

Resumen de ficheros HTML

Iván López de Munain Quintana

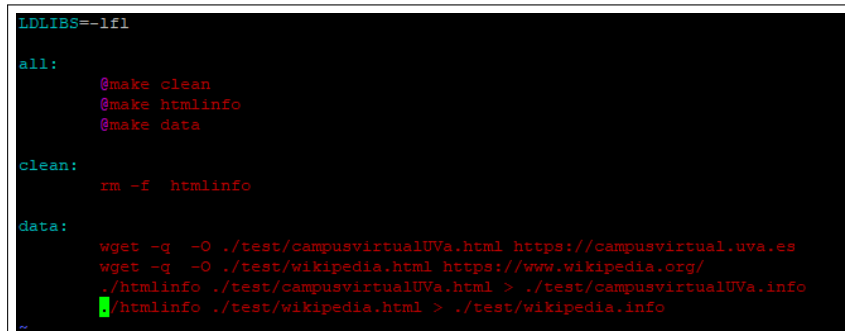
28 de marzo de 2019

I. INTRODUCCIÓN.

En esta primera entrega se va a poner en práctica los conocimientos adquiridos de *Lex* para llevar a cabo un análisis léxico de ficheros HTML. Para realizar estos objetivos se va a descargar los ficheros HTML de las siguientes páginas:

- <https://campusvirtual.uva.es>
- <https://www.wikipedia.org/>

Para conseguir esto se ha implementado el correspondiente *Makefile* con el fin de automatizar la generación del ejecutable, limpieza de carpetas y ficheros temporales y la descarga de los ya comentados HTML. Dicho *Makefile* tiene la siguiente forma:



```
LDLIBS=-lfl

all:
    @make clean
    @make htmlinfo
    @make data

clean:
    rm -f htmlinfo

data:
    wget -q -O ./test/campusvirtualUVa.html https://campusvirtual.uva.es
    wget -q -O ./test/wikipedia.html https://www.wikipedia.org/
    ./htmlinfo ./test/campusvirtualUVa.html > ./test/campusvirtualUVa.info
    ./htmlinfo ./test/wikipedia.html > ./test/wikipedia.info
```

Figura 1: Composición del *Makefile*.

Podemos observar en la Figura 1 que lo primero que se lleva a cabo es el borrado del fichero que se crea al compilar y ejecutar *htmlinfo.l* (*htmlinfo*). Posteriormente se compila dicho fichero *htmlinfo*, se descargan los fichero HTML correspondiente en donde *-q* sirve para que no nos imprima notificaciones innecesarias y *-O* para almacenar el HTML como fichero. Por último se ejecuta el archivo en cuestión pasándole como entrada el HTML correspondiente y almacenando el resultado en *./test/campusvirtualUVa.info* y *./test/wikipedia.info*.

II. ESPECIFICACIONES.

Para lograr evitar los comentarios en HTML he reutilizado parte de una función realizada en clase llamada *IgnoraComentarioMultiLinea*. Antes de nada hay que tener en cuenta que los comentarios en HTML son de la siguiente forma:

```
<!-- Esto es un comentario -->
```

Es por esto que en la parte de ordenes he incluido la siguiente expresión regular para la cual siempre que se satisfaga el patrón se ejecutará *IgnoraComentarioMultiLinea*:

```
"<!--" IgnoraComentarioMultiLinea(&nlin,&ncol);
```

El procedimiento he tenido que implementarlo teniendo en cuenta que la terminación del comentario consta de tres caracteres (`-->`) por lo que he aumentado la profundidad un nivel implementado un *switch* más.

```
while(!done) {
    switch(t=input())
    {
        /* EOF: Bad comment */
        case EOF:
            fprintf(stderr, "___ERROR: unclosed comment, expect */\n");
            done = 1 ;
            break;

        case '-':
            switch(t=input())
            {
                /* EOF: Bad comment */
                case EOF:
                    fprintf(stderr, "___ERROR: unclosed comment, expect */\n");
                    done = 1 ;
                    break;

                case '-':
                    if((c = input()) == '>') { *nc = *nc + 2 ; done = 1; }
                    else { *nc = *nc + 1 ; unput(c); }
            }
            if (!done) { fputc(t, stderr); fputc(c, stderr) ; }
            break;

        default:
            if (t=='\n') { *nl = *nl + 1 ; *nc = 1 ; }
            else { *nc = *nc + 1 ; }
            fputc(t, stderr);
            break;
    }
}

#ifdef VERBOSE
fprintf(stderr, ")-(%d, %d)\n", *nl, *nc);
#endif
```

Figura 2: Código de *IgnoraComentarioMultiLinea(nlin,ncol)*.

