

**Práctica 1:**

# **Analizadores léxicos con `lex`**

---

Juan Pedro Bolívar Puente y Francisco Manuel Herrero Pérez

Ingeniería Informática  
*Universidad de Granada*

Diciembre 2007

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. El programa</b>	<b>2</b>
2.1. Estructura general . . . . .	2
2.2. Limitaciones de memoria . . . . .	3
2.3. Restaurando sesiones . . . . .	5
<b>3. Análisis léxico</b>	<b>5</b>
3.1. Expresiones regulares . . . . .	5
3.1.1. Primera solución . . . . .	6
3.1.2. Síntesis . . . . .	7
3.2. Reentrancia . . . . .	11
<b>4. Conclusión</b>	<b>12</b>
<b>Apéndices</b>	<b>13</b>
<b>A. Código primera propuesta</b>	<b>13</b>

# 1. Introducción

La siguiente práctica tiene como propósito familiarizarse con el generador de analizadores léxicos Lex. Con él, se pretende realizar un programa dónde parte del código sea un lexer generado a partir de una serie de expresiones regulares.

El programa, al que hemos llamado `mailcrawl`, es un buscador de emails en la red, como los que posiblemente usen los perversos emisores de spam. Al programa le pasamos como parámetros una serie de direcciones web y este añadirá a un fichero todas las direcciones de correo que encuentre, a la vez que va siguiendo los enlaces que encuentre en la web.

En éste documento describiremos en un primer punto los algoritmos y la estructura general del programa para luego centrarnos en el análisis léxico, que es el interés principal de la práctica.

## 2. El programa

### 2.1. Estructura general

El algoritmo 1 describe el funcionamiento general del programa y está implementado en `mailcrawl_run()`.

Para recorrer el gran gafo que es la red, dónde cada página es un nodo y cada hipervínculo es una arista, se ha elegido realizar una búsqueda en anchura. Desde un punto de vista técnico es indiferente realizar una búsqueda en profundidad o en anchura, ya que cambia sólo utilizar una estructura LIFO o FIFO para almacenar los hipervínculos encontrados en cada página web. Sin embargo, creemos que es mucho más conveniente realizar la búsqueda en anchura para obtener mejores resultados. Esto se debe a que lo más probable es que la sección de una web dónde más correos electrónicos hay probablemente no sea el primer enlace de la web. Como la web es, a efectos prácticos, *infinita*, en una búsqueda en profundidad casi siempre seguiríamos los primeros enlaces de la web, perdiendonos en nuestra búsqueda sin realizar hallazgos importantes; es mucho más fructuoso que una vez nos dirigimos a un sitio web intentemos analizarlo entero antes de saltar a otro.

---

**Algoritmo 1** *mailcrawl*

---

**Entrada:** *urls\_iniciales*: Las direcciones dónde empezar a buscar.

**Salida:** *salida*: El conjunto de direcciones de correo electrónico encontrados.

```
1: salida  $\leftarrow \emptyset$ 
2: cola  $\leftarrow \emptyset$ 
3: push(cola, urls_iniciales)
4: mientras  $|cola| > 0$  hacer
5:   fichero  $\leftarrow$  descargar(pop(cola))
6:   para todo email  $\in$  (fichero  $\cap$  emails validos) hacer
7:     salida  $\leftarrow$  salida  $\cup$  email
8:   fin para
9:   para todo url  $\in$  (fichero  $\cap$  urls validas) hacer
10:    push(cola, url)
11:  fin para
12: fin mientras
13: return emails
```

*Nota:* *emails validos* y *urls validas* son respectivamente los lenguajes de tercer orden que contienen a *todos* -ya matizaremos esto más adelante- los emails y urls posibles, definidos a través de las expresiones regulares que veremos más adelante.

---

Para descargar las páginas web hemos utilizado la librería `libcurl`<sup>1</sup> que implementa los principales protocolos de transferencia de ficheros. El hecho de que esta librería nos abstraiga de los detalles de éstos protocolos puede suponer un inconveniente ya que, por ejemplo, no podemos saber si una URL se corresponde con un directorio o con un fichero, ya que la librería descargaría el `index.html` del directorio sin avisarnos de que la URL no era un fichero. Ésto puede ocasionarnos problemas a la hora de conocer la ruta real a un fichero para deshacer los enlaces relativos, aunque añade discusión al apartado léxico que es lo que nos interesa al fin y al cabo.

