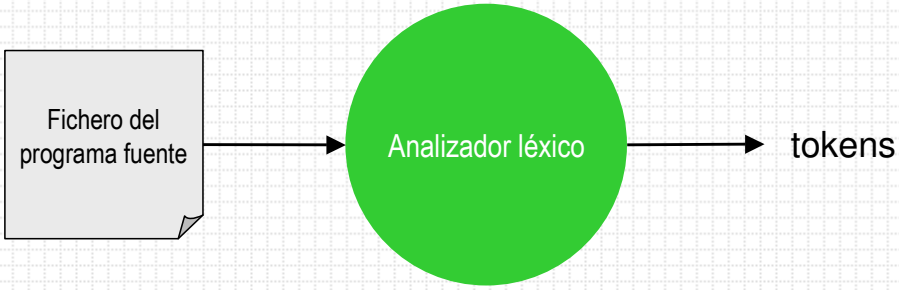
Índice (I)

Marina de la Cruz
Alfonso Ortega

Índice (II)

Funciones del analizador léxico/morfológico

- El analizador léxico/morfológico de un compilador es el responsable de **identificar en el fichero del programa fuente las unidades sintácticas o tokens** del lenguaje particular para el que ha sido construido.



- Además de identificar los tokens, también se encarga de otras tareas como:
  - Eliminar/ignorar espacios en blanco (blancos, tabuladores y saltos de línea)
  - Eliminar/ignorar comentarios
  - Detectar errores morfológicos (símbolo no permitido, identificador demasiado largo, etc)

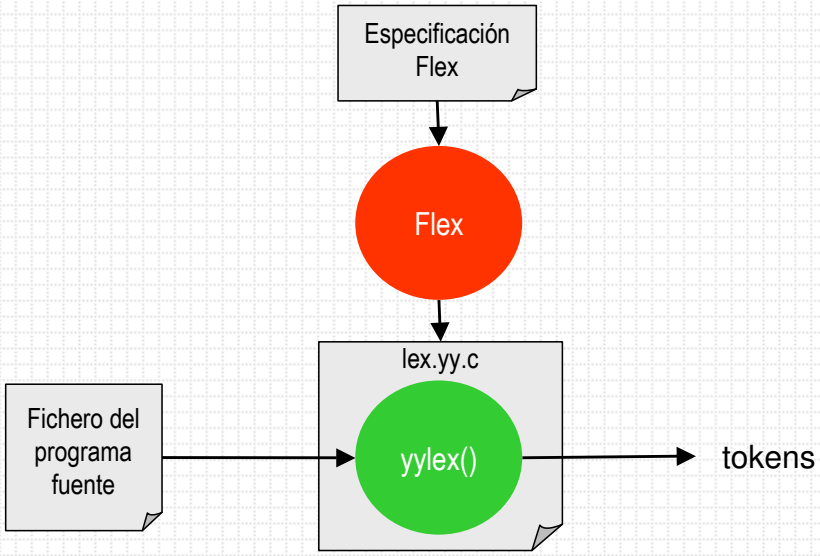
Construcción de un analizador léxico/morfológico

- Un analizador morfológico se puede desarrollar con distintos métodos:
  - con un autómata finito
  - con un programa a medida
  - utilizando una herramienta específica como Flex

- Independientemente del método seleccionado, para construir el analizador morfológico, hay que definir los requisitos que debe cumplir el analizador:
  - los tokens que tiene que reconocer:
    - identificadores
    - palabras reservadas
    - constantes (numéricas, lógicas,cadenas de caracteres, etc)
    - símbolos simples (+, -, *, /, etc)
    - símbolos múltiples (==, >=, etc)
  - el formato de los comentarios
  - los errores morfológicos que debe detectar

Funcionamiento básico de Flex

- Flex es una herramienta para construir analizadores léxicos.
- Flex recibe como entrada un conjunto de descripciones de tokens, y genera la función C **yylex()** que es un analizador léxico que reconoce dichos tokens.
- Los tokens se describen mediante **patrones** que son extensiones de las **expresiones regulares**.
- Al conjunto de las descripciones de tokens se le llama **especificación Flex**.