Otra especificación es que tenemos que tener en cuenta que las etiquetas pueden estar escritas tanto en mayúsculas como en minúsculas como mezcladas (se ve cómo conseguir esto a continuación). Además, para abarcar este amplio campo de posibilidades en casos concretos como "Title" he usado la opción (*?i:*) gracias a la cual la regla que definas es insensible a mayúsculas y minúsculas (el resto de definiciones se pueden ver en el script). A continuación podemos ver varios ejemplos:

SPACE	[\t]+
IDENT	[a-zA-Z]+
ETIQUETA	\<{IDENT}
TITLE	\<(?i: title)\>
CIERRETITLE	<\/(?i: title)\>

Para obtener el nombre del fichero HTML sobre el que se está aplicando el analizador léxico basta con que en el main se imprima los argumentos correspondientes que se pasan al procedimiento. Para la obtención del título del documento ya se ha dado parte de la solución en párrafos anteriores, solo falta construir una expresión regular como la siguiente:

```
{SPACE}*{IDENT}({SPACE}{IDENT})*/{CIERRETITLE} printf("TITLE: %s\n", yytext);
```

en donde se hace uso del contexto por la derecha para que solo imprima lo que precede a un cierre de título (*</title>*). La expresión regular representa cero o más espacios seguido de una palabra en mayúsculas, minúsculas o mezcladas y por último seguido de cero o más espacios y palabras (en caso de ser un título que consta de varias palabras).

A continuación se va a mostrar cómo se ha conseguido obtener un resumen estadístico sobre la frecuencia de las distintas etiquetas en el documento estudiado. La expresión regular que he usado es *{ETIQUETA}[^>]* y cada vez que se satisfaga dicho patrón se ejecuta el procedimiento *procesamientoEtiquetas()*:

```
void procesamientoEtiquetas() {
    int tag=0;
    int i;
    char *etiq;
    etiq=(char*)malloc(1+strlen(yytext)*sizeof(char *));
    memcpy(etiq,yytext+1,sizeof(yytext));
    for(i=0;i<=contador;i++){
        if(strcmp(etiq,etiquetas[i])==0){
            numEtiquetas[i]++;
            tag=1;
        }
    }
    if(tag==0){
        etiquetas[contador]=etiq;
        contador++;
    }
}
```

Figura 3: Código de *procesamientoEtiquetas()*.

En el código de la Figura 3 lo único que se hace es almacenar las etiquetas en el array global *etiquetas* en caso de que dichos tags no se hayan almacenado aun, en caso contrario se incrementará en una unidad el número de veces que se ha observado. Además también se nos pide que las

ordenemos alfabéticamente y cambiemos a letras minúsculas (función *tolower()*), todo esto se ha conseguido con el siguiente código en C:

```
//ordenamiento alfabetico de las etiquetas
char *tmp;
int aux;
for(int i=0;i<contador; i++) {
    for(int j=0; j<contador; j++) {
        if(strcmp(etiquetas[i], etiquetas[j]) < 0) {
            tmp = etiquetas[i];
            aux = numEtiquetas[i];
            etiquetas[i] = etiquetas[j];
            numEtiquetas[i] = numEtiquetas[j];
            etiquetas[j] = tmp;
            numEtiquetas[j] = aux;
        }
    }
}

//conversion a minusculas
for(int i=0;i<contador;i++){
    int w=0;
    while(etiquetas[i][w] ) {
        etiquetas[i][w]=tolower(etiquetas[i][w]);
        w++;
    }
}

//impresion del resumen estadistico de etiquetas
printf("TAGSTATS:\n");
for(int i=0;i<contador;i++){
    //NOS ASEGURAMOS QUE NO HAYA ETIQUETAS !DOCTYPE
    if(strcmp(etiquetas[i], "!doctype") != 0) {
        printf("\t%s\t%d\n", etiquetas[i], (numEtiquetas[i]+1));
    }
}
```

Figura 4: Normalización e impresión del resumen estadístico de etiquetas.