## 2.2. Limitaciones de memoria

Por otro lado, es necesario encontrar un método para evitar entrar en un bucle sin fin en un ciclo del grafo. Para ello debemos guardar todas las

---

<sup>1</sup><http://curl.haxx.se/libcurl/>

páginas visitadas en una estructura para después comprobar que las nuevas páginas que metamos en la cola no han sido ya visitadas antes. También se comprueba que un correo no esté duplicado, algo que ocurre más frecuentemente de lo que podría parecer, ya que son frecuentes cosas como `<a href="mailto:pepito@menganito.com">pepito@menganito.com</a>`.

Se ha elegido con éste fin un árbol *red-black* ya que provee tiempos de búsqueda, inserción y borrado logarítmicos con unas constantes ocultas ligeramente mejores que la de los árboles *avl*. Ese tipo de estructuras están en la librería estándar de C++, pero no en C. Como implementarlas desde cero sería un trabajo tedioso que se sale de los propósitos de ésta práctica, hemos decidido tomarlas de la GNUlib<sup>2</sup>. Éste hecho nos ha obligado a entregarle el programa bajo la licencia GPL v3<sup>3</sup>

También hay que tener en cuenta que mientras la red es virtualmente infinita la memoria principal de un computador no lo es. Por este motivo se establece un límite de entradas a guardar en esta caché. Cuando el tamaño de la caché supera el límite reducimos su tamaño a la mitad, para evitar que esta operación deba hacerse con demasiada frecuencia ya que tiene complejidad lineal. Para poder realizar esta limpieza guardamos un índice en cada nodo que refleja el orden en el que fué añadido, es decir es un entero entre 0 y *tamano\_cache*. Al realizar el reajuste borramos los elementos tal que  $indice < tamantiguo - (tamantiguo - tamnuevo)$  y actualizamos los índices de los que quedan para que sigan estando en el rango  $[0, tamnuevo]$ . El código puede estudiarse `cache_adjust()`.

Por otro lado, también hay que controlar el tamaño de la cola. No sería una solución viable no permitir que entren más elementos cuando se ha llenado, ya que esto degeneraría en algo parecido a la búsqueda en profundidad ya que sólo añadiríamos a la cola el primer enlace de cada página. Lo que haremos será no permitir que entren más elementos en la cola hasta que esta esté a la mitad de su capacidad límite, para seguir llenándola entonces.

---

<sup>2</sup><http://www.gnu.org/software/gnulib/>

<sup>3</sup>Aunque nosotros, partidarios del software libre, lo hacemos con gusto, esperamos que esto no suponga ningún problema respecto a la entrega de la práctica.

## 2.3. Restaurando sesiones

Pensamos que, como el proceso de búsqueda de emails puede ser muy lento y largo, puede ser interesante que alguien quiera parar una sesión en cualquier momento y continuarla en otro. Así, con el parametro `resume` podemos especificar un fichero de dónde se tomarán las URLs iniciales. Al finalizar la ejecución del programa se guardará en ese fichero el estado de la cola.

La sintáxis de este fichero es una secuencia de URLs, una por cada línea, envueltas en comillas simples. El propio analizador léxico normal se usa para cargar el fichero, así que en realidad se puede introducir cualquier entrada válida para éste.

Por otro lado, para que la finalización del programa se realice de forma limpia y se vuelque la cola en el fichero capturamos las señales `SIGINT` y `SIGTERM`, para poder cerrar el programa simplemente pulsando `ctrl+c` en la consola. Normalmente esto ocurre mientras se descarga una web, por lo que puede que transcurran un par de segundos antes de que se cierre el programa ya que no podemos asegurarnos de que la señal se ha emitido en un estado seguro de las estructuras que manejamos y no podemos cerrar directamente.

## 3. Análisis léxico

### 3.1. Expresiones regulares

En el guión de prácticas se pedía desarrollar por separado la solución los integrantes del grupo.

Después de debatir sobre el problema un poco por encima, nos pusimos manos a la obra por separado. Francisco Manuel hizo un primer prototipo que buscaba URLs completas y correos electrónicos en texto plano. Por otro lado, Juan Pedro presentó una versión previa de lo que sería el software definitivo que ya usaba `libcurl` y con unas expresiones regulares parecidas a las definitivas, que admitían también hipervínculos relativos y estaban un poco más modularizadas.

