex.yy.c`. Veremos un ejemplo de esto en la sección 2.4.3.

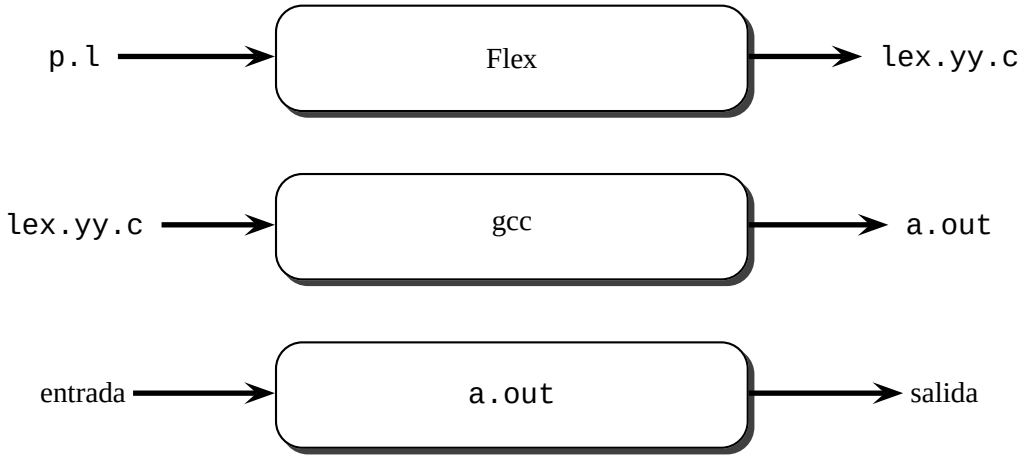


Figura 2.3: Creación de un analizador léxico con *Flex*

2.4.3. Ejemplo de analizador léxico con *Flex*

Un ejemplo de fichero en *Flex*, que especifica las expresiones regulares para un lenguaje de programación al estilo de Pascal, denominado MICRO, es el siguiente `lexico1.l`:

```
%{
#include "lexico1.h"
%}
digito              [0-9]
letra               [a-zA-Z]
entero              {digito}+
%%
[ \n\t]+                            ;
"--"(.*)[\n]                        ;
begin                               return BEGINN;
end                                 return END;
read                                return READ;
write                               return WRITE;
{letra}({letra}|{digito}|_)*        return ID;
{entero}                            return INTLITERAL;
"("                                 return LPAREN;
")"                                 return RPAREN;
";"                                 return SEMICOLON;
","                                 return COMMA;
":="                                return ASSIGNOP;
"+"                                 return PLUSOP;
"-"                                 return MINUSOP;
{entero}[.]{entero}                 return REALLIT;
.                                   error_lexico();
%%

error_lexico()
{
  printf("\nERROR, simbolo no reconocido %s\n",yytext);
```

```
}

main() {
    int i;
    while (i=yylex())
        printf("%d %s\n",i,yytext);
    printf("FIN DE ANALISIS LEXICO\n");
}
```

Como vemos, el fichero `lexico1.l` incluye una sentencia del preprocesador de C, para incluir el fichero `lexico1.h`. Éste contendría la declaración de los códigos de tokens:

```
#define             BEGINN          1
#define             END             2
#define             READ            3
#define             WRITE           4
#define             ID              5
#define             INTLITERAL      6
#define             LPAREN          7
#define             RPAREN          8
#define             SEMICOLON       9
#define             COMMA           10
#define             ASSIGNOP        11
#define             PLUSOP          12
#define             MINUSOP         13
#define             REALLIT         14
```

En este caso, para conseguir el ejecutable final, sólo es necesario ejecutar la siguiente secuencia de comandos:

```
flex lexico1.l
gcc lex.yy.c -lfl -o lexico1
```