Descripción del primer ejemplo (I)

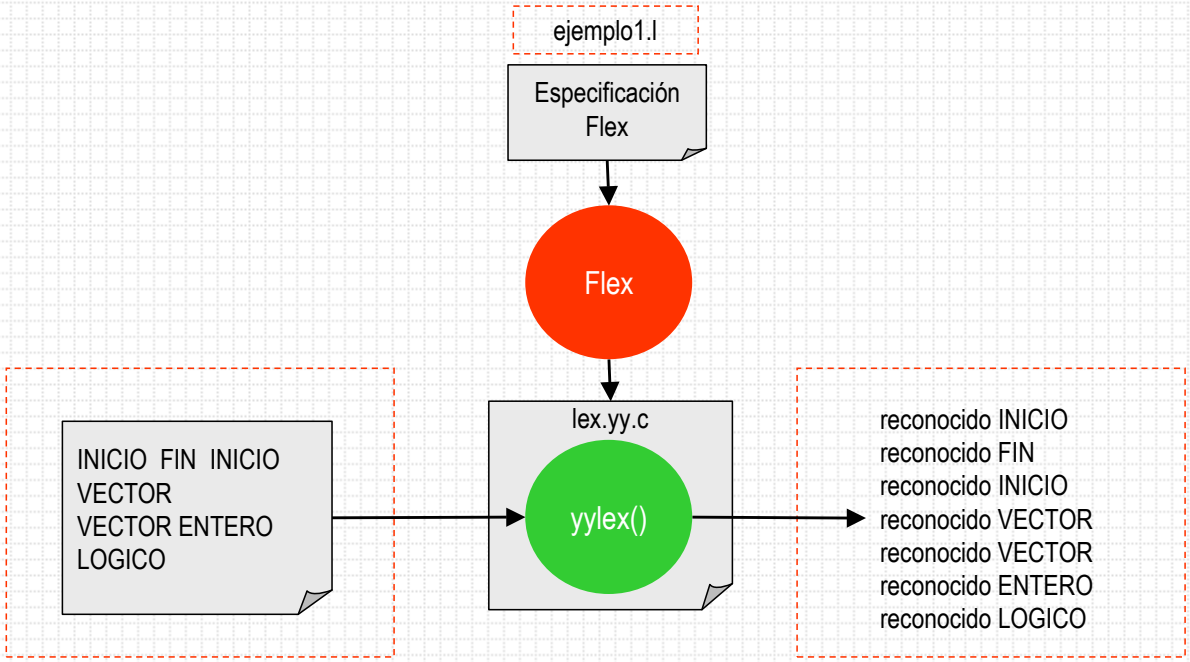
- Construir un analizador léxico, utilizando la herramienta Flex, que cumpla las siguientes especificaciones:
  - reconoce en el fichero de entrada las palabras "INICIO" , "FIN", "VECTOR", "ENTERO" y "LOGICO" .
  - Cada vez que el analizador reconoce una de las palabras anteriores, muestra en la salida estándar un mensaje de aviso de token reconocido. Los mensajes indican qué token se ha identificado con el siguiente texto:

    "reconocido INICIO"
    "reconocido FIN"
    "reconocido VECTOR"
    "reconocido ENTERO"
    "reconocido LOGICO"

  - El fichero de especificación Flex se llamará **ejemplo1.l**.

Descripción del primer ejemplo (II)

- Una visión gráfica del primer ejemplo sería:



El fichero de especificación Flex (I)

Estructura del fichero

- La estructura del fichero de especificación Flex se compone de tres secciones separadas por líneas que contienen el separador **%%**.

sección de definiciones

```
%{
     /* delimitadores de código C */
%}

%%
```

sección de reglas

```
%%
```

sección de funciones de usuario

El fichero de especificación Flex (II)

La sección de definiciones (I)

- La sección de definiciones contiene la siguiente información
  - Código C encerrado entre líneas con los caracteres **%{** y **%}** que se copia literalmente en el fichero de salida **lex.yy.c** antes de la definición de la función **yylex()**. Habitualmente esta sección contiene declaraciones de variables y funciones que se utilizan posteriormente en la sección de reglas así como directivas **#include**.
  - Definiciones propias de Flex, que permiten asignar un nombre a una expresión regular o a una parte, y posteriormente utilizar ese nombre en la sección de reglas. Estas definiciones se verán con más detalle cuando se estudien los patrones de Flex.
  - Opciones de Flex similares a las opciones de la línea de comandos.
  - Definición de condiciones de inicio. Estas definiciones se verán con más detalle cuando se estudien los patrones de Flex.
  - Cualquier línea que empiece con un espacio en blanco se copia literalmente en el fichero de salida **lex.yy.c**. Habitualmente se utiliza para incluir comentarios encerrados entre /* y */. (Esta funcionalidad depende de la versión de Flex)