Vamos a exponer primero la explicación de Francisco Manuel sobre su solución y luego debatiremos la solución final tomada. Aunque en Francisco Manuel también hizo aportaciones a ésta solución final, no expon-

dremos por separado la solución de Juan Pedro ya que es parecida para evitar prolongar demasiado el documento con información redundante.

### 3.1.1. Primera solución

Nuestras primeras ideas para la realización del trabajo fueron desde el principio sobre la búsqueda de expresiones relacionadas con el mundo de internet, búsqueda de url, correos, determinado código HTML, etc. Decidimos por tanto hacer un programa que buscara hipervínculos y correos electrónicos.

El código completo de mi propuesta puede consultarse en el anexo 1. A continuación explico sus expresiones regulares.

Al realizar las expresiones regulares para los hipervínculos he tenido en cuenta la diferenciación entre mayúsculas y minúsculas en html.

Para los hipervínculos y correos he usado las siguientes expresiones:

```
\ "(h|H)(t|T)(t|T)(p|P)\: [^\"]*\"
```

Busco la expresión "http:" y una vez encontrada añado todo lo que haya hasta que se encuentra con las " que cierran el hipervínculo.

```
{file_char}+@{file_char}+
```

Busco cualquier expresión que sea letras o números, una @, y que después sigan letras o números.

Uso la definición `file_char` que coge todas las letras minúsculas o mayúsculas y cualquier número.

```
file_char ([A-Za-z][0-9])
```

Todas las reglas ejecutan el comando ECHO que es una macro de lex que escribe todas las cadenas encontradas en el archivo de salida, al que ha sido dirigido salida.

Más tarde, mi compañero me enseñó el programa que estaba creando y lo que quería conseguir con él, y tras pensar que sería más interesante

lo segundo, nos centramos en su idea, y en ver como podíamos llevarla a cabo.

Las expresiones regulares debían de ser más y mucho mas concretas, diferenciando entre url base, y url relativa, y la del correo mas concisa a la hora de buscar, y no simplemente limitarse a buscar la arroba entre dos cadena de caracteres. En la parte de mi compañero se hablara sobre las expresiones regulares que fuimos creando para dicho programa y su evolución.

### 3.1.2. Síntesis

A continuación van a describirse las expresiones regulares de la solución definitiva y el por qué de ellas y su estructura. No se incluye en los anexos el código de entrada de `lex` de ésta solución ya que puede consultarse en `mailcrawl_lex.l` entre el código fuente que acompaña a este documento.

Primero vamos a describir las definiciones globales. Vayamos por partes, primero tenemos<sup>4</sup>:

```
ext:
([xX]?[hH][tT][mM][lL]?|[xX][mM][lL]|[pP][hH][pP][1-5]?|
[aA][sS][pP][xX]|[tT][xX][tT])

proto:
([hH][tT][tT][pP][sS]?|[fF][tT][pP])

subdomain:
([wW]{3})[[:alnum:]]?

basedomain:
[[:alpha:]]{2}|([cC][oO][mM]|[oO][rR][gG]|[nN][eE][tT]|
[gG][oO][vV]|[bB][iI][zZ]|[iI][nN][fF][oO]|[nN][aA][mM][eE]|
[aA][eE][rR][oO]|[bB][iI][zZ]|[iI][nN][fF][oO]|[jJ][oO][bB][sS]|
[mM][uU][sS][eE][uU][mM])
```

La `ext` describe las posibles extensiones de los tipos de ficheros que queremos recorrer -html, php, etc.-, tanto en mayúsculas como en minúsculas, ya que las URLs son *case-insensitive*. `proto` son los protocolos que aceptaremos en las URLs, que son casi todos los válidos para `libcurl`.

---

<sup>4</sup>Se describirán las definiciones como `nombre:<nueva linea>expresión regular` ya que algunas son muy largas y no caben en una sola linea.



`subdomain` nos indica los subdominios usados normalmente para el servidor web de una página. Esto lo necesitamos porque a menudo se especifica directamente una URL sin el protocolo, pero tenemos que saber que no es una URL relativa ya que comúnmente se asume que el protocolo está implícito en el `www`. Al `www` le añadimos una letra o número opcionales ya que muchas webs usan esto para distribuir la carga entre distintos servidores.

