Análisis léxico

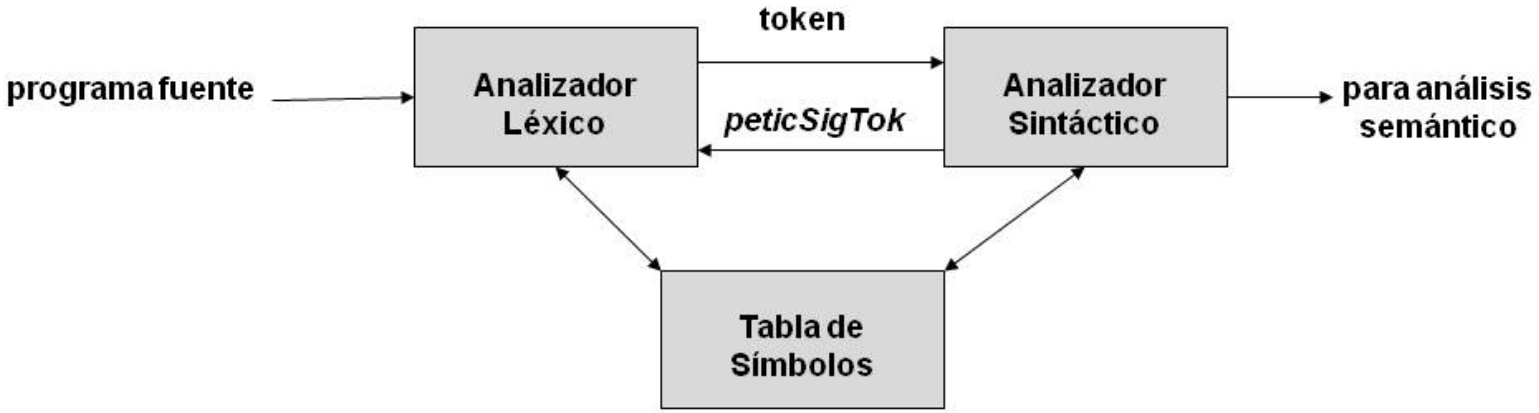
Dpto. de Ingeniería de la Información y las Comunicaciones

Facultad de Informática
UNIVERSIDAD DE MURCIA

Índice: secciones principales

Objetivo del analizador léxico

- Lee los **caracteres de entrada** y elabora una secuencia de **tokens**, que serán los *símbolos terminales* de la gramática.
- Puesto que lee del texto fuente, realiza funciones secundarias en la interfaz de usuario (eliminar comentarios, blancos, ...).
- Guarda información sobre algunos componentes léxicos en forma de **atributos**.
  - **Token** : nombre + atributo (opcional).
  - **Patrón** : descripción de la forma que los lexemas de un token pueden tomar.
  - **Lexema** : secuencia de caracteres en el programa fuente que concuerda con el patrón de un token.



Objetivo del analizador léxico

- *Detecta*, *recupera* y, a veces, intenta *reparar* **errores léxicos**, aunque en este nivel son pocos los que se pueden detectar.
  - De entre las técnicas para **recuperación de errores** destaca el "*modo pánico*", por su generalidad y eficiencia.
  - Como técnica de **corrección de errores** la "*corrección de errores de distancia mínima*" intenta obtener el programa más parecido al original pero sin errores léxicos.
- Relaciona **mensajes de error** del compilador con el programa fuente.

La separación de la fase de análisis de la compilación en análisis léxico y análisis sintáctico puede simplificar alguna de dichas fases, mejorar la eficiencia del compilador y su transportabilidad.