El fichero de especificación Flex (III)

La sección de definiciones (II)

- En la figura se muestra una primera aproximación del fichero de especificación Flex del primer ejemplo (ejemplo1.l) Únicamente se ha completado la sección de definiciones. A medida que se construya el ejemplo se completarán las demás secciones del fichero.

ejemplo1.l

```
%{
#include <stdio.h>          /* para utilizar printf en la
                               sección de reglas */
%}

%option noyywrap

%%
```

- La directiva **#include <stdio.h>** se necesita porque en la sección de reglas se utiliza la función **printf** para mostrar en la salida estándar los mensajes de aviso de token reconocido.
- El significado de la opción **noyywrap** se explica en el siguiente apartado.

El fichero de especificación Flex (IV)

La sección de definiciones (III)

- Significado de la opción **noyywrap**:
  - La función **yylex()** puede analizar morfológicamente varios ficheros, encadenando uno detrás de otro con el mecanismo que se describe a continuación.
  - Cuando **yylex()** encuentra un fin de fichero, llama a la función **yywrap()**. Si la función devuelve 0, el análisis continúa con otro fichero, y si devuelve 1, el análisis termina.
  - ¿Quién proporciona la función **yywrap()** ? En Linux, la librería de Flex proporciona una versión por defecto de **yywrap()** que devuelve 1. Hay que enlazar con esa librería con la opción -lf.
  - La opción **noyywrap** evita se invoque a la función **yywrap()** cuando se encuentre un fin de fichero, y se asuma que no hay más ficheros que analizar.
    Esta solución es más cómoda que tener que escribir la función o bien enlazar con alguna librería.

El fichero de especificación Flex (V)

La sección de reglas (I)

- La sección de reglas contiene:
  - Los patrones que describen los tokens y código C.
    Cada patrón se sitúa en una línea, seguido de uno o más espacios en blanco y a continuación el código C que se ejecuta cuando se encuentra dicho patrón en la entrada que se está analizando. El código C se cierra entre llaves {}.
  - Si una línea empieza por un espacio en blanco se considera código `C` y se copia literalmente en el fichero de salida. También se asume que es código `C` todo lo que se escriba entre `%{` y `%}`, y se copia literalmente en el fichero de salida. (Esta funcionalidad depende de la versión de Flex)
- ¿Cómo se comporta la rutina de análisis **yylex()** ?
  - Busca en la entrada los patrones que se definen en la sección de reglas, es decir, los tokens. Cada vez que se encuentra un token, se ejecuta el código C asociado con el patrón. Si no se identifica ningún token, se ejecuta **la regla por defecto** : el siguiente carácter en la entrada se considera reconocido y se copia en la salida.
  - El código C asociado a cada patrón puede tener una sentencia **return**, que devuelve un valor al llamador de la función **yylex()**cuando se identifique el patrón en la entrada. La siguiente llamada a **yylex()** comienza a leer la entrada en el punto donde se quedó la última vez. Cuando se encuentra el fin de la entrada **yylex()** devuelve 0 y termina.
  - Si ninguna regla tiene una sentencia **return**, **yylex()** analiza la entrada hasta que encuentra el fin de la misma.

El fichero de especificación Flex (VI)

La sección de reglas (II)

- Sección de reglas del primer ejemplo:
  - Los patrones de Flex para definir los tokens se estudiarán más adelante, y ya se verá que el patrón para representar una palabra es la misma palabra.
  - La sección de reglas del primer ejemplo contiene:
    - Una regla para la palabra "INICIO", que se compone de su patrón (que es la misma palabra) y el código C que se va a ejecutar cuando se identifique en la entrada. El código muestra en la salida estándar el aviso "reconocido INICIO".
    - Reglas similares a la anterior para las palabras "FIN", "VECTOR", "ENTERO" y "LOGICO" .

ejemplo1.l

```
%{
#include <stdio.h>          /* para utilizar printf en la
                               sección de reglas */
%}
%option noyywrap

%%
INICIO   { printf("reconocido INICIO\n"); }
FIN      { printf("reconocido FIN\n"); }
VECTOR   { printf("reconocido VECTOR\n"); }
ENTERO   { printf("reconocido ENTERO\n"); }
LOGICO   { printf("reconocido LOGICO\n"); }
%%
```


El fichero de especificación Flex (VII)

La sección de funciones de usuario (I)

