CHAPTER 16

Keyword Searching

OVERVIEW

When a Web site starts to grow beyond a couple of pages, it can be time-consuming for visitors to find what they are looking for. As more Web pages are connected to one another via hypertext links, the resulting spaghetti-like relationship only exacerbates the complexity.

Although some Web sites allow a browser to find pages by keyword, these large search databases typically point you only to a particular page on a particular site rather than all the pages on the site that match the keyword. Furthermore, the big Internet search engines do not necessarily have up-to-date references to all the pages at your site.

Having a keyword search engine installed on your Web site solves these problems. First, it provides a quick way for people to find the pages they are looking for. Additionally, because the script is searching the actual pages on the Web site, the results of the search are always up-to-date.

The main interface to the keyword search engine allows you to search on any number of keywords. By default, the keywords are searched as parts of larger words, or you can use an exact match search for whole words. Figure 16.1 illustrates the keyword search engine.

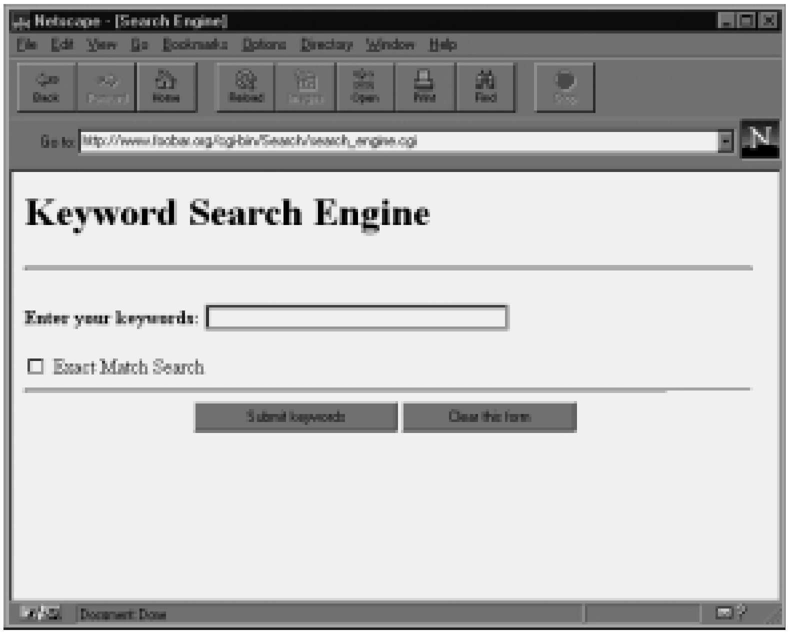
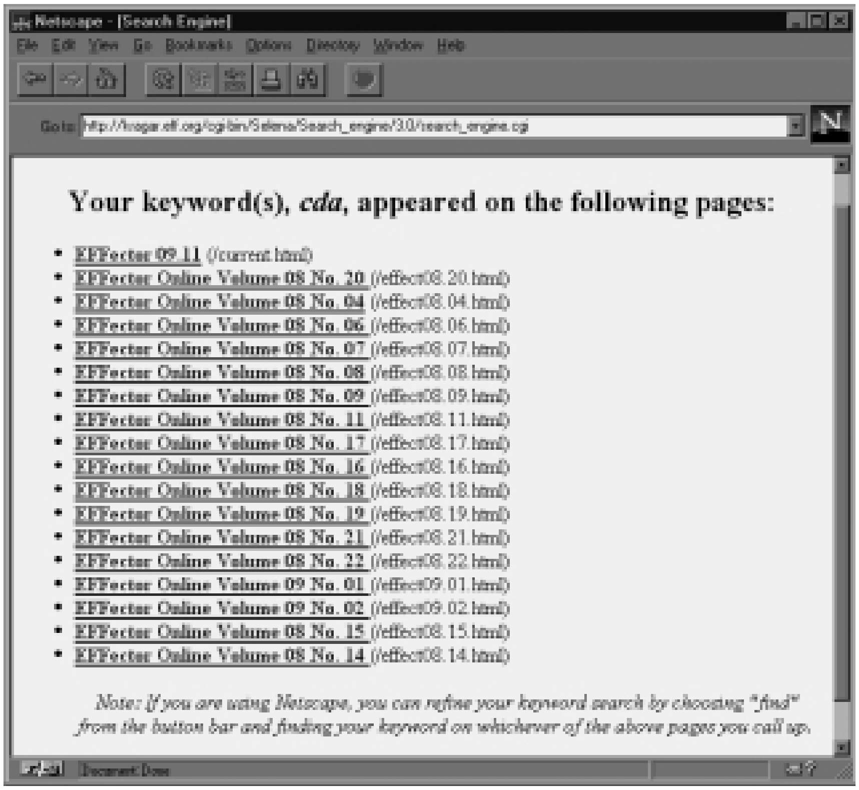