`basedomain` indica el conjunto de dominios base de uso general aprobados por la ICANN, que son: `com`, `org`, `net`, `gov`, `biz`, `info`, `name`, `jobs` y `museum`. Los códigos de países se codificarán directamente como dos caracteres alfanuméricos, ya que estos varían continuamente y sería tedioso incluir los cientos de códigos válidos.

Ahora vamos a ver algunos conjuntos de caracteres útiles:

```
dir_char      [[:alnum:]\_\-\b\~\=\+\&\*\%]
file_char     {dir_char}|\.
domain_char   [[:alnum:]\_\-\.]
user_char     [[:alnum:]\_\-\.]
```

Lo primero que puede que nos sorprenda es que distinguimos entre `dir_char` y `file_char`. Esto se debe a que tendremos que distinguir cuando el último elemento del camino de una URL es un directorio -y el servidor nos ha redireccionado por ejemplo a un `index.html`- o un fichero para saber si la URL base que tendremos que componer con los ficheros relativos es la URL hasta el último `"/` o la URL más un `"/`. De esta forma, supondremos que los directorios no contienen puntos en sus nombres y que los ficheros sí, con lo que conseguiremos acertar en el 99 % por ciento de los casos.<sup>5</sup>

Finalmente tenemos en `domain_char` los caracteres válidos en un dominio, que están un poco más restringidos. `user_char` son los caracteres válidos en un nombre de usuario.

---

<sup>5</sup>Mientras escribo esta memoria he descubierto una forma para que `libcurl` nos proporcione la URL real del fichero descargado una vez aplicadas las redirecciones; es decir, `http://www.miweb.org` se convertiría en `http://www.miweb.org/index.html`. De esta forma, todas las URLs serían de ficheros y existiría una única forma de obtener las URL base para los enlaces relativos que funcionaría en el 100 % de los casos, simplificando además el código. Aunque sería trivial modificar el código para que funcione de esta forma, en lo que sigue seguiremos utilizando la versión antigua ya que hace más interesante el análisis de las expresiones regulares. Me haré una copia local del programa dónde sí aplicaré los cambios oportunos, que el profesor puede pedirnos si lo desea.

Notar que en realidad no nos hemos esforzado demasiado en asegurar que caracteres exáctamente son válidos y no, ya que la información que se encuentra es muy diversa y depende de las plataformas etcétera. Aunque excluyamos de nuestro lenguaje extraños casos válidos, estamos seguro de que con esto aceptamos la mayor parte de los casos (además, en las URL deben sustituirse muchos caracteres especiales por %código\_utf).

Ahora vamos a pasar ya algunas expresiones regulares interesantes que compondrán las subpartes de las expresiones definitivas:

```
ip_add      ([ :digit:]{1,3}\.){4}
domain      {domain_char}+\.{basedomain}
host        ({ip_add}|{domain})
user        {user_char}+(:"{user_char}+)?@
url_base    ({proto}:"//"{user}?)|({user}?{subdomain}\.){host}
params      "?"{file_char}*
file        {file_char}*"."{ext}
directory   {dir_char}+
path        ({file_char}|" / ")*"/"
```

Las tres primeras están claras qué son. Una dirección IP tiene la forma  $x.x.x.x$ , con  $x \in [0, 255]$ . Realmente podríamos restringir a que los números estén en el rango adecuado, pero complicaría bastante la expresión. Hay cosas que es mejor comprobar en código. La definición de `domain` y `host` también debería ser muy intuitiva. Precisamente es por esto que hemos modularizado el código de las expresiones regulares: podemos formar expresiones complejas como una URL a partir de partes sencillas.

`user` es el nombre de usuario de la conexión que debe especificarse antes del dominio con la forma `usuario:contraseña@dominio.com`, siendo el campo `:contraseña` opcional.

`url_base` es la primera parte de una URL absoluta. Como se ha dicho antes, se asume que `www.manolo.es` es una URL absoluta. Realmente eso no es cierto y uno en una página web, si incluye un enlace a esa URL, nuestro navegador asumirá que es una URL relativa, es decir, que queremos descargar un directorio con ese nombre. Sin embargo para determinar si una URL es de tipo directorio o fichero, como hemos discutido antes, tenemos que pasar la URL que el usuario del programa introduce por el lexer, y queremos que pueda introducir las webs con ese formato. Realmente, sospecho que casi podemos asegurar que no existen webs con enlaces relativos con ese formato, así que incluimos esto en la expresión regular para darle esa comodidad al usuario<sup>6</sup>.