- La sección de funciones de usuario se copia literalmente en el fichero de salida.
- Esta sección habitualmente contiene las funciones escritas por el usuario para ser utilizadas en la sección de reglas, es decir, funciones de soporte.
- En esta sección también se incluyen las funciones de Flex que el usuario puede redefinir, por ejemplo, la función **yywrap()** se situaría en esta sección.
- La función **yylex()** generada por Flex, tiene que ser invocada desde algún punto, habitualmente desde el analizador sintáctico. Pero para realizar pruebas de la función **yylex()** se puede incorporar una función **main()** en la sección de funciones de usuario dentro del fichero de especificación Flex. Una versión muy simple sería:

```
int main()
{
        return yylex();
}
```

Esta llamada única a **yylex()** permite realizar el análisis léxico hasta encontrar el fin de la entrada siempre que en ningún fragmento de código C asociado a los patrones de los tokens aparezca una sentencia **return** que haga terminar a **yylex()**. En ese caso el **main()** sería distinto como se mostrará mas adelante.

El fichero de especificación Flex (VIII)

La sección de funciones de usuario (II)

- Se completa el fichero **ejemplo1.l** incluyendo una función **main()** en la sección de funciones de usuario.

ejemplo1.l

```
%{
#include <stdio.h>        /* para utilizar printf en la
                            sección de reglas */
%}
%option noyywrap

%%
INICIO   { printf("reconocido INICIO\n"); }
FIN      { printf("reconocido FIN\n"); }
VECTOR   { printf("reconocido VECTOR\n"); }
ENTERO   { printf("reconocido ENTERO\n"); }
LOGICO   { printf("reconocido LOGICO\n"); }
%%

int main()
{
        return yylex();
}
```


Solución del primer ejemplo (I)

Creación del ejecutable

- A partir del fichero de especificación Flex **ejemplo1.l** se genera el ejecutable de la siguiente manera:
  - Compilar la especificación Flex

    flex ejemplo1.l se crea el fichero lex.yy.c

  - Generar el ejecutable:

    gcc -Wall -o ejemplo1 lex.yy.c se crea el fichero ejemplo1

Solución del primer ejemplo (II)

Realización de pruebas

- Para probar el funcionamiento del analizador léxico implementado en el primer ejemplo, se arranca el ejecutable y se realizan las siguientes pruebas:

ENTRADA Y SALIDA	PROCESO
cadena cualquiera cadena cualquiera	No se identifica ningún token. Se copia la entrada en la salida.
INICIO reconocido INICIO	Se identifica el token "INICIO". Se muestra un mensaje de aviso.
FIN reconocido FIN	Se identifica el token "FIN". Se muestra un mensaje de aviso.
VECTORENTERO reconocido VECTOR reconocido ENTERO	Se identifican los tokens "VECTOR" y "ENTERO". Observar que ambos tokens se reconocen aunque no están separados. Se muestran los mensajes de aviso correspondientes.
^D	Fin de la entrada. El analizador morfológico termina su ejecución.

Texto azul: entrada del usuario
Texto rojo: respuesta del analizador

Segundo ejemplo (I)

Diseño del fichero de especificación (I)

- En el primer ejemplo, la función **main()** invoca a la función **yylex()** una sola vez, y como en ninguna de las reglas aparece una sentencia **return**, la función **yylex()** se ejecuta hasta encontrar el fin de la entrada.
- En el segundo ejemplo, se quiere modificar la especificación Flex para que la función **yylex()** devuelva un valor diferente para cada token identificado, y la función **main()** sea la responsable de mostrar por la salida estándar el aviso correspondiente del token identificado.
- Los cambios que hay que hacer son los siguientes:
  - Definir constantes diferentes para los tokens. En la sección de definiciones se añaden las siguientes sentencias:

```
#define TOK_INICIO 1
#define TOK_FIN 2
#define TOK_VECTOR 3
#define TOK_ENTERO 4
#define TOK_LOGICO 5
```

  - Modificar la sección de reglas para que el código C asociado a cada token en lugar de mostrar en la salida estándar un aviso, devuelva la constante definida para el token:

```
INICIO   { return TOK_INICIO; }
FIN      { return TOK_FIN; }
VECTOR   { return TOK_VECTOR; }
ENTERO   { return TOK_ENTERO; }
LOGICO   { return TOK_LOGICO; }
```

Segundo ejemplo (II)

Diseño del fichero de especificación (II)