Como hemos comentado antes, también se puede crear el analizador léxico compilando de forma separada el fichero generado por *Flex* y algún fichero en C que hayamos creado de forma independiente. Un ejemplo podría ser el siguiente `lexico2.l`, que una vez procesado con *Flex*, generaría el fichero `lex.yy.c`. Este fichero se compilaría con `main.c` para obtener el ejecutable final. `lexico2.l` incluiría también el fichero de cabecera con la declaración de tokens, `lexico1.h`, pero vemos que, en este caso, la sección de rutinas de usuario está vacía:

```
%{
#include "lexico1.h"
%}
digito              [0-9]
letra               [a-zA-Z]
entero              {digito}+

%%
[ \n\t]+                                    ;
"--"(.*)[\n]                                ;
begin                                       return BEGINN;
end                                         return END;
read                                        return READ;
write                                       return WRITE;
{letra}({letra}|{digito}|_)*                return ID;
```

```
{entero}                                    return INTLITERAL;
"("                                         return LPAREN;
")"                                         return RPAREN;
";"                                         return SEMICOLON;
","                                         return COMMA;
":="                                        return ASSIGNOP;
"+"                                         return PLUSOP;
"-"                                         return MINUSOP;
{entero}[.]{entero}                         return REALLIT;
.                                           printf("Error en carácter %s",yytext
%%
```

En el fichero `main.c` aparecerán las rutinas de usuario necesarias. En este caso se reducen a una función `main()` algo más elaborada que la de `lexico1.l`:

```
#include <stdio.h>
#include "lexico1.h"

extern char *yytext;
extern int  yyleng;
extern FILE *yyin;
FILE *fich;
main()
{
    int i;
    char nombre[80];
    printf("INTRODUCE NOMBRE DE FICHERO FUENTE:");
    gets(nombre);
    printf("\n");
    if ((fich=fopen(nombre,"r"))==NULL) {
        printf("***ERROR, no puedo abrir el fichero\n");
        exit(1); }
    yyin=fich;
    while (i=yylex()) {
        printf("TOKEN %d",i);
        if(i==ID) printf("LEXEMA %s  LONGITUD %d\n",yytext,yyleng);
        else printf("\n");}
    fclose(fich);
    return 0;
}
```

Para obtener el ejecutable con esta segunda opción, pondríamos:

```
flex lexico2.l
gcc main.c lex.yy.c -lfl -o lexico2
```

2.5. Interfaz del analizador léxico

La simulación de un autómata finito determinista, tal y como se ha visto en la sección anterior, pasa por el uso de dos rutinas de entrada: una primera para leer carácter a carácter el programa de entrada, y otra segunda para devolver, en caso de que sea necesario, un carácter usado como símbolo de anticipación.

Unos mínimos conocimientos sobre el funcionamiento de los sistemas operativos nos permiten comprender que es muy ineficiente el acceso a la entrada para realizar una lectura uno a uno de los caracteres del programa fuente. Por el contrario, es mucho más eficiente la lectura del fichero de entrada en bloques de tamaño equivalente al de los bloques de disco (por ejemplo 4096 bytes).

Por tanto, un compilador debe hacer uso de un nivel de buffers por encima del esquema de entrada que nos ofrece el sistema operativo. Por otra parte, el diseño de este mecanismo de buffers debe realizarse con cuidado, de modo que permita el uso de secuencias largas de caracteres de anticipación.

En el caso de usar la herramienta *Flex*, ésta proporciona el acceso a variables y funciones para que el usuario disponga de algo de control en la entrada/salida. Por ejemplo, podemos usar las variables globales `yytext`, `yyleng` e `yylineno`, y las funciones `yymore()`, `yyless(n)` e `yywrap()`.

- `yytext` es un puntero a la última cadena de texto que se ajustó al patrón de una expresión regular.
- `yyleng` es una variable global que contiene la longitud de `yytext`.
- Activando la opción *yylineno*[1] en la sección de declaraciones, flex genera un analizador que mantiene el número de la línea actual leída desde su entrada en la variable global `yylineno`.
- `yymore()` sirve para indicar que la próxima vez que se empareje una regla, el valor nuevo debería ser concatenado al valor que contiene `yytext` en lugar de reemplazarlo.
- `yyless(n)` permite retrasar el puntero de lectura, de manera que apunta al carácter `n` de `yytext`.
- `yywrap()` es una función función que se ejecuta cada vez que se alcanza el final del fichero.

Además, podemos cambiar la entrada y salida estándar haciendo uso de los punteros `yyin` e `yyout` y de las funciones `input()`, `output(c)` y `unput(c)`.