Expresiones regulares y autómatas finitos

Expresiones regulares

Dado un alfabeto V, los símbolos $\emptyset$ (vacío), λ (lambda) y las operaciones de $|$ (unión), $\circ$ (concatenación) y $*$ (cierre o clausura), se cumple que:

1. El símbolo $\emptyset$ es una e.r. que describe el lenguaje $L(\emptyset) = \emptyset$.
2. El símbolo λ es una e.r. que describe el lenguaje $L(\lambda) = \{\lambda\}$
3. Cualquier símbolo $a_i \in$ V es una e.r. que describe el lenguaje $L(a_i) = \{a_i\}$
4. Si R_1 y R_2 son e.r., entonces también lo es $R_1|R_2$, que describe el lenguaje $L(R_1|R_2) = L(R_1) \cup L(R_2)$.
5. Si R_1 y R_2 son e.r., entonces también lo es $R_1 \circ R_2$, que describe el lenguaje $L(R_1 \circ R_2) = L(R_1) \circ L(R_2)$.
6. Si R es una e.r entonces R^* también lo es y describe al lenguaje $L(R^*) = L(R)^*$.
7. Si R es una e.r entonces (R) también lo es y describe al lenguaje $L((R)) = (L(R))$.
8. No hay más que las anteriores.

Expresiones regulares y autómatas finitos

Autómata Finito Determinista (AFD)

Es una quíntupla $\{Q, V, \delta, q_0, F\}$ donde :

- Q es un conjunto finito de estados.
- V es un alfabeto.
- $q_0 \in Q$ es el estado inicial.
- $F \subseteq Q$ es el conjunto de estados finales.
- $\delta : Q \times V \rightarrow Q$ es la función de transición.

Autómata Finito No Determinista (AFND)

Es una quíntupla $M = (Q, V, \Delta, q_0, F)$ donde todos los componentes coinciden con los de la definición de AFD excepto la función de transición que se define de la siguiente forma:

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

Diseño de un analizador léxico

Deben realizarse las siguientes **tareas**:

1. **Especificación y reconocimiento de componentes léxicos**. Se identifican los tokens del lenguaje utilizando *expresiones regulares*. **Describimos:**
   - Los lexemas que vamos a reconocer como tokens.
   - Los lexemas que ignoramos.
2. **Construcción del autómata finito que reconozca las e.r. anteriores**. Existen distintas técnicas y alternativas.
3. **Implementación del AFD**.
4. **Diseño de la interfaz entre la entrada y el AL**.
5. **Diseño de la interfaz entre el AL y el AS**. Debe devolver el token reconocido y la información relativa a él (que se almacenará en la tabla de símbolos).
6. **Especificación de la estrategia de manejo de errores léxicos**.
7. **Diseño de la tabla de símbolos**. Aunque puede inicializarse aquí, generalmente no es una tarea propia del AL.

1 Especificación y reconocimiento de componentes léxicos

Los tokens se pueden *describir* mediante **expresiones regulares** y se pueden *reconocer* mediante **autómatas finitos**.

Una expresión regular para un token describe todos los lexemas que dicho token puede tener asociados.

Ejemplo: E.R. que describe los identificadores en Java

$$(L|\_|\$)(L|\_|\$|D)*$$
$$\text{con } L: \quad a|b|...|z|A|...|Z$$
$$\text{y } D: \quad 0|1|2|...|9$$

Criterios para identificar tokens

→ Intentamos establecer todas las cadenas del l.f. con significado propio:

- Palabras clave.
- Operadores y signos de puntuación.
- Identificadores.
- Constantes enteras, reales y de tipo carácter. Cadenas de caracteres.

→ Por simplicidad interesan: *tokens sencillos* y *lexemas independientes*.

1 Especificación y reconocimiento de componentes léxicos

Expresiones Regulares de 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.

1 Especificación y reconocimiento de componentes léxicos

Expresiones Regulares de Flex

- 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.

1 Especificación y reconocimiento de componentes léxicos

Expresiones Regulares de Flex

Precedencia de Operadores

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

$* \; ? \; +$

concatenación

repetición $\{\}$

$\$ \; \hat{}$

$|$

$/$

1 Especificación y reconocimiento de componentes léxicos

Problemas que pueden aparecer a la hora de reconocer tokens:

- **Caracteres de anticipación**

  A veces necesitamos leer uno o más caracteres "extra" de la entrada. **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)*.

- **Comentarios** deben ser reconocidos y eliminados.

- **Blancos**

  Delimitadores: Los elimina.