En la figura 4 podemos ver además la consecución de la siguiente especificación de no incluir la etiqueta `<!doctype...>` en el resumen. Una vez depuradas todas las etiquetas conseguir esto es muy sencillo pues basta con imprimir las etiquetas que cumplan la condición de ser distintas a `"!doctype"`. Ciertamente se podría haberse usado *Lex* y expresiones regulares para conseguir esto.

A continuación se pide obtener las distintas *URLs* que aparezcan en las etiquetas `<a>`, `<script>` y `<link>`. A la hora de resolver esta especificación hay que tener en cuenta que cada etiqueta puede tener uno o más atributos distintos en los que puede haber una *URL*:

- `<a>:href`
- `<link>:href`
- `<script>:src`

Este apartado está claramente pensado para solucionarlo usando condiciones *Start* y usar el contexto izquierdo, pero en mi caso he decidido extraer por completo las etiquetas con sus atributos y procesar posteriormente la cadena para obtener solo el link deseado. Bien es cierto que desde una perspectiva crítica el haber hecho el procesado en C, en vez de haber usado condiciones iniciales, me ha dado una solución más farragosa y menos eficiente. Mediante el uso del contexto izquierdo se podría haber implementado lo siguiente:

```
% Start A

%%

{LINKS}.*{HREF}.*+    {BEGIN A;}

<A>                    {procesamientoLINKS ();
                        BEGIN 0;}
```

Cabe destacar que mi implementación no es del todo correcta porque como en el enunciado dice que se debe mostrar la *URL* sin comillas, había dado por hecho (craso error) que todas las *URLs* se encuentran entre comillas en un fichero HTML. Debido a que me he dado cuenta el mismo día de la entrega que esto no es así siempre, no tengo tiempo suficiente como para cambiar tanto la documentación como la implementación porque, como se va a mostrar a continuación, para la obtención de las *URLs* "limpias" me he basado en encontrar la posición en la que se encuentran las comillas tanto de inicio como de fin para extraer la cadena del medio. A continuación muestro cómo he desarrollado mi solución:

```
LINKS    \<((?i:a)|(?i:link))
HREF      (?i:href)

LINKS2    \<(?i:script)
SRC        (?i:src)

%%

{LINKS}.*{HREF}.*+>    {procesamientoLINKS ();
                        REJECT;}

{LINKS2}.*{SRC}.*+>    {procesamientoLINKS ();
                        REJECT;}

%%

/*
 * Almacenamiento de las urls de a, script y link (sin depurar) en arrayEnlaces
 */

void procesamientoLINKS(){
    char *links;
    int indice;
    int tag=1;
    links=(char*)malloc(1+strlen(yytext));
    strcpy(links,yytext);
    arrayEnlaces[contadorEnlaces]=links;
    contadorEnlaces++;
}
```

He vuelto a usar el contexto derecho para eliminar ">" (aunque realmente no sería necesario). En las expresiones regulares he tenido en cuenta que las etiquetas tienen atributos con nombres distintos (href, src). La orden *REJECT* sirve para volver y ejecutar distintas alternativas de

expresiones regulares que también hayan sido definidas y coincidan con estas (es para que las estadísticas de los tags se obtuviesen de forma correcta).

Voy a explicar una de las definiciones de las expresiones regulares (LINKS) que se traduce como `<A|<a|<Link|<link|<llink|...` siendo las opciones restantes las combinaciones posibles de mayúsculas y minúsculas sobre la palabra "link". El resto de definiciones son análogas a esta.