---

<sup>6</sup>Con la solución comentada en la nota 5 no haría falta pasar las cadenas introducidas

`directory` y `file` son muy sencillas. Lo único a comentar es que en `file` forzamos a que la extensión esté entre las especificadas, ya que es absurdo buscar correos electrónicos en ficheros binarios, por ejemplo.

Por otro lado, podemos incluir parámetros para un script en `php` u otro lenguaje después del símbolo `?`, y `params`, nos sirve para identificarlos. Finalmente `path` constituye una ruta relativa o absoluta.

Finalmente, tenemos las expresiones regulares:

```
email      {user_char}+@{host}

url_dir    {url_base}({path}{directory}?{params}?)?
url_file   {url_base}{path}{file}{params}?

rel_dir    {path}{directory}{params}?
rel_file   {path}?{file}{params}?

%

{email}      mailcrawl_put_email(yyextra, yytext);
\"{url_dir}\"  mailcrawl_add_url(yyextra, take_quotes(yytext), URL_DIR);
\"{url_file}\"  mailcrawl_add_url(yyextra, take_quotes(yytext), URL_FILE);
\"{rel_dir}\"   mailcrawl_add_rel(yyextra, take_quotes(yytext), URL_DIR);
\"{rel_file}\"  mailcrawl_add_rel(yyextra, take_quotes(yytext), URL_FILE);

\n|. {}
```

Éstas son bastante intuitivas, ya que es simplemente poner las partes que hemos compuesto anteriormente para formar emails -`email`-, URLs absolutas a directorio -`url_dir`-, y a fichero -`url_file`- y enlaces relativos a directorio -`rel_dir`- y a fichero -`rel_file`. Ya está sobradamente justificado porque los enlaces se dividen en estas cuatro categorías. Aún así vamos a comentar un par de cosas.

Primero, las URLs se encuentran entrecomilladas ya que de lo contrario los espacios en blanco nos extenderían mucho más allá de dónde debemos. En los ficheros `html`, `php`, etc. las cadenas siempre van entrecomilladas, por lo que así encontraremos todos los enlaces. Una solución para obtener más precisión podría haber sido haber identificado el `<a href=' '...` pero no es necesario. Es más, es contraproducente, ya que en la era del AJAX y la web 2.0 es muy probable que muchos enlaces se encuentren ofuscados en código JavaScript.

---

por el usuario por el lexer y también solucionamos este problema.

Por otro lado, en los directorios relativos forzamos a que siempre exista un camino, ya que de lo contrario muchas frases entrecomilladas serían identificadas como directorios relativos sin ser esto cierto. En los directorios no tomamos esta precaución, ya que serán mínimas las sentencias que no sean enlaces.

### 3.2. Reentrancia

Desde el principio, el código generado por `lex` nos decepcionaba en cuanto a que no satisfacía las reglas de *buena programación*, especialmente, utilizaba y obliga a utilizar variables globales, convirtiéndolo en código no reentrante.

Esto tenía sentido en los años 70 cuando fue desarrollado el `lex` original, dónde el concepto de proceso estaba naciendo y el de hebra ni existía. Sin embargo, en la actualidad, dónde todos los procesadores son multinúcleo, es imperativo realizar código multihebra para aprovechar al máximo la CPU y es irrisorio plantearse construir un generador de código como `lex` que genere código no reentrante. Es por esto que las versiones modernas de `lex`, como el universalmente extendido GNU `flex` que utilizamos, permiten generar analizadores léxicos reentrantes.

En nuestro programa en un principio no parece necesario que el lexer sea reentrante. Sin embargo, multihebrar el programa sería muy interesante ya que descargar una sola web a la vez es una pérdida de tiempo y ancho de banda innecesaria. Nuestro tiempo es muy limitado y eso se sale totalmente de los propósitos de la asignatura y es por eso que no hemos implementado esta función, pero aferrándonos al principio del “*por-si-acaso*” y satisfaciendo así nuestra curiosidad, hemos decidido generar un analizador reentrante.

Esto no se encuentra en el guión de prácticas, así que pondremos aquí las diferencias respecto a un lexer no reentrante.

