



**PROCESADORES DE LENGUAJE**

**Ingeniería Informática**  
**Especialidad de computación**  
**Tercer curso, segundo cuatrimestre**



Departamento de Informática y Análisis Numérico  
Escuela Politécnica Superior de Córdoba  
Universidad de Córdoba

---

# **GUION DE LAS CLASES DE PRÁCTICAS**

**FLEX/LEX**

## Contenido

Fichero makefile .....	3
Primer ejemplo mínimo .....	4
Segundo ejemplo mínimo .....	5
Conversión de letras mayúsculas a minúsculas.....	6
Verbos y no verbos .....	7
Verbos, adjetivos, adverbios, pronombres, ... ..	9
Reconocimiento y almacenamiento de palabras .....	12
Reconocimiento y almacenamiento de palabras, segunda versión .....	17
Componentes léxicos de un lenguaje de programación .....	22
(1ª versión) .....	22
Componentes léxicos de un lenguaje de programación .....	26
(2ª versión) .....	26
Comando REJECT .....	29
Función yymore().....	31
Reconocimiento de comentarios.....	33
AFD-Comentarios1.l.....	35
AFD-Comentarios2.l.....	37
Reconocimiento de cadenas, primera versión .....	40
Reconocimiento de cadenas, segunda versión .....	42
AFD-Cadenas_1.....	44
AFD-Cadenas_2.....	46

## Fichero makefile

```
# Si se desea generar ejemplo0.exe entonces teclear
# make
#
# Si se desea generar otro "ejemplo" entonces teclear
#         make NAME=ejemplo

NAME = ejemplo0

# Directives for the compiler
# -c: the output is an object file, the linking stage is not
done.
# -g: debug
# -Wall: all warnings
# -ansi: standard language
# -O2: optimization level
# -Wno-unused-function -Wno-sign-compare -Wno-implicit-
function-declaration: disabled error messages
CFLAGS = -c -g -Wall -ansi -O2 -Wno-unused-function -Wno-sign-
compare -Wno-implicit-function-declaration

# Directive for flex's library
LFLAGS = -lfl

# Predefined macros
#
# $@: name of the target
# $^: all the dependencies
# $<: first dependency
#
#####

##
$(NAME).exe: $(NAME).o
    @echo "Generating" $@
    @gcc $< $(LFLAGS) -o $@

$(NAME).o: lex.yy.c
    @echo "Compiling" $< "and generating" $@
    @gcc $(CFLAGS) $< -o $@
    @echo "--> Deleting the auxiliary file: lex.yy.c"
    @rm -f lex.yy.c

lex.yy.c: $(NAME).l
    @echo "Generating" $@
    @flex $<

clean:
    @echo "--> Deleting object files"
    @rm -f lex.yy.c *.o
```

## Primer ejemplo mínimo

- **Descripción**
  - Ejemplo mínimo de flex.
  - El analizador léxico reproduce literalmente todo lo que se teclea.
- **Compilación**
  - Se crea el fichero minimo.exe  
*make NAME=minimo* (Nota: sin la extensión .l)
- **Ejecución**

```
$ ./minimo.exe  
Prueba  
Prueba  
123  
123  
CONTROL D  
$
```
- **Fichero**
  - minimo.l

```
/****** Programa mínimo de lex *****/  
  
%%
```

## Segundo ejemplo mínimo

- **Descripción**
  - Ejemplo mínimo de flex pero con comentarios
  - El analizador léxico reproduce literalmente todo lo que se teclea
- **Compilación**
  - Se crea el fichero ejemplo0.exe  
*make NAME= ejemplo0*
- **Ejecución**
  - \$ ./ejemplo0.exe*
  - Prueba*
  - Prueba*
  - 123*
  - 123*
  - CONTROL D*
  - \$*
- **Fichero**
  - ejemplo0.l

## Conversión de letras mayúsculas a minúsculas

- **Descripción**
  - El analizador léxico convierte las letras mayúsculas en minúsculas, elimina blancos al final de la línea y sustituye una serie de blancos por uno solo.
- **Compilación**
  - Se crea el fichero ejemplo1.exe  
*make NAME= ejemplo1*
- **Ejecución**
  - \$.ejemplo1.exe*  
*Conversión de letras MAYÚSCULAS y espacios finales*  
*conversión de letras mayúsculas y espacios finales*
- **Fichero**
  - ejemplo1.l

```
/** ejemplo1.l */

/** Zona de las declaraciones o definiciones */
%{
    /* Descripción
       El analizador léxico
       - convierte la letras mayúsculas en minúsculas,
       - elimina blancos al final de la línea
       - y sustituye una serie de blancos por uno solo.
    */
}%

/***** Zona de las reglas *****/
%%
[A-Z]      putchar(yytext[0]+'a'-'A');
"Á"        printf("á");
"É"        printf("é");
"Í"        printf("í");
"Ó"        printf("ó");
"Ú"        printf("ú");
[ ]+$      ;
[ ]+       putchar(' ');
%%

/** Zona de funciones auxiliares */

/** No hay funciones auxiliares */
```

## Verbos y no verbos

- Fichero

- o ejemplo2.l

```
/** ejemplo2.l */

/** Zona de las declaraciones o definiciones */
%{
    /*
        Descripción: el analizador léxico distingue entre verbos
        y no verbos
    */
}%

/***** Zona de las reglas *****/
%%

[\\t ]+      ;      /* Se ignoran los espacios en blanco */

es |
soy |
eres |
era |
ser |
siendo |
sido |
hacer |
hace |
hizo |
puede |
tiene |
tener |
ir           { printf("%s: es un verbo\\n", yytext); }

[a-zA-Z]+    { printf("%s: no es un verbo\\n", yytext); }

.|\\n        { ECHO; /* resto de cadenas */ }
%%

/** Zona de funciones auxiliares */

int main()
{
    yylex();

    return 0;
}
```