- Modificar la función **main()** para que llame a la función **yylex()** y muestre un mensaje de aviso diferente para cada token en función del valor devuelto por **yylex()**. La función **main()** llama repetidas veces a la función **yylex()** hasta que se termina la entrada, es decir, hasta que la función **yylex()** devuelve 0.

```
int main()
{
   int token;
   while (1)
   {
      token = yylex();
      if (token == TOK_INICIO)
         printf("reconocido INICIO\n");
      if (token == TOK_FIN)
         printf("reconocido FIN\n");
      /* ifs para los tokens TOK_VECTOR, TOK_ENTERO y TOK_LOGICO)
      if (token == 0)
        break;
   }
   return 0;
}
```


Segundo ejemplo (III)

Creación del ejecutable y realización de pruebas

- A partir de las modificaciones descritas en el apartado anterior, crear un fichero de especificación Flex, de nombre **ejemplo2.l**
- Generar el ejecutable:
  - Compilar la especificación Flex

    flex ejemplo2.l se crea el fichero lex.yy.c

  - Generar el ejecutable:

    gcc -Wall -o ejemplo2 lex.yy.c se crea el fichero ejemplo2

- Si se realizan las mismas pruebas que se hicieron con el primer ejemplo, se obtendrán los mismos resultados ya que no se ha modificado la funcionalidad del analizador léxico sino su diseño.

Tercer ejemplo

- Se crea un nuevo fichero **ejemplo3.l** a partir de **ejemplo2.l**, eliminando de este último las definiciones del los tokens que se traspasan a un fichero de cabecera **tokens.h.** El nuevo fichero **ejemplo3.l**, en su sección de definiciones contiene la directiva que incluye dicho fichero de cabecera.

- Generar el ejecutable:
  - Compilar la especificación Flex

    flex ejemplo3.l se crea el fichero lex.yy.c

  - Generar el ejecutable:

    gcc -Wall -o ejemplo3 lex.yy.c se crea el fichero ejemplo3

- Si se realizan las mismas pruebas que se hicieron con los dos primeros ejemplos, se obtendrán los mismos resultados ya que no se ha modificado la funcionalidad del analizador léxico sino su diseño.

Los patrones de Flex (I)

Descripción

- Los patrones de Flex son:
  - el mecanismo para representar los tokens.
  - una extensión de las expresiones regulares.

- Los patrones están formados por:
  - caracteres "normales" que se representan a sí mismos
  - metacaracteres que tienen un significado especial

- Para utilizar un metacarácter como carácter "normal" hay que ponerlo entre comillas. Por ejemplo, como el asterisco es un metacarácter, si se quiere reconocer el token asterisco, hay que definirlo como "*".

- En los siguientes apartados se describen algunos de los metacaracteres de Flex, en concreto, aquellos que son necesarios para especificar los tokens del lenguaje ALFA.

- Un mismo token se puede expresar con distintos patrones.

Los patrones de Flex (II)

Metacaracteres (I)

.	Representa cualquier caracter exceptuando el salto de línea "\n".
[ ]	Representa cualquiera de los caracteres que aparecen dentro de los corchetes. Para indicar un rango de caracteres se utiliza el símbolo menos "-".

- `[xyz]` representa una "x", una "y" o una "z"
- `[abj-oZ]` representa una "a", una "b" cualquier letra de la "j" a la "o", o una "Z"
- `[ \t]` representa el espacio y el tabulador
- `[0-9]` representa los dígitos del 0 al 9
- `[a-z]` representa las letras minúsculas
- `[A-Z]` representa las letras mayúsculas
- `[a-zA-Z]` representa las letras minúsculas y las mayúsculas
- `[0-9a-zA-Z]` representa los dígitos del 0 al 9, las letras minúsculas y las mayúsculas

Los patrones de Flex (III)

Metacaracteres (II)

*	Indica 0 ó más ocurrencias de la expresión que le precede.

- `ab*` representa todas las palabras que empiezan por una "a" seguida de 0 o más "b", por ejemplo "a", "ab", "abb", "abbb".
- `[a-zA-Z][a-zA-Z0-9]*` representa todas las palabras que empiezan por una letra minúscula o mayúscula seguida de 0 o más letras o dígitos, como por ejemplo "v1", "indice", "maximo", "D".

+	Indica 1 ó más ocurrencias de la expresión que le precede.

- `x+` representa todas las palabras formadas por "x", por ejemplo "x", "xx", "xxx".

`[0-9]+` representa los números de uno o más dígitos, por ejemplo "12", "465", "000", "097".

Los patrones de Flex (IV)

Metacaracteres (III)

| | Identifica la expresión que le precede o la que le sigue.