Para que el código generado por GNU `flex` sea reentrante sólo debemos añadir la siguiente línea en la sección de declaraciones:

```
%option reentrant
```

El lexer reentrante ahora depende de un objeto de tipo `yyscan_t`. En éste objeto podemos incluir un parámetro controlado por el usuario cuyo

tipo viene controlado por una macro -por defecto vale `void*`- que podemos redefinir para ahorrarnos andar haciendo castings a diestro y siniestro. Es por esto que en la sección de declaraciones de C podemos encontrar esto:

```
1 #define YY_EXTRA_TYPE mailcrawl_p
```

Notar que las expresiones regulares pueden seguir usando `yytext`, `yylen` y el resto de macros, funciones y variables globales de la misma forma que antes, ya que Flex define macros del preprocesador para que no haya que complicar nada en ese aspecto.

Lo único que hay que cambiar es la llamada al lexer. Ahora debemos construir el objeto de tipo `yyscan_t` y luego añadirle los parámetros - fichero de entrada, el parámetro opcional antes descrito, etc.- con funciones asociadas al objeto y finalmente llamar a `yylex()` pero pasándole el objeto, que despues tendremos que destruir. A continuación se muestra a modo ilustrativo el código llamador de nuestro programa, que tras lo expuesto debe ser lo suficientemente autoexplicativo.

```
1  yylex_init(&scanner);
2
3  yyset_in(fin, scanner);
4  yyset_extra(mc, scanner);
5
6  yylex(scanner);
7
8  yylex_destroy(scanner);
```

## 4. Conclusión

Podemos concluir, por un lado las expresiones regulares obtenidas no son totalmente perfectas. A lo largo de esta sección se han descrito algunos casos excepcionales que conseguirían “engañarlas” de distintas formas. Sin embargo, son válidas en la mayoría de los casos y, en nuestra opinión, estas ambigüedades son intrínsecas al problema y no son causa de que hayamos escatimado esfuerzos en limar todos los detalles.

Por ejemplo, no podemos determinar si una URL apunta al fichero real a descargar o a un directorio que lo contiene o a una redirección hasta que no analizamos la conexión HTTP con el servidor. Igualmente, para

analizar saber correctamente que cosas son enlaces relativos y qué posiblemente sería adecuado un analizador semántico -por ejemplo, generado por GNU Bison o cualquier sucedaneo de yacc- que pueda obtener más información del código HTML o de los scripts JavaScript.

En definitiva estamos contentos con el resultado de la práctica ya que no nos hemos limitado a enunciar una serie de expresiones regulares que busquen elementos genéricos, sino que realmente las hemos adaptado al contexto y los requerimientos de un problema real, dónde el analizador léxico es sólo parte de la solución.

## Apéndices

### A. Código primera propuesta

```
/*----- Seccion de Declaraciones -----*/
%{
#include <stdio.h>

FILE* entrada;
FILE* salida;
int nhiper, ncorreo;
void escribir_datos (int dato1, int dato2);
%}

file_char ([A-Za-z][0-9])

%%
/*----- Seccion de Reglas -----*/

\"(h|H)(t|T)(t|T)(p|P)\:[^\"]*\" {ECHO, fprintf (salida, \"\n\");nhiper++;}
{file_char}+@{file_char}+ {ECHO, fprintf (salida, \"\n\");ncorreo++;}
.|\\n {}

%%
/*----- Seccion de Procedimientos -----*/

int main (int argc, char *argv[])
{
if (argc ==3)
{
entrada = fopen (argv[1], \"r\");
```

```

        salida = fopen (argv[2], "w");
        if ( entrada == NULL)
        {
            printf ("No puedo abrir el fichero %s .\n", argv[1]);
            exit(-1);
        }
        if ( salida == NULL)
        {
            printf ("No puedo crear o sobrescribir el fichero %s .\n", argv[2]);
            exit(-1);
        }
    }
else
{
    printf ("\nNecesito los nombres de los ficheros de entrada y salida para seguir\n");
    exit(-1);
}

nhiper = ncorreo = 0;

yyin = entrada;
yylex ();

escribir_datos(nhiper, ncorreo);

fclose(salida);
fclose(entrada);

return 0;
}

void escribir_datos (int dato1, int dato2)
{
    printf ("Hiper= %d\tCorreos=%d\n",dato1, dato2);
}

```