Netscape - [Search Engine]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
Go to: http://www.foobar.org/cgi-bin/Search/search_engine.cgi
Keyword Search Engine
Enter your keywords:
Exact Match Search
Submit keywords
Clear this form
Document: Done

Figure 16.1 Keyword search query screen.

By default, the keyword search engine searches all files with **.HTM** or **.HTML** extensions in all directories below the document root. The search engine can also be configured to exclude certain classes of directories and files in case you want to protect certain areas of the Web server.

The results of a search include the title of a document as well as the HTML reference to that document. Figure 16.2 demonstrates the results of a search on the abbreviation "cda" as part of the Electronic Frontier Foundation (EFF) archive.


Netscape - [Search Engine]
File Edit View Go Bookmarks Options Directory Window Help
Go to: http://kragar.eff.org/cgi-bin/Selena/Search_engine/3.0/search_engine.cgi
Document: Done

Your keyword(s), *cda*, appeared on the following pages:

- EFFector 09.11 (/current.html)
- EFFector Online Volume 08 No. 20 (/effect08.20.html)
- EFFector Online Volume 08 No. 04 (/effect08.04.html)
- EFFector Online Volume 08 No. 06 (/effect08.06.html)
- EFFector Online Volume 08 No. 07 (/effect08.07.html)
- EFFector Online Volume 08 No. 08 (/effect08.08.html)
- EFFector Online Volume 08 No. 09 (/effect08.09.html)
- EFFector Online Volume 08 No. 11 (/effect08.11.html)
- EFFector Online Volume 08 No. 17 (/effect08.17.html)
- EFFector Online Volume 08 No. 16 (/effect08.16.html)
- EFFector Online Volume 08 No. 18 (/effect08.18.html)
- EFFector Online Volume 08 No. 19 (/effect08.19.html)
- EFFector Online Volume 08 No. 21 (/effect08.21.html)
- EFFector Online Volume 08 No. 22 (/effect08.22.html)
- EFFector Online Volume 09 No. 01 (/effect09.01.html)
- EFFector Online Volume 09 No. 02 (/effect09.02.html)
- EFFector Online Volume 08 No. 15 (/effect08.15.html)
- EFFector Online Volume 08 No. 14 (/effect08.14.html)

Note: If you are using Netscape, you can refine your keyword search by choosing "find" from the button bar and finding your keyword on whichever of the above pages you call up.

Figure 16.2 *The results of a search of "cda" in the EFF archive.*

INSTALLATION AND USAGE

The keyword search files should be installed in a directory called **Search**. Figure 16.3 shows an example of how the files should be structured and the permissions set so that they can be accessed by the Web server.

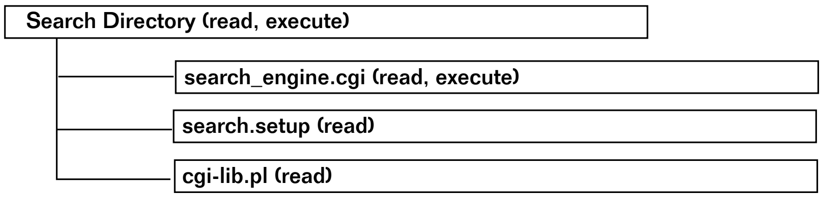

Search Directory (read, execute)
search_engine.cgi (read, execute)
search.setup (read)
cgi-lib.pl (read)

Figure 16.3 *Directory structure of the search engine CGI scripts.*

The root **Search** directory must have permissions that allow the Web server to read and execute files.

search_engine.cgi is the script that performs the keyword search. In addition, when it is called with no parameters, it prints an HTML form asking the user to enter a query. This file must be readable and executable.

search.setup is the file that contains all the setup options for the keyword search engine. This file must be readable.

cgi-lib.pl is an external library required by the keyword search engine. This file must be readable.