- `yyin` es un puntero al descriptor de fichero de dónde se leen los caracteres.
- `yyout` es un puntero al fichero en el que escribe el analizador léxico al utilizar la opción ECHO.
- `input()` lee desde el flujo de entrada el siguiente carácter.
- `output(c)` escribe el carácter `c` en la salida.
- `unput(c)` coloca el carácter `c` en el flujo de entrada, de manera que será el primer carácter leído en próxima ocasión.

2.6. Manejo de errores léxicos

El analizador léxico detectará un error cuando no exista una transición válida en el autómata finito para el carácter de entrada desde el estado actual. Cuando esto suceda, se debe informar del error e intentar recuperarse para continuar el análisis.

Para realizar una recuperación de errores usando la herramienta *Flex*, podríamos pensar en añadir al final del conjunto de expresiones regulares, una última:

```
.       imprime_mensaje_de_error()
```

Si embargo, el analizador generado imprimiría un mensaje de error por cada carácter erróneo que apareciera en la entrada. Esta recuperación no sería adecuada. Pensemos, por ejemplo, en la entrada

```
abc#########de
```

[1] `%option yylineno`

Aunque no existen métodos generales que funcionen bien en todos los casos, algunos de ellos se comportan de forma aceptable y son bastante eficientes. Por ejemplo:

- Recuperación en *modo pánico*: este tipo de estrategia es la más común. Consiste en ignorar caracteres extraños hasta encontrar un carácter válido para un nuevo token.

  Para implementar el *modo pánico* en *Flex* será necesario pensar en una expresión regular capaz de reconocer un conjunto de caracteres erróneos previos al comienzo de un nuevo token.

- Borrar un carácter extraño.
- Insertar un carácter que falta (por ejemplo, reemplazar `2C` por `2*C`).
- Reemplazar un carácter incorrecto por otro correcto (por ejemplo, reemplazar `intager` por `integer` si el lugar en donde aparece el primer lexema no es el indicado para un identificador).
- Intercambiar dos caracteres adyacentes.
- Encontrar el número menor de transformaciones del tipo indicado anteriormente que convierta el programa fuente en otro que no tenga errores léxicos. Esta aproximación es muy costosa y no se suele usar en la práctica.

La recuperación de errores en modo pánico durante el análisis léxico puede producir otros errores en las siguientes fases. Por ejemplo, con el siguiente programa:

```
1 var numero : integer;
2 begin
3    num?ero:=10;
4 end
```

el compilador podría producir los siguientes mensajes de error referidos a la línea 3:

- Error léxico: carácter no reconocido (`?`)
- Error semántico: identificador no declarado (`num`)
- Error sintáctico: falta operador entre identificadores
- Error semántico: identificador no declarado (`ero`)

En otras ocasiones, sin embargo, la recuperación en modo pánico no conlleva efectos en otros ámbitos, como sería el caso del siguiente programa, en el que se generaría un solo aviso de error léxico (semejante al primero del ejemplo anterior) en la línea 4:

```
1 var i,j: integer;
2 begin
3    i:=1;
4    ?
5    j:=2;
6 end
```

Es importante tener en cuenta que existen errores léxicos que no son recuperables, como ocurre si no se cierra un comentario.

2.7. Ejemplo de diseño

En esta sección se siguen los pasos expuestos en las secciones anteriores para el diseño de un analizador léxico de un sencillo lenguaje de programación denominado *MICRO-C*, cuyas especificaciones básicas son las que siguen. El diseño con *Flex* se deja como ejercicio para el laboratorio de prácticas.

- Todos los programas comienzan con: `main()`.
- Sólo se define un tipo de dato: entero.
- No se declaran variables.
- Los identificadores empiezan por letra, _ o $, y se componen de letras, dígitos, y los símbolos _ y $.
- Sólo hay constantes enteras.
- Los comentarios comienzan por `//` y terminan con un carácter de fin de línea (EOL).
- Se pueden usar las siguientes sentencias:
  - `ID = expresión;`
  - `print ( lista de expresiones entre comas);`
- El lenguaje tiene las siguientes palabras reservadas: `main`, `print`.
- Las sentencias se separan con `;`.
- El cuerpo del programa está delimitado por los caracteres `{` y `}`.
- Los separadores de tokens son: espacios en blanco, tabuladores y EOL.