- **Descripción**
  - El analizador léxico distingue entre verbos y no verbos
  - Los verbos que reconoce están predefinidos.
- **Compilación**
  - Se crea el fichero ejemplo2.exe  
*make NAME= ejemplo2*
- **Ejecución**
  - \$/ejemplo2.exe*
  - Esta persona es escritor.*
  - Esta: no es un verbo*
  - persona: no es un verbo*
  - es: es un verbo*
  - escritor: no es un verbo*
  - Él puede ser astronauta.*
  - Él: no es un verbo*
  - puede: es un verbo*
  - ser: es un verbo*
  - astronauta: no es un verbo*



## Verbos, adjetivos, adverbios, pronombres, ...

- Fichero

- ejemplo3.l

```
/** ejemplo3.1 */

/** Zona de las declaraciones o definiciones */
%{
    /* Descripción
       El analizador léxico distingue entre verbos, adjetivos,
       adverbios, artículos,...
    */
}%

%%
[\\t ]+      /* ignora los espacios en blanco */ ;

es |
soy |
eres |
era |
ser |
siendo |
sido |
hacer |
hace |
hizo |
puedo |
puede |
tiene |
tener |
tengo |

ir          { printf("%s: es un verbo\\n", yytext); }

muy |
mucho |
bastante |
lentamente |
velozmente { printf("%s: es un adverbio\\n", yytext); }

a |
para |
desde |
de |
debajo |
encima |
detras |
entre      { printf("%s: es una preposicion\\n", yytext); }
si |
```

```

entonces |
y |
pero |
o          { printf("%s: es una conjuncion\n", yytext); }

alto |
bella |
inteligente |
amable |
feliz |
alegre      { printf("%s: es un adjetivo\n", yytext); }

yo |
tú |
él |
ella |
nosotros |
vosotros |
ellos      { printf("%s: es un pronombre\n", yytext); }

[a-zA-Z]+ {
    printf("%s: No reconocido; puede ser un nombre \n",
           yytext);
}

.|\\n      {ECHO; putchar('\\n'); /* resto de caracteres */ }

%%

/****Zona de funciones auxiliares ****/

int main()
{
    yylex();

    return 0;
}

```

- **Descripción**
  - El analizador léxico distingue entre verbos, adjetivos, adverbios, artículos,... que están predefinidos.
- **Compilación**
  - Se crea el fichero ejemplo3.exe  
*make NAME= ejemplo3*

- **Ejecución**

*\$ ./ejemplo3.exe*

*él puede ser muy inteligente*

*él: es un pronombre*

*puede: es un verbo*

*ser: es un verbo*

*muy: es un adverbio*

*inteligente: es un adjetivo*

*ella es muy inteligente*

*ella: es un pronombre*

*es: es un verbo*

*muy: es un adverbio*

*inteligente: es un adjetivo*

- **Observación**

- El análisis léxico no tiene en cuenta la reglas gramaticales, como muestra el siguiente ejemplo

*muy es inteligente ella*

*muy: es un adverbio*

*es: es un verbo*

*inteligente: es un adjetivo*

*ella: es un pronombre*

## Reconocimiento y almacenamiento de palabras

- **Descripción**

- El analizador léxico reconoce las palabras y las almacena en una tabla.
- Permite definir el tipo de cada palabra:
  - verbo, adv (adverbio), adj (adjetivo) y pron (pronombre)
- Muestra el uso del operador <sup>^</sup>
- Por ejemplo, si se teclea "al principio de la línea"  
*verbo amar luchar*  
define las palabras "amar" y "luchar" como verbos

- **Compilación**

- Se crea el fichero ejemplo4.exe  
*make NAME= ejemplo4*

- **Ejecución**

*\$ ./ejemplo4.exe*

*amar luchar*

*amar: no reconocida*

*luchar: no reconocida*

*verbo amar luchar*

*amar luchar*

*amar: verbo*

*luchar: verbo*

*verbo jugar* (Nota: la palabra verbo no está al principio de línea)

*verbo: no reconocida*

*jugar: no reconocida*

*jugar*

*jugar: no reconocida*

*verbo jugar*

*jugar*

*jugar: verbo*

*ella es muy inteligente*

*ella: no reconocida*

*es: no reconocida*

*muy: no reconocida*

*inteligente: no reconocida*

*adj inteligente*

*verbo es*

*pron ella*

*adv muy*  
*ella es muy inteligente*  
*ella: pronombre*  
*es: verbo*  
*muy: adverbio*  
*inteligente: adjetivo*