Server-Specific Setup and Options

The **search.setup** file contains the search engine configuration variables. The following is a list of these setup items.

`$root_web_path` is the directory path on the Web server at which you want to start your search. For example, `/usr/local/etc/httpd/htdocs` would make the search start in the **htdocs** directory under the **/usr/local/etc/httpd** directory. Using this configuration, the search would also continue searching all the files in the subdirectories underneath **htdocs**.

`$server_url` is the HTTP reference to the Web server that is serving the documents. When the searched files are found under the `$root_web_path`, they are appended with this server URL at the front of the path to the found file in order to provide a valid hypertext reference to search on. For example, if the `$server_url` were http://www.foobar.com/ and the file were found in **home/birzniek/index.html**, the full URL displayed to the user would become

```
http://www.foobar.com/home/birzniek/index.html
```

`$search_results_title` is the HTML title of the page that gets returned as the result of a successful keyword search.

`$search_script` is the name of the search engine CGI script program. This information is needed so that the HTML keyword search form printed by the script will post the keyword search information to the right script for processing. By default, the value of this parameter is `search_engine.cgi`.

`@unwanted_files` is a list of pattern-matched filenames that you wish to exclude from the search. These filenames can be pattern matches against the full local directory path, including the filename of each file being searched. Thus, you can pattern-match against directories as well as files you wish to be excluded from the search. The pattern matching is done according to normal UNIX pattern matching and regular expression rules. The example piece of code given next excludes a series of documents and a directory that you may typically not want to include. The directory (**Backup**) does not need a regular expression, because the pattern match implicitly matches zero or more characters at the beginning and end of each entry in the unwanted-files list. A regular expression is used for HTML files so that the beginning and end of the HTML filenames can be specified more explicitly while keeping the middle of the file open to any number of characters.

```
@unwanted_files = ("Error(.*)html",
             "error(.*)html",
             "Backup/",
             "feedback(.*)html");
```

NOTE

The notation used in the `@unwanted_files` array is the standard UNIX regular expression format. The parentheses allow the results of the regular expression to be seen by the Perl script. The period (.) tells Perl to match against any one character. The asterisk (*) tells Perl to match any number of the previous characters. Because the previous character

is a pattern-matched character that means any one character, the combination of ".*" matches zero or more of any characters. For example, if `error(.*)html` is an unwanted file, it will match any file beginning with "error" and ending with "html." This keyword would match files such as errorlog.html and error_message.html. The first two items on the sample list of unwanted files excludes any HTML file that has the word "Error" or "error" in the filename.

Typically, as a Web manager, you may have a series of common error files that tell the user that the document was not authorized or found. It does not make sense to return these documents as part of a search.

In addition, a feedback form is typically not part of the information that users are searching for. If they want to leave feedback, a hypertext link is usually available in the main home page area.

You could also use the `$unwanted_files` list to exclude directories that you may currently have protected against viewing by unauthorized people. If you regularly copy your HTML files into a **Backup** directory, you might want to exclude any HTML documents from being searched in that directory. The `Backup/` part of the example serves to do this.

WARNING

Even if your directories are restricted by a password or some other security mechanism, you should protect those directories from being searched via the search_engine.cgi script. Because the search engine is a CGI program, it has no idea what sort of Web server security may have been activated on the system. The search engine will search all directories indiscriminately as long as it can find new directories and files that do not match the unwanted files list.

Obviously, you do not want the results of a search to include specially protected documents. Although the user may not be able to click on the returned hypertext link, it can be a security breach for a user to know that an inaccessible document is on the server. It becomes a tempting goal for a hacker.