Partiendo de la especificación anterior, se pueden definir las siguientes expresiones regulares:

$$
\begin{aligned}
\text{ID} &\equiv (A|\$|\_)(A|D|\$|\_)^* \\
\text{MAIN} &\equiv \text{main} \\
\text{PRINT} &\equiv \text{print} \\
\text{INTLITERAL} &\equiv D^+ \\
\text{PARENI} &\equiv (\\
\text{PAREND} &\equiv) \\
\text{LLAVEI} &\equiv \{ \\
\text{LLAVED} &\equiv \} \\
\text{SEMICOLON} &\equiv ; \\
\text{COMMA} &\equiv , \\
\text{ASSIGNOP} &\equiv = \\
\text{PLUSOP} &\equiv + \\
\text{MINUSOP} &\equiv - \\
\text{SCANEOF} &\equiv \text{EOF} \\
\text{BLANCOS} &\equiv (\ |\backslash n|\backslash t) \\
\text{COMMENT} &\equiv //C^*\text{EOL} \\
A &\equiv a|b|\ldots|z|A|B|\ldots|Z \\
D &\equiv 0|1|\ldots|9 \\
C &\equiv \text{cualquier caracter excepto EOL}
\end{aligned}
$$

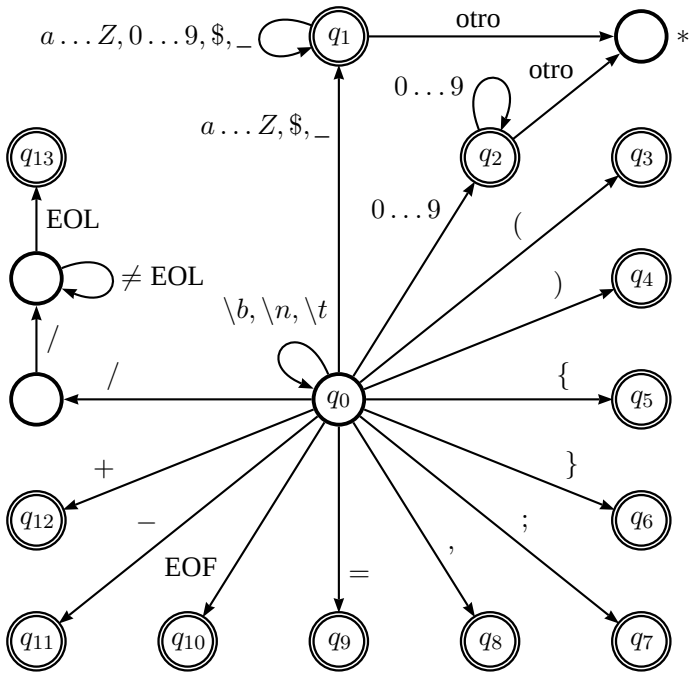
Cada expresión regular da lugar a un autómata finito. El estado final de cada uno de estos autómatas representa el reconocimiento de un lexema correspondiente al patrón descrito por la expresión regular. Si combinamos todos estos autómatas en uno solo, compartiendo el estado inicial, se obtiene un autómata global que reconoce todos los patrones. Es interesante tener en cuenta varias consideraciones:

- La expresión regular correspondiente a los espacios en blanco no da lugar a un estado final, puesto que habitualmente no interesa considerar los espacios en blanco en el análisis sintáctico. Por ello, cuando

se lea un espacio en blanco en el estado inicial del autómata, simplemente se consumirá de la entrada y se continuará el análisis desde el estado inicial.

- La expresión D^+ representa la *clausura positiva* de D, es decir, $L(D^+) \equiv L(D^*) - \{\lambda\}$.
- Las expresiones regulares de ID e INTLITERAL requieren el uso de un carácter de anticipación. En general esto sucede cuando los lexemas de un token pueden ser prefijos de los lexemas de otro token, o bien cuando se emplean expresiones regulares que finalizan con clausuras de subconjuntos del alfabeto. Esto último sucede en este ejemplo. El carácter de anticipación sirve para comprobar si el lexema se puede seguir ampliando con caracteres de la entrada que forman parte del subconjunto sobre el que se aplica la clausura, o por el contrario se ha encontrado un carácter fuera de dicho subconjunto que marca el final del lexema. En este último caso, el carácter de anticipación se tiene que devolver a la entrada. En el autómata, esta devolución se puede representar con un estado especial.

El autómata completo se muestra a continuación:



Seguidamente se muestra el código de simulación del diagrama de transiciones. El código está dividido en dos ficheros: `lexico.h` y `lexico.c`.