*pron inteligente*

*---> : la palabra inteligente ya está definida*

- **Fichero**

- ejemplo4.l

```
/** ejemplo4.l */

/** Zona de las declaraciones o definiciones */
%{
    /* Descripción
       El analizador léxico reconoce las palabras y las
       almacena en una tabla.

       Permite definir el tipo de cada palabra.
       Por ejemplo, si se teclea al principio de la línea
       verbo amar luchar
       define las palabras "amar" y "luchar" como verbos
    */

enum {
    BUSCAR = 0, /* Estado por defecto. */
    VERBO,
    ADJETIVO,
    ADVERBIO,
    NOMBRE,
    PREPOSICION,
    PRONOMBRE,
    CONJUNCION
};

int estado;

int poner_palabra(int tipo, char *palabra);
int buscar_palabra(char *palabra);
%}
```

```

%%
\n    { estado = BUSCAR; } /* Fin de línea:
                               vuelve al estado por defecto */

^verbo    { estado = VERBO; }
^adj      { estado = ADJETIVO; }
^adv      { estado = ADVERBIO; }
^nombre   { estado = NOMBRE; }
^prep     { estado = PREPOSICION; }
^pron     { estado = PRONOMBRE; }
^conj     { estado = CONJUNCION; }

[a-zA-Z]+ { /* palabra normal: la define o la busca */
            if(estado != BUSCAR)
            {
                /* define la palabra actual */
                poner_palabra(estado, yytext);
            }
            else
            {
                switch(buscar_palabra(yytext))
                {
                    case VERBO:
                        printf("%s: verbo\n", yytext);
                        break;
                    case ADJETIVO:
                        printf("%s: adjetivo\n", yytext);
                        break;
                    case ADVERBIO:
                        printf("%s: adverbio\n", yytext);
                        break;
                    case NOMBRE:
                        printf("%s: nombre\n", yytext);
                        break;
                    case PREPOSICION:
                        printf("%s: preposición\n", yytext);
                        break;
                    case PRONOMBRE:
                        printf("%s: pronombre\n", yytext);
                        break;
                    case CONJUNCION:
                        printf("%s: conjunción\n", yytext);
                        break;
                    default:
                        printf("%s: no reconocida \n",
                               yytext);
                        break;
                }
            }
        }
; /* Se ignora el resto de cadenas */
%%

```

```

/* Se define el tipo de lista enlazada de palabras y tipos */
struct Ficha_palabra {
    char *palabra_nombre;
    int palabra_tipo;
    struct Ficha_palabra *siguiente;
};

/* Se declara lista_palabra como una variable global
   que contendrá las palabras que se declaren */
static struct Ficha_palabra *lista_palabra;

extern void *malloc();

int main()
{
    yylex();

    return 0;
}

/*****

int poner_palabra(int tipo, char *palabra)
{
    struct Ficha_palabra *p;

    if(buscar_palabra(palabra) != BUSCAR)
    {
        printf("---> : la palabra %s ya está definida\n",
palabra);
        return 0;
    }

    /* Se introduce la palabra en la lista de palabras*/

    p = (struct Ficha_palabra *) malloc(sizeof(struct
Ficha_palabra ));

    p->siguiente = lista_palabra;
    p->palabra_nombre = (char *) malloc(strlen(palabra)+1);
    strcpy(p->palabra_nombre, palabra);
    p->palabra_tipo = tipo;
    lista_palabra = p;
    return 1; /* palabra definida */
}

```

```

int buscar_palabra(char *palabra)
{
    struct Ficha_palabra  *p = lista_palabra;

    /* se busca la palabra recorriendo la lista simple */
    for(; p; p = p->siguiente)
    {
        if(strcmp(p->palabra_nombre, palabra) == 0)
            return p->palabra_tipo;
    }

    return BUSCAR; /* no encontrada */
}

```



## Reconocimiento y almacenamiento de palabras, segunda versión

- **Descripción**
  - Realiza las mismas acciones que el ejemplo 4, pero utiliza el fichero auxiliar **p44.c** de código en lenguaje C para separar el análisis léxico de la gestión de la lista de palabras.
- **Compilación**
  - Se crea el fichero ejemplo44.exe  
*make NAME= ejemplo44*
- **Ficheros en el subdirectorio ejemplo44**
  - ejemplo44.l
  - p44.c
  - makefile

```
/* ***** ejemplo44.1 ***** */

/* ***** Zona de las declaraciones o definiciones ***** */
%{
    /* Descripción:
       El analizador léxico reconoce las palabras
       y las almacena en una tabla.

       Se utiliza un fichero auxiliar "p44.c"
       para codificar las funciones de la tabla

       Permite definir el tipo de cada palabra.
       Por ejemplo, si se teclea al principio de la línea
       verbo amar luchar
       define las palabras "amar" y "luchar" como verbos
    */

    enum {
        BUSCAR = 0, /* Estado por defecto. */
        VERBO,
        ADJETIVO,
        ADVERBIO,
        NOMBRE,
        PREPOSICION,
        PRONOMBRE,
        CONJUNCION
    };

    int estado;
    int poner_palabra(int tipo, char *palabra);
    int buscar_palabra(char *palabra);
}%
```

```

%%
\n    { estado = BUSCAR; } /* Fin de línea:
                               vuelve al estado por defecto */

^verbo    { estado = VERBO; }
^adj      { estado = ADJETIVO; }
^adv      { estado = ADVERBIO; }
^nombre   { estado = NOMBRE; }
^prep     { estado = PREPOSICION; }
^pron     { estado = PRONOMBRE; }
^conj     { estado = CONJUNCION; }

[a-zA-Z]+ { /* palabra normal: la define o la busca */
    if(estado != BUSCAR)
    {
        /* define la palabra actual */
        poner_palabra(estado, yytext);
    }
    else
    {
        switch(buscar_palabra(yytext))
        {
            case VERBO:
                printf("%s: verbo\n", yytext);
                break;
            case ADJETIVO:
                printf("%s: adjetivo\n", yytext);
                break;
            case ADVERBIO:
                printf("%s: adverbio\n", yytext);
                break;
            case NOMBRE:
                printf("%s: nombre\n", yytext);
                break;
            case PREPOSICION:
                printf("%s: preposición\n", yytext);
                break;
            case PRONOMBRE:
                printf("%s: pronombre\n", yytext);
                break;
            case CONJUNCION:
                printf("%s: conjunción\n", yytext);
                break;
            default:
                printf("%s: no reconocida \n",
                    yytext);
                break;
        }
    }
}

; /* Se ignora el resto de cadenas */
%%