Una vez que he obtenido todos los enlaces (sin depurar) he implementado unas líneas de código en C para obtener la URL deseada (Figura 5 y 6).

```

/*procesamiento de URL teniendo en cuenta que
-la etiqueta a y link poseen el atributo href
-la etiqueta script posee el atributo src

hago un procesamiento de cadenas teniendo en cuenta el nombre del atributo en cuestion
y que la url deseada se encuentre entre comillas

Uso de la funcion memcopy para eliminar los caracteres que se encuentren al inicio de la palabra (ej: href=, src=...)
procesamientoPalabras(char*, char) nos devuelve el indice de la primera ocurrencia de un caracter en un string, cero en caso contrario
-este procedimiento nos va a ayudar para localizar las comillas de inicio y final de la url y quedarnos con lo del medio
*/

int posAux;
for(int i=0;i<contadorEnlaces;i++){
    if(procesamientoPalabras(arrayEnlaces[i],"href")!=0){
        posicion = procesamientoPalabras(arrayEnlaces[i],"href");
        memcopy(arrayEnlaces[i],arrayEnlaces[i]-posicion,strlen(arrayEnlaces[i])-posicion+1);
        posicion2 = procesamientoPalabras(arrayEnlaces[i],"");
        memcopy(arrayEnlaces[i],arrayEnlaces[i]+posicion2+1,strlen(arrayEnlaces[i])-posicion2+1);
        posAux = procesamientoPalabras(arrayEnlaces[i]," ");
        posicion3 = procesamientoPalabras(arrayEnlaces[i],"");
    }else if(procesamientoPalabras(arrayEnlaces[i],"src")!=0){
        posicion = procesamientoPalabras(arrayEnlaces[i],"src");
        memcopy(arrayEnlaces[i],arrayEnlaces[i]-posicion,strlen(arrayEnlaces[i])-posicion+1);
        posicion2 = procesamientoPalabras(arrayEnlaces[i],"");
        memcopy(arrayEnlaces[i],arrayEnlaces[i]+posicion2+1,strlen(arrayEnlaces[i])-posicion2+1);
        posAux = procesamientoPalabras(arrayEnlaces[i]," ");
        posicion3 = procesamientoPalabras(arrayEnlaces[i],"");
    }
    final[i]=(char *)malloc(posicion3+1);

    //compruebo que las primeras comillas aparezcan antes que el primer espacio
    if(posicion3<posAux){
        for(int j=0;j<posicion3;j++){
            final[i][j]=arrayEnlaces[i][j];
        }
    }
}

```

Figura 5: *Procesamiento enlaces(1).*

En la Figura 5, para todas las cadenas sin depurar, obtengo la posición donde se encuentra "href" o "src" y borro todos los caracteres anteriores (para esto uso la función *memcopy*). Posteriormente obtengo el índice donde se encuentran las primeras comillas de la cadena formada en la orden anterior (estas comillas corresponden con comienzo del enlace) y otra vez mediante *memcopy* borro los caracteres anteriores. Por último, de la misma forma, obtengo la posición de las primeras comillas de la cadena creada en el paso previo (corresponden con el final del enlace) y mediante un bucle leo todos los caracteres hasta dicha posición y los almaceno en un array de Strings.

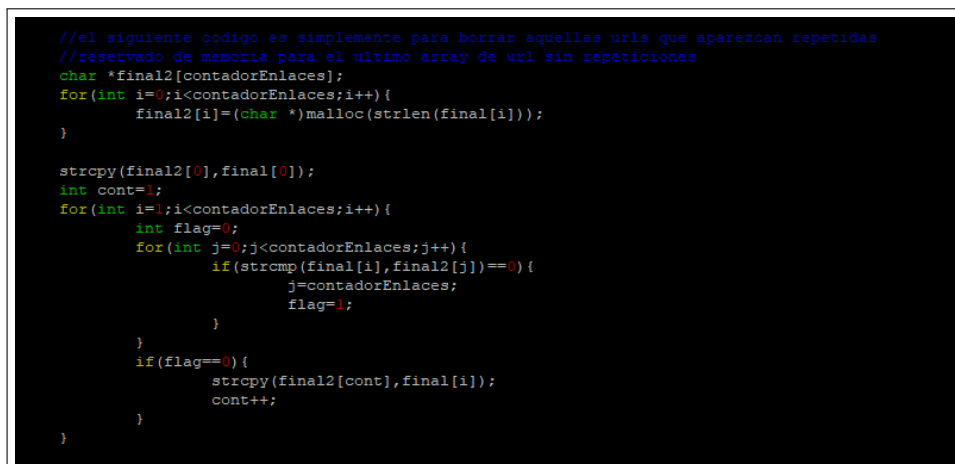
Al almacenar los Strings (final del código de la Figura 5) se puede ver que hago una comprobación de si las últimas comillas se encuentran antes que el primer espacio, esto tiene una explicación. Esto es debido a que hay enlaces que no se encuentran entre comillas (cosa que no había previsto) por lo que al ejecutar este código puedo obtener cadenas indeseadas. De esta forma, solo obtengo los links que aparecen entre comillas en el documento HTML pero no obtengo fragmentos indeseados.