  No delimitadores: Los elimina y agrupa lexemas.

- **Token EOF** Es interesante, de cara al analizador sintáctico, tenerlo como token.

1 Especificación y reconocimiento de componentes léxicos

Problemas que pueden aparecer a la hora de reconocer tokens:

- **Palabras clave**

  Si **no** son **reservadas**, el analizador léxico las reconoce como identificadores, y es el sintáctico quien debe identificarlas.

  Si son **reservadas** debe reconocerlas como palabras clave mediante una **tabla de palabras reservadas** o un **autómata finito**.

  En este caso, 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.

Diseño de un analizador léxico

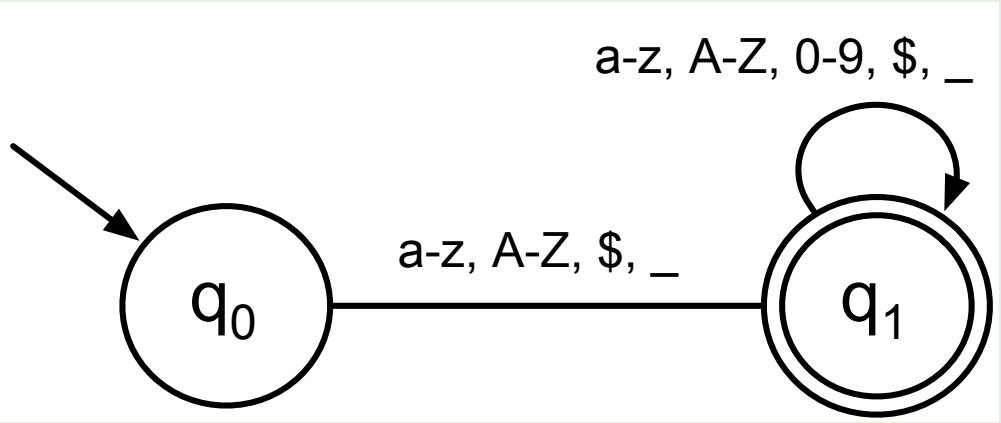
Deben realizarse las siguientes **tareas**:

1. **Especificación y reconocimiento de componentes léxicos**. Se identifican los tokens del lenguaje utilizando *expresiones regulares*.
2. **Construcción del autómata finito que reconozca las e.r. anteriores**. Existen distintas técnicas y alternativas.
3. **Implementación del AFD**.
4. **Diseño de la interfaz entre la entrada y el AL**.
5. **Diseño de la interfaz entre el AL y el AS** Debe devolver el token reconocido y la información relativa a él (que se almacenará en la tabla de símbolos).
6. **Especificación de la estrategia de manejo de errores léxicos**.
7. **Diseño de la tabla de símbolos**. Aunque puede inicializarse aquí, generalmente no es una tarea propia del AL.

2 Construcción del autómata finito

Los autómatas finitos se construyen a partir de las expresiones regulares que describen los tokens.

Ejemplo: Autómata que reconoce los identificadores en Java



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

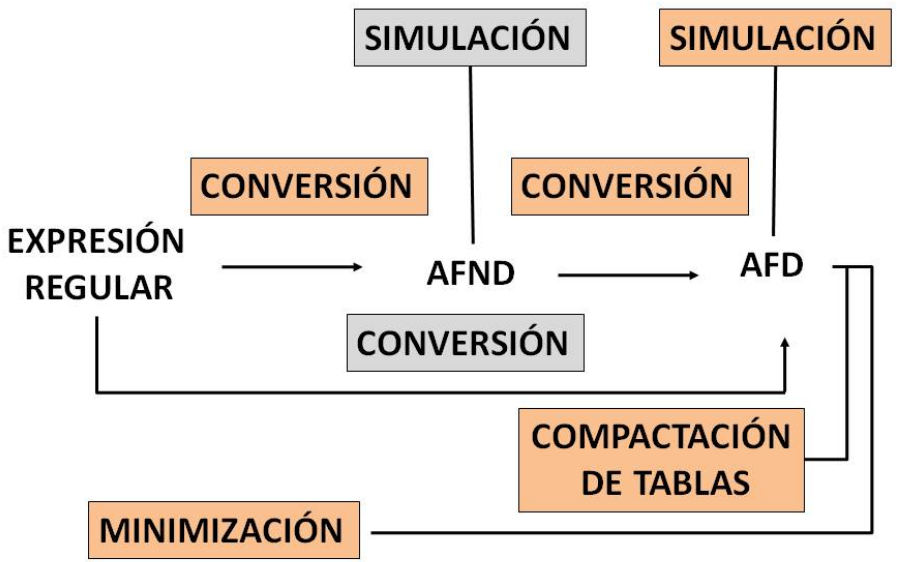
→ Para simular un autómata finito con un programa suele interesar que éste sea determinista y mínimo.