```

o p44.c

```
#include <stdio.h>
#include <string.h>

enum {
    BUSCAR = 0, /* Estado por defecto. */
    VERBO,
    ADJETIVO,
    ADVERBIO,
    NOMBRE,
    PREPOSICION,
    PRONOMBRE,
    CONJUNCION
};

/* Se define el tipo de lista enlazada de palabras y tipos */
struct Ficha_palabra
{
    char *palabra_nombre;
    int palabra_tipo;
    struct Ficha_palabra *siguiente;
};

/* Se declara lista_palabra como una variable global que
contendrá las palabras que se declaren */
static struct Ficha_palabra *lista_palabra;

extern void *malloc();
int yylex();

int poner_palabra(int tipo, char *palabra);
int buscar_palabra(char *palabra);

int main()
{
    yylex();

    return 0;
}

/*****
int poner_palabra(int tipo, char *palabra)
{
    struct Ficha_palabra *p;

    if(buscar_palabra(palabra) != BUSCAR) {
        printf("---> : la palabra %s ya está definida\n",
            palabra);
        return 0;
    }
}
```

```

/* Se introduce la palabra en la lista de palabras*/

p = (struct Ficha_palabra *)
    malloc(sizeof(struct Ficha_palabra));

p->siguiente = lista_palabra;
p->palabra_nombre = (char *) malloc(strlen(palabra)+1);
strcpy(p->palabra_nombre, palabra);
p->palabra_tipo = tipo;
lista_palabra = p;
return 1; /* palabra definida */
}

int buscar_palabra(char *palabra)
{
    struct Ficha_palabra *p = lista_palabra;

    /* se busca la palabra recorriendo la lista simple */
    for(; p; p = p->siguiente) {
        if(strcmp(p->palabra_nombre, palabra) == 0)
            return p->palabra_tipo;
    }

    return BUSCAR; /* no encontrada */
}

```

- **Makefile**

```
# Si se desea generar ejemplo44.exe entonces teclear
# make
#
# Si se desea generar otro "ejemplo" entonces teclear
#         make NAME=ejemplo
NAME = ejemplo44

# Directives for the compiler
# -c: the output is an object file, the linking stage is not
done.
# -g: debug
# -Wall: all warnings
# -ansi: standard language
# -O2: optimization level
# -Wno-unused-function -Wno-sign-compare -Wno-implicit-
function-declaration: disabled error messages
CFLAGS = -c -g -Wall -ansi -O2 -Wno-unused-function -Wno-sign-
compare -Wno-implicit-function-declaration
# Directive for flex's library
LFLAGS = -lfl

# Predefined macros
# $@: name of the target
# $^: all the dependencies
# $<: first dependency
#
#####

# MODIFICADO
$(NAME).exe: $(NAME).o p44.o
    @echo "Generating" $@
    @gcc $^ $(LFLAGS) -o $@

# NUEVO
p44.o: p44.c
    @echo "Compiling" $< " and generating" $@
    @gcc $(CFLAGS) $< -o $@

$(NAME).o: lex.yy.c
    @echo "Compiling" $< " and generating" $@
    @gcc $(CFLAGS) $< $(LFLAGS) -o $@

lex.yy.c: $(NAME).l
    @echo "Generating" $@
    @flex $<

clean:
    @echo Deleting auxiliary files
    @rm -f lex.yy.c *.o
```

## Componentes léxicos de un lenguaje de programación (1ª versión)

- **Descripción**

- El analizador léxico reconoce algunos componentes léxicos de un lenguaje de programación.
- Finaliza el programa cuando se teclea el carácter # al principio de la línea.

- **Compilación**

- Se crea el fichero ejemplo5.exe  
*make NAME= ejemplo5*

- **Ejecución**

```
$ ./ejemplo5.exe  
if (a>0) then b = a; else b = -a;  
Palabra reservada: if --> token 257  
(  
Identificador: a --> token 260  
Operador relacional: > --> token 262  
Numero: 0 --> token 261  
)  
Palabra reservada: then --> token 258  
Identificador: b --> token 260  
=  
Identificador: a --> token 260  
;  
Palabra reservada: else --> token 259  
Identificador: b --> token 260  
=-  
Identificador: a --> token 260  
;  
#  
#  
#  
Fin del programa  
$
```

- **Ficheros en el directorio ejemplo5**

- makefile
- ejemplo5.h
- ejemplo5.l

- **makefile**

```
# Si se desea generar ejemplo5.exe entonces teclear
# make
#
# Si se desea generar otro "ejemplo" entonces teclear
#         make NAME=ejemplo
NAME = ejemplo5

# Directives for the compiler
# -c: the output is an object file, the linking stage is not
done.
# -g: debug
# -Wall: all warnings
# -ansi: standard language
# -O2: optimization level
# -Wno-unused-function -Wno-sign-compare -Wno-implicit-
function-declaration: disabled error messages
CFLAGS = -c -g -Wall -ansi -O2 -Wno-unused-function -Wno-sign-
compare -Wno-implicit-function-declaration
# Directive for flex's library
LFLAGS = -lfl

# Predefined macros
#
# $@: name of the target
# $^: all the dependencies
# $<: first dependency
#
#####
$(NAME).exe: $(NAME).o
    @echo "Generating" $@
    @gcc $< $(LFLAGS) -o $@

$(NAME).o: lex.yy.c
    @echo "Compiling" $< "and generating" $@
    @gcc $(CFLAGS) $< -o $@
    @echo "--> Deleting the auxiliary file: lex.yy.c"
    @rm -f lex.yy.c

lex.yy.c: $(NAME).l
    @echo "Generating" $@
    @flex $<

clean:
    @echo "--> Deleting object files"
    @rm -f lex.yy.c *.o
```

- o **ejemplo5.h**

```
#define IF          257
#define THEN        258
#define ELSE        259
#define ID          260
#define NUMERO      261
#define MAYOR_QUE   262
#define MAYOR_IGUAL 263
```