Cabe destacar que para estudiar la contención de un substring en un String y obtener la posición se usa la función *procesamientoPalabras(char*,char)*:

```
/*
 * Funcion que devuelve la posicion donde se encuentra una subcadena dentro
 * de una cadena
 */
int procesamientoPalabras(char palabra[], char subcadena[]) {

    int posicion;
    if (strstr(palabra, subcadena) != NULL) {
        posicion = strstr(palabra, subcadena) - palabra;
        return posicion;
    } else {
        return 0;
    }
}
```

En la Figura 6 lo único que hago es eliminar las *URLs* repetidas:



```
//el siguiente codigo es simplemente para borrar aquellas url que aparezcan repetidas
//reservado de memoria para el ultimo array de url sin repeticiones
char *final2[contadorEnlaces];
for(int i=0; i<contadorEnlaces; i++){
    final2[i] = (char *) malloc(strlen(final[i]));
}

strcpy(final2[0], final[0]);
int cont=1;
for(int i=1; i<contadorEnlaces; i++){
    int flag=0;
    for(int j=0; j<contadorEnlaces; j++){
        if(strcmp(final[i], final2[j]) == 0){
            j=contadorEnlaces;
            flag=1;
        }
    }
    if(flag==0){
        strcpy(final2[cont], final[i]);
        cont++;
    }
}
```

Figura 6: *Procesamiento enlaces(2).*

NOTA: En el resultado no se encuentran aquellas *URLs* que no se encuentran entre comillas, debida a la falta de tiempo y todos los cambios que implicaría en mi diseño no he podido cambiarlo.

La siguiente especificación consiste también en obtener enlaces pero esta vez de la etiqueta *img*. En esta ocasión me ocurre lo mismo que lo comentado anteriormente en la **NOTA**. Pese a eso, el funcionamiento es muy similar al caso previo pero esta vez posee más atributos que puedan contener links:

- *: src*
- *: usemap*
- *: longdesc*

Es por esto que se han declarado las siguientes expresiones regulares predefinidas, las órdenes correspondientes y el procedimiento que se ejecuta:

```
IMG      \<( ? i : img )
LONGDESC ( ? i : longdesc )
USEMAP   ( ? i : usemap )

%%

{IMG}.*{SRC}.+ />      {procesamientoIMAGES ();
                        REJECT;}

{IMG}.*{LONGDESC}.+ />  {procesamientoIMAGES ();
                        REJECT;}

{IMG}.*{USEMAP}.+ />    {procesamientoIMAGES ();
                        REJECT;}

%%

/*
 * Almacenamiento de las url de imagenes entera(sin depurar) en arrayImágenes
 *
 */
void procesamientoIMAGES(){
    char *img;
    img=(char*) malloc(1+strlen(yytext));
    strcpy(img,yytext);
    arrayImágenes[contadorImágenes]=img;
    contadorImágenes++;
}
```