El primer fichero contiene las declaraciones necesarias para hacer uso del analizador léxico. Este fichero puede ser empleado en la implementación del analizador sintáctico. El fichero define los códigos usados para representar los distintos tokens del lenguaje, así como la función para obtener el siguiente token, llamada `scanner()`, y la variable que contiene el lexema del token `token_buffer`:

lexico.h

```
1 typedef enum token_types {
2    MAIN, PRINT, ID, INTLITERAL, PARENI, PAREND, LLAVEI,
3    LLAVED, SEMICOLON, COMMA, ASSIGNOP, PLUSOP, MINUSOP,
4    SCANEOF } token;
5 extern token scanner(void);
6 extern char token_buffer[];
```

El segundo fichero contiene el código de la función que simula el autómata:

lexico.c

```
#include "lexico.h"
#include <stdio.h>
#include <ctype.h>

void buffer_char(char c);
void clear_buffer(void);
token check_reserved(void);
void lexical_error(char c);
char token_buffer[30];

token scanner(void) {
   int in_char, c;
   clear_buffer();
   if (feof(stdin))
         return SCANEOF;
   while ((in_char = getchar()) != EOF) {
      /* Espacios en blanco */
      if (isspace(in_char))
         continue; /* no se hace nada */
      /* ID */
      else if (isalpha(in_char) || in_char == '_' || in_char == '$') {
         buffer_char(in_char);
         for (c = getchar(); isalnum(c) || c == '_' || c == '$'; c = getchar())
            buffer_char(c);
         ungetc(c,stdin);
         return check_reserved();
      }
      /* INTLITERAL */
      else if (isdigit(in_char)) {
         buffer_char(in_char);
         for (c = getchar(); isdigit(c); c = getchar())
            buffer_char(c);
         ungetc(c,stdin);
         return INTLITERAL;
      }
      /* PARENI */
      else if (in_char == '(')
         return PARENI;
      /* PAREND */
      else if (in_char == ')')
         return PAREND;
      /* LLAVEI */
      else if (in_char == '{')
         return LLAVEI;
      /* LLAVED */
      else if (in_char == '}')
         return LLAVED;
      /* SEMICOLON */
      else if (in_char == ';')
         return SEMICOLON;
      /* COMMA */
      else if (in_char == ',')
         return COMMA;
      /* PLUSOP */
      else if (in_char == '+')
         return PLUSOP;
      /* ASSIGNOP */
```

```
58        else if (in_char == '=')
59           return ASSIGNOP;
60        /* MINUSOP */
61        else if (in_char == '-')
62           return MINUSOP;
63        /* COMMENT */
64        else if (in_char == '/') {
65           c = getchar();
66           if (c == '/') {
67              do
68                 in_char = getchar();
69              while (in_char != '\n');
70           }
71           else {
72              ungetc(c,stdin);
73              lexical_error(in_char);
74           }
75        }
76        /* Error */
77        else
78           lexical_error(in_char);
79     }
80  }
```

El código anterior hace uso de una serie de funciones cuyo código no se muestra aquí. A continuación se describen brevemente:

- `void buffer_char(char c)`: almacena en la siguiente posición libre de `token_buffer` el carácter que se pasa como argumento. Es la función que se emplea para ir construyendo el lexema del token actual.
- `token check_reserved(void)`: comprueba si el lexema almacenado en `token_buffer` corresponde a alguna palabra reservada del lenguaje. En tal caso, se devuelve el código de dicha palabra. Por el contrario, si no corresponde a ninguna palabra reservada, se devuelve el código de token `ID`.
- `void lexical_error(char c)`: genera un mensaje por salida estándar para informar de que se ha producido un error léxico al procesar el carácter `c`.
- `void clear_buffer(void)`: borra el contenido del array de caracteres `token_buffer`.

El código también emplea algunas funciones de la librería estándar de C:

- `int isspace(int c)`: comprueba si el carácter `c` es uno de los espacios en blanco.
- `int isalpha(int c)`: comprueba si el carácter `c` es alfabético, en mayúsculas o minúsculas.
- `int isalnum(int c)`: comprueba si el carácter `c` es alfanumérico.
- `int isdigit(int c)`: comprueba si el carácter `c` es un dígito.