- o **ejemplo5.l**

```
/* ***** ejemplo5.l ***** */
/* ***** Zona de las declaraciones o definiciones ***** */
%{
    /* Descripción:

        * El analizador léxico reconoce algunos componentes
          léxicos de un lenguaje de programación

    */

    /* Fichero que contiene la declaración de los componentes
    léxicos o tokens */
    #include "ejemplo5.h"
}%

/* Definiciones regulares */
espacio      [ \t\n]
espacios     {espacio}+
letra        [a-zA-Z]
digito       [0-9]
identificador {letra}({letra}|{digito})*
numero       {digito}+(\.{digito}+)?(E[+\-]?{digito}+)?

/* ***** Zona de las reglas ***** */
%%
{espacios} { /* no se hace nada */ ; }

if          {printf("\n Palabra reservada: %s --> token %d\n",
                  yytext, IF);}

then        {printf("\n Palabra reservada: %s --> token %d\n",
                  yytext, THEN);}

else        {printf("\n Palabra reservada: %s --> token %d\n",
                  yytext, ELSE);}

{identificador}{printf("\n Identificador: %s --> token %d\n",
```



```

        yytext, ID);}

{numero}      {printf("\n Numero: %s --> token %d\n",
                    yytext,NUMERO);}

">"          {printf("\n Operador relacional: %s --> token %d\n",
                    yytext,MAYOR_QUE);}
">="         {printf("\n Operador relacional: %s --> token %d\n",
                    yytext,MAYOR_IGUAL);}

^#            {printf("\n Fin del programa \n"); return 0;}

.             { ECHO; /* Se muestra por pantalla,
                    pero no se indica nada */ }

%%

/***** Zona de funciones auxiliares *****/

```

## Componentes léxicos de un lenguaje de programación (2ª versión)

- **Descripción**

- Permite el uso de argumentos desde la línea de comandos para indicar el fichero que se desea analizar.

- **Compilación**

- Se crea el fichero ejemplo55.exe  
*make NAME= ejemplo55*

- **Ejecución**

```
$ ./ejemplo55.exe entrada.txt  
Palabra reservada: if --> token 257  
(  
Identificador: dato --> token 260  
Operador relacional: > --> token 262  
Numero: 0 --> token 261  
)  
Identificador: valor --> token 260  
=  
Identificador: dato --> token 260  
;  
Palabra reservada: else --> token 259  
Identificador: valor --> token 260  
=-  
Identificador: dato --> token 260
```

- **Ficheros en el directorio ejemplo5**

- makefile: mismo fichero del ejemplo anterior
- ejemplo5.h: mismo fichero del ejemplo anterior
- entrada.txt
- ejemplo55.l

- **entrada.txt**

```
if (dato > 0)  
    valor = dato;  
else  
    valor = - dato;
```

○ ejemplo55.1

```
/****** ejemplo55.1 *****/
/****** Zona de las declaraciones o definiciones *****/
%{
    /* Descripción:
       * El analizador léxico reconoce elementos
       * de un lenguaje de programación
       * Permite leer desde un fichero
       * y escribir la salida en otro fichero
    */
#include <stdio.h>
#include "ejemplo5.h" /* declaración de los componentes
                        léxicos o tokens */
%}
/* Definiciones regulares */

espacio      [ \t\n]
espacios     {espacio}+
letra        [a-zA-Z]
digito       [0-9]
identificador {letra}({letra}|{digito})*
numero       {digito}+(\.{digito}+)?(E[+\-]?{digito}+)?

/****** Zona de las reglas *****/
%%
{espacios}    { /* no se hace nada */ ; }

if    {fprintf(yyout, "\n Palabra reservada: %s --> token %d\n",
               yytext, IF);}

then {fprintf(yyout, "\n Palabra reservada: %s --> token %d\n",
               yytext, THEN);}

else {fprintf(yyout, "\n Palabra reservada: %s --> token %d\n",
               yytext, ELSE);}

{identificador}
    {fprintf(yyout, "\n Identificador: %s --> token %d\n",
              yytext, ID);}

{numero} {fprintf(yyout, "\n Numero: %s --> token %d\n",
                  yytext, NUMERO);}

">" {fprintf(yyout, "\n Operador relacional: %s --> token %d\n",
              yytext, MAYOR_QUE);}

">=" {fprintf(yyout, "\n Operador relacional: %s --> token %d\n",
              yytext, MAYOR_IGUAL);}
```

```

^#      {fprintf(yyout, "\n Fin del programa \n"); return 0;}
.      { ECHO; /* Se muestra por pantalla,
                pero no se indica nada */ }
%%

/***** Zona de funciones auxiliares *****/

extern FILE *yyin, *yyout;

int main(int argc, char *argv[])
{
    switch(argc)
    {
        case 2:  yyin=fopen(argv[1], "r");
                  break;

        case 3:  yyin=fopen(argv[1], "r");
                  yyout=fopen(argv[2], "w");
    }

    yylex();

    return 0;
}

```

## Comando REJECT

- **Descripción**
  - Muestra el uso de **REJECT**
  - Permite que un texto de entrada pueda ser asociado a más de una expresión regular.
  - Después de emparejar el texto de entrada con una expresión regular, la rechaza para comprobar si el texto puede asociarse a otra expresión regular.