The following is an example usage of all the setup variables in the **search.setup** file:

```
$root_web_path = "/usr/local/etc/httpd/htdocs";
$server_url = "http://www.foobar.com";
$search_results_title = "The WWW Search Engine";
$search_script = "search_engine.cgi";
@unwanted_files = ("Error(.*)html",
                   "error(.*)html",
                   "Backup/",
                   "Images/",
                   "feedback(.*)html");
```

CUSTOMIZING THE HTML IN SEARCH.SETUP

The rest of the **search.setup** file contains subroutines that print HTML code that gives output depending on various actions of the keyword search engine. This HTML information is broken out from the main program to make it easier for you to customize the HTML to your site-specific graphics and HTML standards. If the keyword search engine gets updated, it is less likely that your GUI-specific customizations will have to be repeated, because most of the program logic resides in another file (**search_engine.cgi**).

The `PrintHeaderHTML` subroutine prints the HTML header for the keyword search when it is about to return results.

The code between the `__HEADERHTML__` tags in the Perl script prints to the Web browser. The `$search_results_title` and `$keywords` variables print their values and are generated by **search_engine.cgi**, which uses this subroutine.

```
sub PrintHeaderHTML
{

print <<__HEADERHTML__;
<HTML><HEAD><TITLE>$search_results_title</TITLE></HEAD>
<BODY><CENTER>
<HR><P>
</CENTER>
<CENTER><H2>Your keyword(s), <I>$keywords</I>,
appeared on the following pages:</H2></CENTER><UL>
__HEADERHTML__
} # End of PrintHeaderHTML
```

PrintFooterHTML prints the HTML codes that correspond to the footer of the keyword results set. A sample of this code appears next. The actual HTML code prints between the __FOOTERHTML__ tags.

```
sub PrintFooterHTML
{
print <<__FOOTERHTML__;
<P><CENTER><I>Note: If you are using Netscape, you can
refine your keyword search by choosing "find" from the
button bar and finding your keyword on whichever of
the above pages you call up.
</I>
<P>
<CENTER>
<HR>
</CENTER> </BODY> </HTML>
__FOOTERHTML__

} # End of PrintFooterHTML
```

PrintNoHitsBodyHTML contains the HTML codes that print as the body between the keyword search header and footer if no matches were found.

```
sub PrintNoHitsBodyHTML
{
print <<__NOHITS__;
<P>
<CENTER>
<H2>Sorry, No Pages Were Found With Your Keyword(s).</H2>
</CENTER>
<P>
__NOHITS__

} # End of PrintNoHitsBodyHTML
```

PrintBodyHTML prints the HTML code related to each keyword search hit. Every time a file is found as a match to the keyword search, this routine prints another set of HTML statements related to the URL location and description of the file. When printed as HTML code, the variables $server_url, $filename, and $title are replaced with their values. The first line of the following subroutine gets information that is passed to it for that specific match. The HTML code exists between the __BODYHTML__ tags.

```
sub PrintBodyHTML
{
local($filename, $title) = @_;

print <<__BODYHTML__;
<LI>
<B>
<A HREF=
"$server_url/$filename">$title</A>
</B>
 (/$filename)<BR>
__BODYHTML__

} # End of PrintBodyHTML
```

PrintNoKeywordsHTML prints the form, asking the user to enter a keyword search term and to specify whether or not the search should be an exact match. In the following sample code, the HTML is printed between the __NOKEYHTML__ tags. If you modify the HTML code to omit the exact_match variable, the script will still work but will match the keyword on the basis of a pattern match by default; if the keyword exists as part of a larger word, the program will relay that as a positive match. The exact algorithm is discussed in the "Design Discussion" section. Basically, exact match turns on searching based on "whole word" matches only.

```
sub PrintNoKeywordHTML
{
    print <<__NOKEYHTML__;
<HTML><HEAD><TITLE>Search Engine</TITLE></HEAD>
<BODY BGCOLOR="#ffffff">
<H1>Keyword Search Engine</H1>
<HR>
<P>
<FORM METHOD="POST" ACTION="search_engine.cgi">
<B>Enter your keywords:</B>
<INPUT TYPE="text" SIZE="30" NAME="keywords"
MAXLENGTH="80">
<P>
<INPUT TYPE=checkbox NAME="exact_match"> Exact Match Search
<HR>
<CENTER>
<INPUT TYPE="SUBMIT" VALUE="Submit keywords">
<INPUT TYPE="RESET" VALUE="Clear this form">
</CENTER>
</FORM>
</BODY></HTML>

__NOKEYHTML__

} # End of PrintNoKeywordHTML
```