No voy a detenerme a explicar el código anterior porque es totalmente análogo a la especificación de LINKS. Tanto en la Figura 7 como en la Figura 8 se puede ver la depuración de la cadena hasta la obtención del enlace "limpio" y eliminación de duplicados, el procedimiento es exactamente el mismo que el explicado anteriormente, salvando pequeñas diferencias como que el nombre de los atributos es distinto.


```

printf("IMAGES: \n");
char *finalImagen[contadorImagenes];
for(int i=0;i<contadorImagenes;i++){
    if(procesamientoPalabras(arrayImagenes[i],"src")!=0){
        posicion = procesamientoPalabras(arrayImagenes[i],"src");
        memcpy(arrayImagenes[i],arrayImagenes[i]+posicion,strlen(arrayImagenes[i])-posicion+1);
        posicion2 = procesamientoPalabras(arrayImagenes[i],"");
        memcpy(arrayImagenes[i],arrayImagenes[i]+posicion2+1,strlen(arrayImagenes[i])-posicion2+1);
        posAux = procesamientoPalabras(arrayImagenes[i],"");
        posicion3 = procesamientoPalabras(arrayImagenes[i],"");
    }else if(procesamientoPalabras(arrayImagenes[i],"longdesc")!=0){
        posicion = procesamientoPalabras(arrayImagenes[i],"longdesc");
        memcpy(arrayImagenes[i],arrayImagenes[i]+posicion,strlen(arrayImagenes[i])-posicion+1);
        posicion2 = procesamientoPalabras(arrayImagenes[i],"");
        memcpy(arrayImagenes[i],arrayImagenes[i]+posicion2+1,strlen(arrayImagenes[i])-posicion2+1);
        posAux = procesamientoPalabras(arrayImagenes[i],"");
        posicion3 = procesamientoPalabras(arrayImagenes[i],"");
    }else if(procesamientoPalabras(arrayImagenes[i],"usemap")!=0){
        posicion = procesamientoPalabras(arrayImagenes[i],"usemap");
        memcpy(arrayImagenes[i],arrayImagenes[i]+posicion,strlen(arrayImagenes[i])-posicion+1);
        posicion2 = procesamientoPalabras(arrayImagenes[i],"");
        memcpy(arrayImagenes[i],arrayImagenes[i]+posicion2+1,strlen(arrayImagenes[i])-posicion2+1);
        posAux = procesamientoPalabras(arrayImagenes[i],"");
        posicion3 = procesamientoPalabras(arrayImagenes[i],"");
    }
    finalImagen[i]=(char *)malloc(posicion3+1);

    //compruebo que las primeras comillas aparezcan antes que el primer espacio
    if(posicion3<posAux){
        for(int j=0;j<(posicion3);j++){
            finalImagen[i][j]=arrayImagenes[i][j];
        }
    }
}

```

Figura 7: Procesamiento enlaces de IMG(1).

```

//el siguiente codigo es simplemente para borrar aquellas urls que aparezcan repetidas
//reservado de memoria para el ultimo array de url sin repeticiones
char *finalImagen2[contadorImagenes];
for(int i=0;i<contadorImagenes;i++){
    finalImagen2[i]=(char *)malloc(strlen(finalImagen[i]));
}

strcpy(finalImagen2[0],finalImagen[0]);
cont=1;
for(int i=1;i<contadorImagenes;i++){
    int flag=0;
    for(int j=0;j<contadorImagenes;j++){
        if(strcmp(finalImagen[i],finalImagen2[j])==0){
            j=contadorImagenes;
            flag=1;
        }
    }
    if(flag==0){
        strcpy(finalImagen2[cont],finalImagen[i]);
        cont++;
    }
}

```

Figura 8: Procesamiento enlaces de IMG(2).

Para completar el analizador léxico se requiere que si existiese, se procesara la información de la etiqueta `<base>` para obtener la *URL* que debería añadirse a cualquier dirección de enlace (tanto de las imágenes como de enlaces normales). De esta especificación he supuesto entonces que en cada fichero HTML como mucho va a haber un único enlace en `<base>` que será la dirección absoluta. Por esto, cuando encuentre una etiqueta *base* y obtenga dicha dirección de enlace absoluta actualizaré un flag global llamado *Absoluto* y se imprimirá todos los enlaces (igual que en las especificaciones anteriores) pero precedidos de dicho enlace absoluto. Para lograr esto he añadido lo siguiente en las secciones de definiciones,órdenes y rutinas:

```

BASE      \<( ? i : base )

%%

{BASE}.*{HREF}.*</>      {procesamientoBASE ();
                           REJECT;}

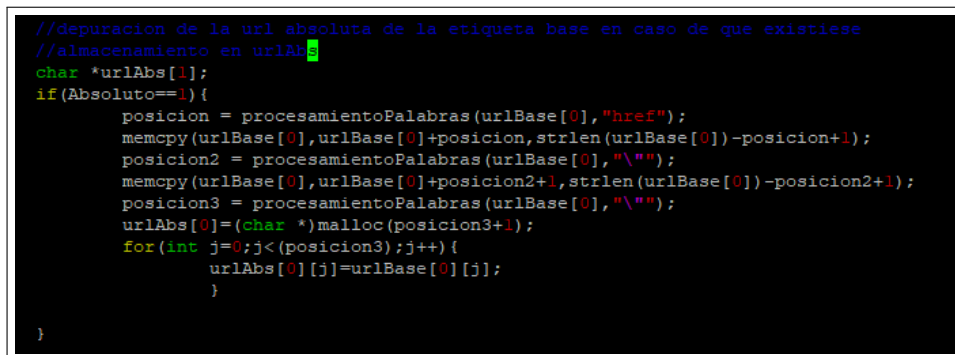
%%

/*
* Procedimiento que obtiene la url(sin depurar) del atributo BASE y actualiza la vari
*/
void procesamientoBASE(){

    char *base;
    Absoluto=1;
    base=(char*) malloc(1+strlen(yytext));
    strcpy(base,yytext);
    urlBase[0]=base;
}