- **Compilación**

- Se crea el fichero pink.exe  
*make NAME= pink*

- **Ejecución**

```
$/pink.exe  
pink  
pink  
int  
Ctrl^D  
Contador de palabras  
pink = 1  
ink = 1  
pin = 2
```

- **Fichero**

- pink.l

```
/****** pink.l *****/  
***** Zona de las declaraciones o definiciones *****/  
%{  
    /* Descripción  
        Se muestra el uso de REJECT  
    */  
    /* Variables globales */  
    int n_pink = 0;  
    int n_ink = 0;  
    int n_pin = 0;  
}%  
  
***** Zona de las reglas *****/  
%%  
pink      {n_pink++; REJECT;}  
ink       {n_ink++; REJECT;}  
pin       {n_pin++; REJECT;}  
.|\\n ;   /* Se descartan el resto de caracteres */  
%%
```

```

/***** Zona de funciones auxiliares *****/
extern FILE *yyin, *yyout;

int main(int argc, char *argv[])
{
    switch(argc)
    {
        case 2: yyin=fopen(argv[1], "r");

                break;

        case 3: yyin=fopen(argv[1], "r");

                yyout=fopen(argv[2], "w");
    }

    yylex();

    fprintf(yyout, "Contador de palabras\n");

    fprintf(yyout, "\t pink = %d\n", n_pink);

    fprintf(yyout, "\t ink = %d\n", n_ink);

    fprintf(yyout, "\t pin = %d\n", n_pin);

    return 0;
}

```

## Función yymore()

- **Descripción**
  - Muestra el uso de `yymore()`
  - Permite concatenar el siguiente texto que sea reconocido con el contenido actual de `yytext`.
- **Compilación**
  - Se crea el fichero `ejemplo0.exe`  
`make NAME= ejemplo0`

- **Ejecución**

```
$ ./hiper.exe
texto
Token = texto
hipertexto
Token = hipertexto
mercado
Token = mercado
hipermercado
Token = hipermercado
enlace
hiperenlace
```

- **Fichero**

- `hiper.l`

```
/** hipers.l */
/** Zona de las declaraciones o definiciones */
%{
    /* Descripción
       Se muestra el uso de yymore()
    */
}%

/***** Zona de las reglas *****/
%%

hiper    yymore();
texto    printf("Token = %s\n",yytext);
mercado  printf("Token = %s\n",yytext);

.|\\n ; /* Se descartan el resto de caracteres */

%%
```

```

/***** Zona de funciones auxiliares *****/

extern FILE *yyin, *yyout;

int main(int argc, char *argv[])
{
    switch(argc)
    {
        case 2: yyin=fopen(argv[1], "r");

                break;

        case 3: yyin=fopen(argv[1], "r");

                yyout=fopen(argv[2], "w");
    }

    yylex();

    return 0;
}

```



## Reconocimiento de comentarios

- Descripción

- El analizador léxico reconoce y cuenta los comentarios del lenguaje C
- Muestra el comentario reconocido y el número de líneas.
- Muestra el uso de estados de Flex
  - Comando **BEGIN**
  - Reglas condicionales controladas por un estado de flex definido por el programador

- Compilación

- Se crea el fichero comentario.exe  
*make NAME= comentario*

- Ejecución

```
$ ./comentario.exe
```

```
/* Comentario de una línea */
```

```
nº comentario = 1, lineasComentario = 1
```

```
/*
```

```
Comentario
```

```
de
```

```
varias
```

```
líneas
```

```
*/
```

```
nº comentario = 2, lineasComentario = 6
```

```
/*
```

```
Comentario anidado
```

```
/*
```

```
No se pueden anidar comentarios
```

- Fichero

- comentario.l

```
%{
```

```
/* Descripción:
```

```
El analizador léxico reconoce
```

```
y cuenta los comentarios del lenguaje C
```

```
Se muestra el uso de
```

```
BEGIN
```

```
reglas condicionales controladas por un estado de flex  
definido por el programador
```

```
*/
```

```
/* Indica el número de comentario */
```

```
int comentario=0;
```

```

/* Indica el número de líneas de cada comentario */
int lineasComentario;
%}

/* Definiciones regulares */
letra      [a-zA-Z]
digito     [0-9]
identificador {letra}({letra}|{digito})*

/* Se define un nuevo estado: ESTADO_COMENTARIO */
%x ESTADO_COMENTARIO

%%

{identificador} { printf("identificador = %s\n", yytext); }

"/*" {
    /* Se activa el reconocimiento de un comentario */
    BEGIN ESTADO_COMENTARIO;
    comentario++;
    lineasComentario=1;
}

<ESTADO_COMENTARIO>"/*"/(.\|\\n)
{
    printf("\\n No se pueden anidar comentarios\\n");
    return 0;
}

<ESTADO_COMENTARIO>"*/"
{
    /* Fin del comentario:
       se vuelve al estado por defecto */
    BEGIN 0;
    printf("\\n nº comentario = %d, lineasComentario =
%d\\n",comentario,
        lineasComentario);
}

<ESTADO_COMENTARIO>.      {;}

<ESTADO_COMENTARIO>\\n    {lineasComentario++;}

.                          {ECHO; }
%%

```

## AFD-Comentarios1.l

- **Descripción**

- El analizador léxico reconoce y cuenta los comentarios del lenguaje C
- No muestra el comentario reconocido ni el número de líneas.
- Muestra el uso de estados de Flex
  - Comando BEGIN
  - Reglas condicionales controladas por estados de flex definidos por el programador

- **Compilación**

- Se crea el fichero AFD-Comentarios1.exe  
*make NAME= AFD-Comentarios1*

- **Ejecución**

```
$ ./AFD-Comentarios1.exe
```

```
/* Comentario de una línea */
```

```
Comentario reconocido
```

```
/*
```

```
Comentario
```

```
de
```

```
varias
```

```
líneas
```

```
*/
```

```
Comentario reconocido
```

```
/*
```

```
Comentario anidado
```

```
/*
```

```
Error: comentario anidado
```