- `A|B` representa la letra "A" o la letra "B". Este patrón se comporta de la misma manera que el patrón `[AB]`.

"..." Representa lo que esté entre las comillas literalmente. Los metacaracteres pierden su significado excepto "\".
El metacaracter \ si va seguido de una letra minúscula se asume que es una secuencia de escape de `C`, como por ejemplo el tabulador \t.

- `"/*"` representa a la agrupación de caracteres "/*".

( ) Agrupan expresiones.

- (ab|cd)+r representa "abr", "ababr", "cdr", "cdcdr", “abcdr”, etc.

Los patrones de Flex (V)

Metacaracteres (IV)

{ nombre } Un nombre encerrado entre llaves significa al expansión de la definición de "nombre".
En la sección de definiciones se pueden definir expresiones regulares y asignarles un nombre. Por ejemplo:

```
DIGITO  [0-9]
LETRA   [a-zA-Z]
```

Posteriormente, cualquier aparición de `{DIGITO}` se sustituye por la expresión regular `[0-9]`, y `{LETRA}` por `[a-zA-Z]`.

La expresión regular de todas las palabras que empiezan por una letra minúscula o mayúscula seguida de 0 o más letras o dígitos:

```
[a-zA-Z]([0-9]|[a-zA-Z])*
```

es idéntica a:

```
{LETRA} ({DIGITO}|{LETRA})*
```

Los patrones de Flex (VI)

Cómo se identifican los patrones en la entrada (I)

- El analizador busca en la entrada cadenas de caracteres que concuerden con alguno de los patrones definidos en la sección de reglas.

- Si se encuentra concordancia con más de un patrón, se selecciona el patrón más largo.
  Por ejemplo, si se definen las reglas:

```
INICIO        { return TOK_INICIO; }
FIN           { return TOK_FIN; }
[A-Z]+        { return TOK_ID;}
```

  La entrada INICIOFIN se identificará como TOK_ID porque el análisis de la entrada se realiza de la siguiente manera:
  - Desde la primera "I" de la cadena de entrada hasta la primera "O", concuerdan dos patrones, el de TOK_INICIO y el de TOK_ID.
  - Cando se lee la letra "F", en ese momento se descarta el patrón correspondiente a TOK_INICIO, y se selecciona el patrón de TOK_ID que es más largo.
    (Se recomienda construir un analizador léxico de prueba)

Los patrones de Flex (VII)

Cómo se identifican los patrones en la entrada (II)

- Si hay concordancia con varios patrones, y además de la misma longitud, se elige el patrón que esté situado primero en la sección de reglas dentro del fichero de especificación Flex. Por lo tanto, **es determinante el orden en que se colocan las reglas.**
  Por ejemplo, si se definen las reglas:

```
[A-Z]+        { return TOK_ID;}
INICIO        { return TOK_INICIO; }
FIN           { return TOK_FIN; }
```

  La entrada INICIO se identificará como TOK_ID porque hay concordancia con dos patrones, el de TOK_INICIO y el de TOK_ID, pero el patrón de TOK_ID aparece antes en el fichero de especificación.

  Al procesar con Flex estas reglas, se muestra un mensaje en el que se indica que las reglas segunda y tercera nunca se van a identificar.

  (Se recomienda construir un analizador léxico de prueba)

Los patrones de Flex (VIII)

Cómo se identifican los patrones en la entrada (III)

- En el ejemplo anterior, para que se identifiquen las palabras reservadas INICIO y FIN correctamente, y no como identificadores, se deben colocar sus correspondientes reglas antes de la regla de los identificadores, de la siguiente manera:

```
INICIO          { return TOK_INICIO; }
FIN             { return TOK_FIN; }
[A-Z]+          { return TOK_ID;}
```

- Cuando se ha determinado el patrón que concuerda con la entrada, además de ejecutarse el código C asociado al patrón, se producen las siguientes acciones:
  - el texto del token se almacena en la variable de Flex **yytext** que es un **char***
  - la longitud se almacena en la variable entera **yyleng**

Cuarto ejemplo (I)

Enunciado

- Diseñar un fichero de especificación Flex, **ejemplo4.l**, usando como guía el fichero de especificación del tercer ejemplo, para construir un analizador léxico que se ajuste a los requisitos que se describen a continuación.