```

La explicación del código anterior es análoga a las dadas en apartados anteriores. Aquí cabe resaltar que solo se espera encontrar un enlace en todo el documento HTML y que la variable global *Absoluto* se pone a 1 para que a la hora de imprimir todos los enlaces obtenidos previamente se concatenen a dicha dirección de enlace absoluta. Para lograr esto basta con depurar el enlace igual que en los casos previos e incluir un *if* en función de *Absoluto* (Figura 9 y Figura 10). Finalmente solo hay que imprimir los resultados obtenidos (Figura 11).

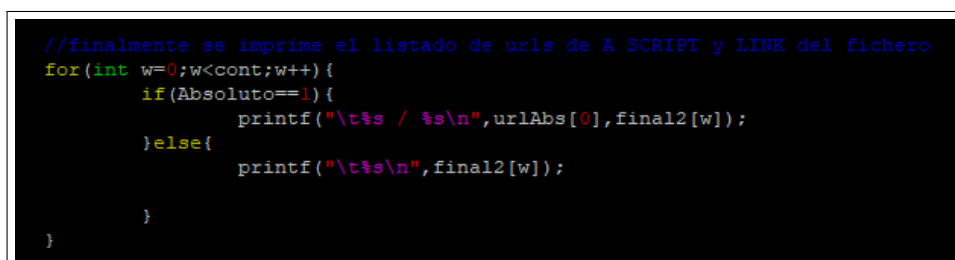


```

//depuracion de la url absoluta de la etiqueta base en caso de que existiese
//almacenamiento en urlAbs
char *urlAbs[1];
if(Absoluto==1){
    posicion = procesamientoPalabras(urlBase[0],"href");
    memcpy(urlBase[0],urlBase[0]+posicion,strlen(urlBase[0])-posicion+1);
    posicion2 = procesamientoPalabras(urlBase[0],"\"");
    memcpy(urlBase[0],urlBase[0]+posicion2+1,strlen(urlBase[0])-posicion2+1);
    posicion3 = procesamientoPalabras(urlBase[0],"\"");
    urlAbs[0]=(char *)malloc(posicion3+1);
    for(int j=0;j<(posicion3);j++){
        urlAbs[0][j]=urlBase[0][j];
    }
}

```

Figura 9: Procesamiento enlace de BASE.



```

//finalmente se imprime el listado de urls de A SCRIPT y LINK del fichero
for(int w=0;w<cont;w++){
    if(Absoluto==1){
        printf("\t%s / %s\n",urlAbs[0],final2[w]);
    }else{
        printf("\t%s\n",final2[w]);
    }
}

```

Figura 10: Impresión de los enlaces de A, SCRIPT y LINK.

```
//finalmente se imprime el listado de urls de IMG del fichero
for(int w=0;w<cont;w++){
    if(Absoluto==1){
        printf("\t%s / %s\n",urlAbs[0],finalImagen2[w]);
    }else{
        printf("\t%s\n",finalImagen2[w]);
    }
}
```

Figura 11: *Impresión de los enlaces de IMG.*