- **Fichero**

- AFD-Comentarios1.l

```
/*
```

```
Adaptado de
```

```
A [f]lex tutorial (Powerpoint slides)
```

```
http://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/lex%20tutorial.ppt
```

```
En
```

```
http://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/index.html
```

```
*/
```

```

/***** Zona de las declaraciones o definiciones *****/
%{

%}

/* ESTADOS DE FLEX: estados del autómata */
%x Q1 Q2 Q3 Q4

%%
"/"          BEGIN(Q1); /* change to Q1 */

<Q1>"*"      BEGIN(Q2); /* change to Q2 */

<Q2>[^*/]    ;          /* stay in Q2 */
<Q2>"*"      BEGIN(Q3); /* change to Q3 */
<Q2>"/"      BEGIN(Q4); /* change to Q4 */

<Q3>"*"      ;          /* stay in Q3 */
<Q3>[^*/]    BEGIN(Q2); /* change to Q2 */
<Q3>"/"      {
                printf("\nComentario reconocido\n");

                /* change to INITIAL: default state */
                BEGIN(INITIAL);
            }

<Q4>[^*]     BEGIN(Q2); /* change to Q2 */
<Q4>"*"      {
                printf("\nError: comentario anidado\n");

                /* change to INITIAL: default state */
                BEGIN(INITIAL);
            }

(.|n)        ECHO;
%%

```

## AFD-Comentarios2.l

- **Descripción**

- El analizador léxico reconoce y cuenta los comentarios del lenguaje C
- Muestra el comentario reconocido.
- Muestra el uso de estados de Flex
  - Comando BEGIN
  - Reglas condicionales controladas por un estado de flex definido por el programador
  - Función yymore()

- **Compilación**

- Se crea el fichero AFD-Comentarios2.exe  
*make NAME= AFD-Comentarios2*

- **Ejecución**

```
$ ./AFD-Comentarios2.exe
```

```
/* Comentario de una línea */
```

```
Comentario reconocido: /* Comentario de una línea */
```

```
/*
```

```
Comentario
```

```
de
```

```
varias
```

```
líneas
```

```
*/
```

```
Comentario reconocido: /*
```

```
Comentario
```

```
de
```

```
varias
```

```
líneas
```

```
*/
```

```
/*
```

```
Comentario anidado
```

```
/*
```

```
Error: comentario anidado
```

- Fichero
  - AFD-Comentarios2.l

```

/*
Adaptado de
A [f]lex tutorial (Powerpoint slides)

http://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/lex%20tuto
rial.ppt

En
http://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/index.html
*/

/***** Zona de las declaraciones o definiciones *****/
%{

%}

/* ESTADOS DE FLEX: estados del autómata */
%x Q1 Q2 Q3 Q4

%%
"/"      {
          yynore ();
          BEGIN (Q1); /* change to Q1 */
        }

<Q1>"*"  {
          yynore ();
          BEGIN (Q2); /* change to Q2 */
        }

<Q2>[^*/] {
          yynore (); /* stay in Q2 */
        }

<Q2>"*"  {
          yynore ();
          BEGIN (Q3); /* change to Q3 */
        }

<Q2>"/"  {
          yynore ();
          BEGIN (Q4); /* change to Q4 */
        }

<Q3>"*"  {
          yynore (); /* stay in Q3 */
        }

```

```

<Q3>[^*/]      {
    yymore();
    BEGIN(Q2); /* change to Q2 */
}

<Q3>"/"        {
    printf("\nComentario reconocido: %s\n",
           yytext);

    /* change to INITIAL: default state */
    BEGIN(INITIAL);
}

<Q4>[^*]       {
    yymore();
    BEGIN(Q2); /* change to Q2 */
}

<Q4>"*"        {
    printf("\nError: comentario anidado:%s\n",
           yytext);

    /* change to INITIAL: default state */
    BEGIN(INITIAL);
}

(.|n)          ECHO;
%%

```

## Reconocimiento de cadenas, primera versión

- **Descripción**
  - Reconoce cadenas delimitadas por comillas simples
- **Compilación**
  - Se crea el fichero cadena\_1.exe  
*make NAME= cadena\_1*

- **Ejecución**

*\$ ./cadena\_1.exe*

*'Ejemplo de cadena'*

*Cadena reconocida = 'Ejemplo de cadena'*

*'Cadena*

*escrita*

*en varias líneas'*

*Cadena reconocida = 'Cadena*

*escrita*

*en varias líneas'*

*'Cadena con \'comillas\' internas'*

*Cadena reconocida = 'Cadena con \'comillas\' internas'*

- **Fichero en el directorio Cadenas**

- cadena\_1.l

```
%{
/* Descripción:
El analizador léxico reconoce
cadenas delimitadas por comillas simples
*/
}%

/* Definiciones regulares */
letra      [a-zA-Z]
digito     [0-9]
identificador {letra}({letra}|{digito})*
cadena     "'"([^'|"\\\'"])*'"

%%

{identificador} {printf("identificador = %s\n", yytext);}
```



```
{cadena} {  
    /* Se ha reconocido una cadena */  
    printf("\t Cadena reconocida = %s\n",yytext);  
}  
  
. {ECHO;}  
  
%%
```

## Reconocimiento de cadenas, segunda versión

- **Descripción**

- Reconoce y cuenta cadenas delimitadas por comillas simples
- Numera las cadenas e indica cuántas líneas contiene.
- Se muestra el uso de
  - Comando BEGIN
  - Reglas condicionales controladas por un estado de flex definido por el programador
  - Función yymore()

- **Compilación**

- Se crea el fichero cadena\_2.exe  
*make NAME= cadena\_2*

- **Ejecución**

```
$/cadena_2.exe  
'Ejemplo de cadena'
```

```
nº cadena = 1, lineas_cadenas = 1  
Cadena reconocida = Ejemplo de cadena
```

```
'Cadena escrita  
en  
varias líneas'
```

```
nº cadena = 2, lineas_cadenas = 3  
Cadena reconocida = Cadena escrita  
en  
varias líneas
```

```
'Cadena con \'comillas\' internas'
```

```
nº cadena = 3, lineas_cadenas = 1  
Cadena reconocida = Cadena con \'comillas\' internas
```