Flex se encarga de la tarea de generar un autómata finito determinista en forma de tabla, a partir del conjunto de e.r.

2 Construcción del autómata finito

Alternativas sin Flex

- La conversión de una e.r. en AFND es más directa que la conversión en AFD.
- Podemos simular un AFND, pero es más rápido convertir el AFND en un AFD y después hacer la simulación del AFD.
- Otra opción es construir directamente un AFD a partir de una expresión regular sin hacer el AFND intermedio.
- Pueden aplicarse algoritmos para **minimizar el número de estados de un AFD** y **producir representaciones rápidas y compactas** de la tabla de transiciones.



Diseño de un analizador léxico

Deben realizarse las siguientes **tareas**:

1. **Especificación y reconocimiento de componentes léxicos**. Se identifican los tokens del lenguaje utilizando *expresiones regulares*.
2. **Construcción del autómata finito que reconozca las e.r. anteriores**. Existen distintas alternativas.
3. **Implementación del AFD**.
4. **Diseño de la interfaz entre la entrada y el AL**.
5. **Diseño de la interfaz entre el AL y el AS** Debe devolver el token reconocido y la información relativa a él (que se almacenará en la tabla de símbolos).
6. **Especificación de la estrategia de manejo de errores léxicos**.
7. **Diseño de la tabla de símbolos**. Aunque puede inicializarse aquí, generalmente no es una tarea propia del AL.

3 Implementación de un autómata finito

A partir de un AFD podemos escribir un programa que lo simule:

1. A partir de la **tabla de transición**.
2. Implementando el **diagrama de transición** con sentencias de selección.
3. Usando la **herramienta Flex**.

- 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 (Lex funciona bajo entornos UNIX).

3 Implementación de un autómata finito con Flex

Formato de un fichero.l :

{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.

3 Implementación de un autómata finito con Flex

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.