NOTE

The `<FORM>` tag refers to the original search engine script and accepts input for keywords as well as for the exact match checking. If you modify this form, you should make sure that the `<INPUT>` tag names `keywords` and `exact_match` have the same spelling and case as shown in the example code.

Running the Script

To use the **search_engine.cgi** program, refer to it as a hypertext reference on your site. When the search engine does not detect that it is being sent any form variables, it automatically prints a form to allow the user to enter search terms. Here is a sample URL for this script if it is installed in a **Search** subdirectory under a **cgi-bin** directory:

```
http://www.foobar.com/cgi-bin/Search/search_engine.cgi
```

DESIGN DISCUSSION

The basic algorithm of the keyword search involves first checking to see whether the script has been given a series of keyword form variables. If the script has not received the variables, the script prints an HTML form asking for keywords to search on. If there is a keyword variable, the script searches for documents down through the directory structure and returns those documents that satisfy the keyword search. A basic chart of this logic is shown in Figure 16.4.

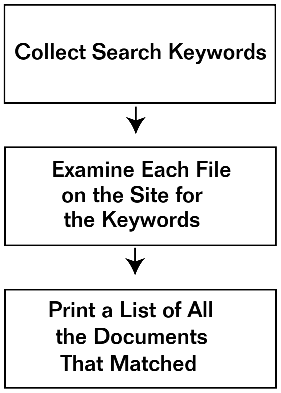

Collect Search Keywords
Examine Each File on the Site for the Keywords
Print a List of All the Documents That Matched

Figure 16.4 *Basic flowchart for the search engine.*

The first line of the following code sets up the location of the supporting files to the program. By default, `$lib` is set to the current subdirectory. Then the **cgi-lib.pl** library is loaded along with the **search.setup** variables.

```
$lib = ".";
require "$lib/cgi-lib.pl";
require "$lib/search.setup";
```

We first print the standard "Content-type: text/html\n\n" HTTP header using `PrintHeader` and then read the incoming form variables using `ReadParse`. This program uses the default associative array `%in` for storing the value of the form variables.

```
print &PrintHeader;
&ReadParse;
```

`$keywords` and `$exact_match` are set to be equal to the form variables for doing keyword search matching.

```
$keywords = $in{'keywords'};
$exact_match = $in{'exact_match'};
```

Once we have the `$keywords` variable, the script needs to `split` it apart so that each keyword can be searched separately within the files. We use the regular expression `/\s+/`. This expression matches on any white space (carriage returns, newlines, spaces, and so on) and `splits` the words accordingly.

```
@keyword_list = split(/\s+/,$keywords);
```

Printing the HTML Search Form

If there were no keywords to search, the HTML search form is output and the Perl program exits.

```
if ($keywords eq "") {
    &PrintNoKeywordHTML;
    exit;
} # End of if keywords
```

If there are keywords to search, the script prints the HTML header using the `PrintHeaderHTML` subroutine and moves to the heart of the keyword search engine code.

```
&PrintHeaderHTML;
```

Performing the Keyword Search

Before going further, let's step back and see how the script conducts a keyword search. The routine traverses the directory structure under `$root_web_path` and also parses the HTML files to see whether they contain the keywords we are searching for. If a match is found, the script determines the HTML titles and builds a list of successful "hits" for the client.

Traversing the Directory Structure of Your Web Site

As the script goes down a directory looking for entries, one of those entries may be a directory. In that case, the directory is opened, and it becomes the new directory to traverse. To keep traveling down the directory tree, the script must keep track of where it has been. An array called @dirs keeps track of this by containing the names of the open directories that the script has not yet finished searching. As a directory gets opened for searching, it is appended as a new element to the end of the @dirs array.