- **Fichero en el directorio Cadenas**

- cadena\_2.l

```
%{
```

```
/* Descripción:  
El analizador léxico reconoce  
y cuenta cadenas delimitadas por comillas simples  
Se muestra el uso de  
- BEGIN  
- reglas condicionales controladas  
por un estado de flex definido por el programador  
- yymore()
```

```

    */

    /* Indica el número de comentario */
    int numero_cadenas=0;

    /* Indica el número de líneas de cada comentario */
    int lineas_cadenas;
}%}

/* Definiciones regulares */
letra      [a-zA-Z]
digito     [0-9]
identificador {letra}({letra}|{digito})*

/* Se define un nuevo estado de Flex: ESTADO_CADENA */
%x ESTADO_CADENA

%%

{identificador}    {printf("identificador = %s\n", yytext);}

""                {
                    /* Se activa el reconocimiento de la cadena */
                    BEGIN ESTADO_CADENA;
                    numero_cadenas++;
                    lineas_cadenas=1;
                }

<ESTADO_CADENA>""
    { /* Fin de la cadena:
      se vuelve al estado por defecto */
      BEGIN 0;

      printf("\n n° cadena = %d, lineas_cadenas = %d\n",
        numero_cadenas, lineas_cadenas);

      yytext[yytextleng-1]='\0';

      printf("\t Cadena reconocida = %s\n",yytext);
    }

<ESTADO_CADENA>"\\\\"    {yytext++;}

<ESTADO_CADENA>.\n      {yytext++;}

<ESTADO_CADENA>.\n      {lineas_cadenas++; yytext++;}

.                        {ECHO;}

%%

```

## AFD-Cadenas\_1

- **Descripción**
  - Reconoce cadenas delimitadas por comillas simples
  - No muestra la cadena reconocida
  - Se simula el funcionamiento de un autómata finito determinista (AFD)
  - Se muestra el uso de
    - Comando BEGIN
    - Reglas condicionales controladas por estados de flex definidos por el programador
- **Compilación**
  - Se crea el fichero AFD-Cadenas\_1.exe  
*make NAME= AFD-Cadenas\_1*
- **Ejecución**

```
$/AFD-Cadenas_1.exe
'Ejemplo de cadena'
    Cadena reconocida

'Cadena
escrita
en varias líneas'
    Cadena reconocida

'Cadena con \'comillas\' internas'
    Cadena reconocida
```
- **Fichero en el directorio Cadenas**
  - AFD-Cadenas\_1.l

```
/*
Autómata que reconoce cadenas delimitadas por comillas simples,
pero no las muestra
```

```
Adaptado de
A [f]lex tutorial (Powerpoint slides)
```

```
http://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/lex%20tutorial.ppt
```

```
En
http://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/index.html
*/
```

```
/****** Zona de las declaraciones o definiciones *****/
%{

%}
```

```

/* ESTADOS DE FLEX: estados del autómata */
%x Q1 Q2 Q3 Q4

%%
"'"
    BEGIN(Q1);      /* change to Q1 */

<Q1>[^'\\""]      ;      /* stay in Q1 */
<Q1>""\\"      BEGIN(Q2);      /* change to Q2 */
<Q1>""'"      {
    printf("\nCadena reconocida\n");

    /* change to INITIAL: default state */
    BEGIN(INITIAL);      }
}

<Q2>[^\\"]      BEGIN(Q1);      /* change to 1 */
<Q2>"/"      ;      /* stay in Q2 */

(.|n)      ECHO;
%%

```

## AFD-Cadenas\_2

- **Descripción**
  - Reconoce y cuenta cadenas delimitadas por comillas simples
  - Muestra la cadena reconocida
  - Se simula el funcionamiento de un autómata finito determinista (AFD)
  - Se muestra el uso de
    - Comando BEGIN
    - Reglas condicionales controladas por estados de flex definidos por el programador
    - Función yymore()
- **Compilación**
  - Se crea el fichero AFD-Cadenas\_2.exe  
*make NAME= AFD-Cadenas\_2*
- **Ejecución**

```
$ ./AFD-Cadenas_1.exe  
'Ejemplo de cadena'
```

```
nº cadena = 1, lineas_cadenas = 1  
Cadena reconocida = Ejemplo de cadena
```

```
'Cadena escrita  
en  
varias líneas'
```

```
nº cadena = 2, lineas_cadenas = 3  
Cadena reconocida = Cadena escrita  
en  
varias líneas
```

```
'Cadena con \'comillas\' internas'
```

```
nº cadena = 3, lineas_cadenas = 1  
Cadena reconocida = Cadena con \'comillas\' internas
```

- **Fichero en el directorio Cadenas**
  - AFD-Cadenas\_2.l

```
/*  
Autómata que reconoce cadenas delimitadas por comillas simple y  
las muestra
```

```
Adaptado de  
A [f]lex tutorial (Powerpoint slides)
```

<http://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/lex%20tutorial.ppt>

En

<http://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/index.html>  
\*/

```
/* **** Zona de las declaraciones o definiciones **** */
%{

%}

/* ESTADOS DE FLEX: estados del autómata */
%x Q1 Q2

%%
"'" {
    /* yymore(); */
    BEGIN(Q1); /* change to Q1 */
}

<Q1>[^'\\] {
    yymore();
    /* stay in Q1 */
}

<Q1>"\\ " {
    yymore();
    BEGIN(Q2); /* change to Q2 */
}

<Q1>"'" {
    yytext[yytextlen-1]='\0';
    printf("\nCadena reconocida= %s\n",yytext);

    /* change to INITIAL: default state */
    BEGIN(INITIAL);
}

<Q2>[^\\] {
    yymore();
    BEGIN(Q1); /* change to 1 */
}

<Q2>"/ " {
    yymore();
    /* stay in Q2 */
}

(.|n) ECHO;
%%
```