3 Implementación de un autómata finito con Flex

Uso de Flex

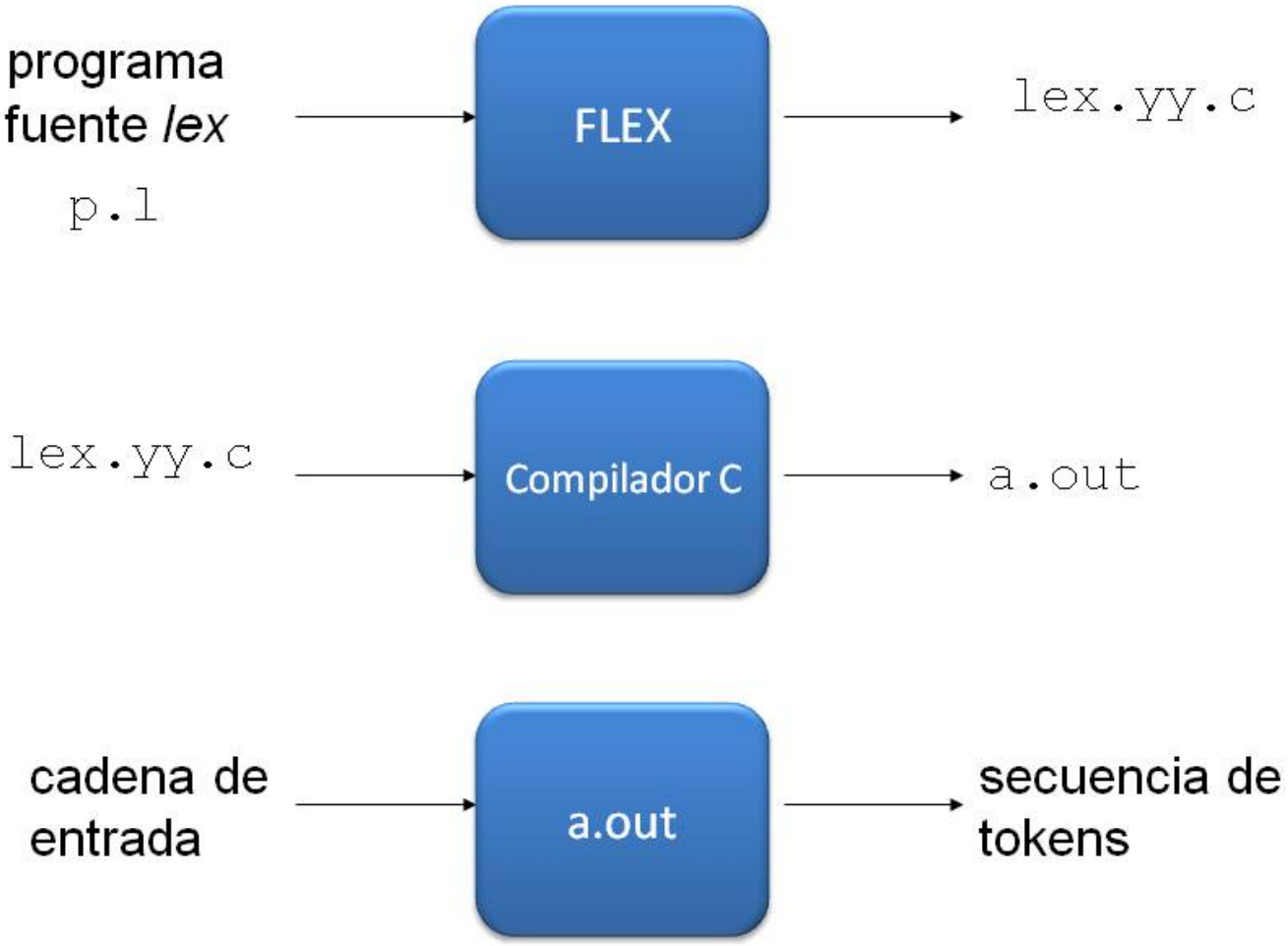
- A partir de un fichero, que suele tener extensión .l, genera otro, llamado lex.yy.c, que contiene la función `yylex()`, con la que se reconoce, cada vez que es llamada, un lexema y se ejecuta una acción.
- La creación de un analizador léxico usando flex se realizaría con los siguientes comandos:

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

según el siguiente esquema:

3 Implementación de un autómata finito con Flex

Esquema del analizador Léxico



Ejemplos: lexico1.h

```
#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
```

Ejemplo 1: 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, símbolo no reconocido %s\n",yytext);
}

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

Ejemplo 2: lexico2.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;
.                                           printf("Error en carácter %s",yytext );
%%
```

Ejemplo 2: main.c