- Los tokens que debe reconocer el analizador léxico son:
  - TOK_ID: Palabra formada por letras mayúsculas o minúsculas y dígitos del 0 al 9, y cuyo carácter inicial siempre es una letra.
  - TOK_NUM: palabra formada por dígitos del 0 al 9.
  - La definición de los tokens se realiza en el fichero **tokens.h**

- Las acciones que debe realizar el analizador léxico cuando reconozca algún token son las siguientes:
  - Cuando se reconozca un identificador se mostrará el mensaje "TOK_ID=<lexema>"
  - Cuando se reconozca un número se mostrará el mensaje "TOK_NUM=<lexema>"

  El acceso al lexema del último token reconocido se realiza a través de la variable **yytext**.

- Compilar la especificación Flex y generar el ejecutable.

Cuarto ejemplo (II)

Realización de pruebas

- Para probar el funcionamiento del analizador léxico implementado, se arranca el ejecutable y se realizan las siguientes pruebas:

ENTRADA Y SALIDA	PROCESO
variable1 TOK_ID=variable1	Se identifica un token de tipo TOK_ID. Se genera la salida especificada en el enunciado.
1234 TOK_NUM=1234	Se identifica un token de tipo TOK_NUM. Se genera la salida especificada en el enunciado.
44valor TOK_NUM=44 TOK_ID=valor	Se identifican los tokens de tipo TOK_NUM y TOK_ID. Observar que ambos tokens se reconocen aunque no están separados. Se genera la salida especificada en el enunciado.
ÑÑ_$$ ÑÑ_$$	No se identifica ningún token. Se muestra la entrada (acción por defecto)
^D	Fin de la entrada. El analizador morfológico termina su ejecución.

Texto azul: entrada del usuario
Texto rojo: respuesta del analizador

Los tokens del lenguaje ALFA (I)

Identificación

- Los tokens se identifican a partir de la gramática.
- Todo lo que aparece literalmente en las partes derechas de las reglas de la gramática:
  - **palabras reservadas** TRABAJO DEL ALUMNO
  - **símbolos simples** TRABAJO DEL ALUMNO
  - **símbolos compuestos** TRABAJO DEL ALUMNO
- **Identificadores**: cadenas de caracteres alfanuméricos, comenzando siempre con una letra (entendiendo carácter alfanumérico cualquier letra minúscula o mayúscula, o cualquier dígito entre el 0 y el 9). TRABAJO DEL ALUMNO
- **Constantes enteras**: cadenas de dígitos entre el 0 y el 9. TRABAJO DEL ALUMNO

Fichero de especificación Flex para el lenguaje ALFA (I)

Construcción

- Partiendo del cuarto ejemplo visto anteriormente se construirá la especificación Flex para el lenguaje ALFA realizando las siguientes tareas:
  - modificar el fichero **tokens.h** introduciendo una directiva **#define** por cada uno de los tokens del lenguaje ALFA. En la página web del laboratorio se puede encontrar una versión del fichero tokens.h.
  - Hacer una copia del fichero **ejemplo4.l** en un nuevo fichero **alfa.l**
  - En la sección de definiciones del fichero **alfa.l** incluir las definiciones de las expresiones regulares que se van a utilizar para los patrones de los tokens de tipo identificador y constante entera.
  - En la sección de reglas del fichero **alfa.l** incluir una regla para cada token, en cuya acción se devuelva el token. Tener en cuenta el orden de las reglas (ver siguiente transparencia).
  - En la sección de funciones de usuario del fichero **alfa.l** modificar la función **main()** para que muestre un mensaje por pantalla cada vez que reciba un token de la función **yylex()**. El mensaje contiene dos campos, el primero es una cadena que decribe el tipo de token según la siguiente lista:
    - TOKEN_ID para los identificadores
    - TOKEN_KEY para todas las palabras reservadas
    - TOKEN_NUM para los números

    El segundo campo del mensaje es el número del token correspondiente según las definiciones contenidas en el fichero **tokens.h**

Fichero de especificación Flex para el lenguaje ALFA (II)

El orden de las reglas

- El orden de las reglas es importante cuando se identifica en la entrada una cadena de caracteres que concuerda con dos patrones de la misma longitud ya que se selecciona la regla que aparece primero en el fichero de especificación Flex.
- Por ejemplo, las palabras reservadas (INICIO, VECTOR, SI, ...) concuerdan a la vez con dos patrones:
  - el patrón que define la propia palabra reservada
  - el patrón que define a los identificadores

  Como ambas concordancias son de la misma longitud, si la primera regla que se sitúa en la sección de reglas fuera la correspondiente a los identificadores, nunca se detectarían en la entrada las palabras reservadas porque siempre serían consideradas como identificadores. **La solución es situar primero las reglas de las palabras reservadas.**
- Cuando ocurre una situación como la descrita en el párrafo anterior, al compilar la especificación, se recibe el aviso de que hay patrones que nunca se van a reconocer.
- Realizar la siguiente prueba: en el fichero **alfa.l** situar el patrón de los identificadores delante de los patrones de las palabras reservadas. ¿Qué ocurre cuando la entrada es, por ejemplo, la palabra "MIENTRAS" ?