```
#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;
}
```

Diseño de un analizador léxico

Deben realizarse las siguientes **tareas**:

1. **Especificación y reconocimiento de componentes léxicos**. Se identifican los tokens del lenguaje utilizando *expresiones regulares*.
2. **Construcción del autómata finito que reconozca las e.r. anteriores**. Existen distintas alternativas.
3. **Implementación del AFD**.
4. **Diseño de la interfaz entre la entrada y el AL**.
5. **Diseño de la interfaz entre el AL y el AS** Debe devolver el token reconocido y la información relativa a él (que se almacenará en la tabla de símbolos).
6. **Especificación de la estrategia de manejo de errores léxicos**.
7. **Diseño de la tabla de símbolos**. Aunque puede inicializarse aquí, generalmente no es una tarea propia del AL.

4-5 Interfaz del analizador léxico

La **entrada** suele hacerse desde **fichero** → Se requiere una comunicación adecuada con él.

La lectura puede realizarse:

- **Carácter a carácter** → *devolución de uno o varios caracteres* ineficiente.
- **Por líneas o bloques** → menos llamadas *a funciones E/S* y *devolución* más eficiente.

4-5 Interfaz del analizador léxico

Con Flex:

→ Podemos usar las variables globales *yytext*, *yyleng* e *yylineno*, y las funciones *yymore()*, *yyless(n)* e *yywrap()*.

- *yytext*: puntero a la última cadena de texto que se ajustó al patrón de una expresión regular.
- *yyleng*: variable global que contiene la longitud de *yytext*.
- *yylineno*: 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()*: función que se ejecuta cada vez que se alcanza el final del fichero.

[1] `%option yylineno`

4-5 Interfaz del analizador léxico

Con Flex:

→ 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* puntero al descriptor de fichero de dónde se leen los caracteres.
- *yyout* 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.

Diseño de un analizador léxico

Deben realizarse las siguientes **tareas**:

1. **Especificación y reconocimiento de componentes léxicos**. Se identifican los tokens del lenguaje utilizando *expresiones regulares*.
2. **Construcción del autómata finito que reconozca las e.r. anteriores**. Existen distintas alternativas.
3. **Implementación del AFD**.
4. **Diseño de la interfaz entre la entrada y el AL**.
5. **Diseño de la interfaz entre el AL y el AS** Debe devolver el token reconocido y la información relativa a él (que se almacenará en la tabla de símbolos).
6. **Especificación de la estrategia de manejo de errores léxicos**.
7. **Diseño de la tabla de símbolos**. Aunque puede inicializarse aquí, generalmente no es una tarea propia del AL.

6 Manejo de errores

El *analizador léxico* debe ser capaz de **detectar** los **errores léxicos** y:

- Debe **informar**.
- Sería deseable que intentara **recuperarse**.

El "*scanner*" detectará un error léxico cuando no exista transición válida en el autómata finito para el carácter de la entrada.

Para **realizar una recuperación de errores usando 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
```

6 Manejo de errores

Para recuperación no existen métodos generales que funcionen siempre bien.

- "**Modo pánico**" → *Se ignoran 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.
- Otras técnicas **más sofisticadas, pero más costosas** de implementar:
  - Borrar un carácter extraño.
  - Insertar un carácter perdido.
  - Reemplazar un carácter por otro correcto.
  - Intercambiar caracteres adyacentes.

6 Manejo de errores

→ La recuperación de errores durante el A.L. puede producir otros en las siguientes fases

Ejemplo

La recuperación provoca nuevos errores:

```
int numero;
{
    num ?ero = 10;
}
```

La recuperación no provoca nuevos errores:

```
int i,j;
{
    i = 1;
    ?
    j = 2;
}
```

→ Hay errores que no son recuperables.

Ejemplo de diseño de un analizador léxico

Especificación del lenguaje

- Todos los programas comienzan con '`main ()`'.
- **Tipo de dato**: entero.
- No se declaran **variables**.
- Los **identificadores** empiezan por letra, '_' o '$' y se componen de letra, '_', '$' o dígito.
- Solo hay **constantes** enteras.
- Los **comentarios** comienzan por '//' y terminan con *new-line*.
- **Sentencias**:
  - `ID=expresion;`
    para asignaciones de expresiones aritméticas con '+', '−' y paréntesis.
  - `print ( lista de expresiones entre comas );`
- **Palabras reservadas**: '`main`', '`print`'.
- Las sentencias se separan con ';'.
- El cuerpo del programa está delimitado entre '{' y '}'.
- Separadores de tokens: espacios en blanco, tabuladores y *new-line*.
- Los **caracteres especiales**: '(', ')', '=', ';', '+', '−', ',', '{' y '}'.