Gestión de los espacios en blanco (I)

- Otras de las funciones del analizador morfológico, además de identificar los tokens en el fichero de entrada, es ignorar los espacios en blanco, los tabuladores y los saltos de línea que aparecen en la entrada.
- Una solución para incorporar esta funcionalidad es:
  - diseñar el patrón que identifique los caracteres que se quieran ignorar.
  - Incorporar una regla para dicho patrón y que tenga como acción "no hacer nada", que en Flex se especifica con un punto y coma en el lugar del código C.

Control de la posición de los tokens en el fichero de entrada (I)

- **El analizador léxico** es la parte del compilador que accede al fichero de entrada y por lo tanto **es el que conoce la posición (línea y carácter) de los tokens en el fichero de entrada**. Esta posición es muy importante para informar de los errores de compilación ya que los errores deben de ubicarse en una posición del fichero de entrada.
- Para conocer la posición de los tokens en el fichero de entrada se pueden utilizar dos variables, una para almacenar la línea y otra para el carácter (columna) dentro de la línea. Ambas variables se pueden declarar en la sección de definiciones del fichero de especificación Flex, por ejemplo:

```
%{
        int numero_linea = 1;    /* número de línea */
        int numero_caracter = 0;    /* número de carácter */
%}
```

Control de la posición de los tokens en el fichero de entrada (II)

- La actualización de las variables que almacenan la posición de los tokens en el fichero de entrada se realiza en el código de las reglas. Por ejemplo:
  - cuando se identifique la palabra clave INICIO se avanza en la posición del fichero 6 caracteres (también se puede usar la variable **yyleng** para conocer cuántos caracteres se avanzan en la entrada).
  - Cuando se encuentra un identificador o un número entero, se puede utilizar la variable de Flex **yyleng** para incrementar el número de carácter.
  - cuando se identifica un tabulador o un espacio en blanco, se avanza en la posición del fichero 1 carácter.
  - cuando se encuentra un salto de línea se incrementa en 1 el número de línea, y se reinicia a 0 el número de carácter dentro de la línea.
- Puede observarse que la acción asociada a los caracteres tabulador y espacio en blanco, es diferente a la asociada al carácter de salto de línea. Por este motivo, aunque los tres tipos de carácter deben ser igualmente ignorados, se utilizan reglas diferentes para poder incorporar acciones distintas.

Gestión de los comentarios (I)

- Para gestionar los comentarios es necesario:
  - Diseñar el patrón que identifique el comentario.
  - Incorporar una regla para dicho patrón y que tenga como acción incrementar en 1 el número de línea y reiniciar a 0 el número de carácter dentro de la línea.

Gestión de errores (I)

- Si no se reconoce ninguno de los patrones diseñados, la opción por defecto del analizador generado automáticamente por Flex es copiar la entrada en la salida. Para evitar este comportamiento, se puede definir un tipo de "token de error", de manera que cuando no se identifique ninguno de los tokens del lenguaje, se considere que se ha encontrado un error, y tenga como acción asociada devolver ese "token de errror".

- Para incorporar esta funcionalidad:
  - definir en el fichero **tokens.h** el nuevo token de error con valor -1.
  - diseñar el patrón que identifique el "token de error"
  - Incorporar una regla para dicho patrón y que tenga como acción devolver el token definido en el fichero **tokens.h**

- En la función **main()** se incorpora una nueva comprobación para el error.

Ficheros de entrada/salida de Flex

- **FILE* yyin**
  - Es el fichero de entrada, del que lee la función **yylex()**
  - Por defecto es la entrada estándar.
  - El usuario puede asignarle cualquier variable de tipo FILE*, por supuesto, antes de que se inicie el análisis, es decir, antes de invocar a la función yylex().

- **FILE* yyout**
  - Es el fichero de salida, en el que se escribe la regla por defecto (cuando no concuerda ningún patrón se copia la entrada en la salida)
  - Por defecto es la salida estándar.
  - El usuario puede asignarle cualquier variable de tipo FILE*.