Ejemplo de diseño de un analizador léxico

1 Especificación y reconocimiento de componentes léxicos

Expresiones regulares

ID $\equiv (L|\$|\_)(L|D|\$|\_)^*$
MAIN $\equiv$ main
PRINT $\equiv$ print
INTLITERAL $\equiv$ D$^+$
PARENI $\equiv$ (
PAREND $\equiv$)
LLAVEI $\equiv$ {
LLAVED $\equiv$ }

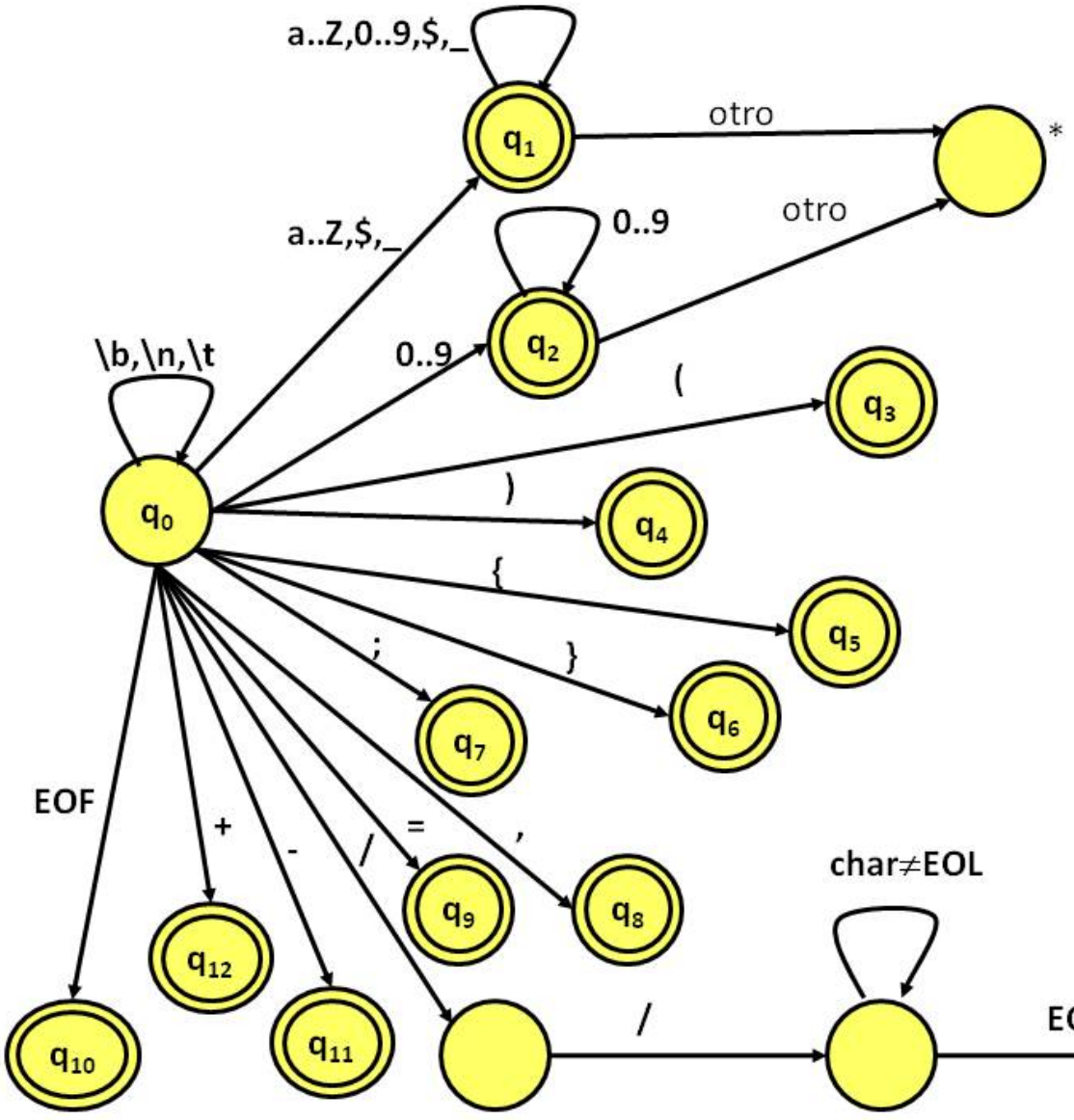
SEMICOLON $\equiv$;
COMMA $\equiv$,
ASSIGNOP $\equiv$ =
PLUSOP $\equiv$ +
MINUSOP $\equiv$ -
COMENT $\equiv$ // C^* EOL
BLANCOS $\equiv$ (|\n|\t)
SCANEOF $\equiv$ EOF

Siendo:
$L \equiv$ a | b |... | z | A | ... | Z
$D \equiv$ 0 | 1 |...| 9
C cualquier carácter excepto fin de línea.

Se necesitan caracteres de anticipación!!

Ejemplo de diseño de un analizador léxico

2 Autómata finito



Palabras clave reservadas pero no en el autómata!!

Ejemplo de diseño de un analizador léxico

3 Implementación del autómata

→ Mediante **sentencias de selección**.

4 Interfaz entre la entrada y el "scanner"

→ Se usa la entrada estándar (`stdin`).

→ Se llama a la función `getchar` para leer un carácter.

→ Los *caracteres de anticipación* se devuelven a la entrada con `unget`.

5 Interfaz entre el "scanner" y el "parser"

→ La función `scanner` devuelve el código de token.

→ Los caracteres leídos para ID o INTLITERAL se van insertando, mediante la función `buffer_char`, en `token_buffer`, que contendrá el lexema.

- La función `check_reserved` comprueba si `token_buffer` contiene el lexema de palabra clave o ID, y devuelve el código.
- La función `clear_buffer` vacía el buffer para un nuevo token.

6 Manejo de errores

→ La función `lexical_error` informa del error, al producirse, y continúa el A.L. con el siguiente carácter.

Ejemplo de diseño de un analizador léxico (1)

Código

"lexico.h"

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

"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];
```

...continúa

Ejemplo de diseño de un analizador léxico (2)

Código *...continuación*

```
token scanner(void) {
  int in_char,c;
   clear_buffer ();
  if (feof(stdin)) return SCANEOF;
  while ((in_char = getchar())!= EOF) {
    if (isspace(in_char))
      continue; /*do nothing*/
    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();
    } else if (isdigit(in_char)) {
       buffer_char(in_char);
      for(c=getchar(); isdigit(c);c=getchar())
         buffer_char(c);
      ungetc(c,stdin);
      return INTLITERAL;
```

Ejemplo de diseño de un analizador léxico (3)

Código

...continuación

```
} else if (in_char == '(')
    return PARENI;
else if (in_char == ')')
    return PAREND;
else if (in_char == '{')
    return LLAVEI;
else if (in_char == '}')
    return LLAVED;
else if (in_char == ';')
    return SEMICOLON;
else if (in_char == ',')
    return COMMA;
else if (in_char == '+')
    return PLUSOP;
else if (in_char == '=')
    return ASSIGNOP;
else if (in_char == '-')
    return MINUSOP;
else if (in_char == '/')  {
    c = getchar();
    if (c == '/') {
        do
            in_char = getchar();
        while (in_char != '\n');
    } else {
        ungetc(c, stdin);
        lexical_error (in_char);
    }
}
else
        lexical_error (in_char);
  } // while
} // scanner
```

Ejemplo de diseño de un analizador léxico

Ejercicio para discutir

¿Qué implicaciones, en cuanto al diseño, tendría el hecho de incluir en el lenguaje anterior el operador de división: '/'?

- Añadiríamos la expresión regular correspondiente:

$$\texttt{DIVOP} \equiv \texttt{/}$$

- En el diseño de las expresiones regulares y el autómata, tendríamos que tener en cuenta que sería necesario el "*lookahead*" en el reconocimiento de este token.
- Esto se traduciría en una modificación del código anterior:

```
else if (in_char == '/') {
  c = getchar();
  if (c == '/') {
    do
      in_char = getchar();
    while (in_char != '\n');
  } else {
    ungetc(c, stdin );
    lexical_error (in_char );
  }
}
```