The following code sets up the initial variables for the algorithm. The @dirs array of directories is used as a placeholder so that we can go back up the directory tree when we run out of files to read in a subdirectory. $cur_dir is the current directory number, a reference to the element in @dirs for the directory we are currently reading. The directory handles in this program are referred to as the string "DIR" followed by the current directory number indicated by "DIR$cur_dir." $number_of_hits is the current number of successful hits found in the files. The number of hits is equal to the number of files that will be returned as matches for the keyword terms.

```
$number_of_hits = 0;
$cur_dir = 0;
@dirs = ($root_web_path);
```

We initialize the process by opening the directory handle using the reference "DIR$cur_dir" and the path that has been passed to the @dirs array. When the $end_of_all_files flag is set to 1, it stops the searching routine, because it means that we have finished searching every file in every directory that we can search.

```
opendir("DIR$cur_dir", $dirs[$cur_dir]);
$end_of_all_files = 0;
```

The following while loop does not exit until the script is finished searching all the files. Within this loop is another one that goes on forever unless the last command is encountered inside. It is inside this second while loop that the directory tree for HTML documents is traversed.

```
while (!($end_of_all_files)) {
       while (1) {
```

First, the script gets from the $filename variable a reference to the next valid directory or filename. Next, $fullpath is set to the current path plus filename. Then, for the entry that was received, the routine goes through multiple cases and does different things on the basis of those cases. The five basic cases are discussed next.

```
$filename = &GetNextEntry("DIR$cur_dir",
            $dirs[$cur_dir]);
$fullpath = "$dirs[$cur_dir]/$filename";
```

Case 1: No More Files in Current Directory

In case 1, the file is `NULL` but there are still entries in the `@dirs` variable, so the program goes back up the directory tree and continues searching in a previous directory where it left off. Specifically, the program closes the current directory, subtracts 1 from the `$cur_dir` variable, and then issues a `next` command to force another iteration through the `while(1)` loop.

```
     if (!($filename) && $cur_dir > 0) {
         closedir("DIR$cur_dir");
         $cur_dir--;
         next;
     }
```

Case 2: The End of the Search

In case 2, there are no more filenames to search on but the script has been through all the previous entries in the `@dirs` array. Thus, the search needs to end. We close the current directory handle, set the `$end_of_all_files` to 1, and issue the `last` command to break completely out of the `while(1)` loop.

```
     if (!($filename)) {
         closedir("DIR$cur_dir");
         $end_of_all_files = 1;
            last;
     }
```

Case 3: The File Is a Directory

Case 3 discovers that the filename is actually a directory, so the script descends into the directory if it is both readable and executable. The program checks to see whether the file is a directory using the `-d` flag. It checks for readability and execute rights by using the `-r` and `-x` flags. Finally, the program goes down the directory tree if the filename is a directory: it increments the current directory counter, `$cur_dir`, by 1, pushes a new path onto the `@dirs` array, and opens a new directory handle. Finally, the `next` command is used to force the script to go back to the top of the `while(1)` loop.

```
if (-d $fullpath) {
    if (-r $fullpath && -x $fullpath) {
     $cur_dir++;
     $dirs[$cur_dir] = $fullpath;
     opendir("DIR$cur_dir", $dirs[$cur_dir]);
     next;
    } else {
     next;
    }
} # End of case 3 (File is a directory)
```

Case 4: The File is "Unwanted"

In case 4, the script checks to see whether the file about to be searched is unwanted. The program starts by setting the `$unwanted_file` flag to zero. Then the script performs a pattern match against each unwanted file in the `@unwanted_files` array to see whether the filename and path are unwanted. If they are unwanted, the `$unwanted_file` flag is set to 1. After all the `@unwanted_files` have been checked, if the `$unwanted_file` flag is equal to 1, the `next` command is issued to reiterate through the `while(1)` loop.

```
$unwanted_file = 0;
foreach (@unwanted_files) {
    if ($fullpath =~ /$_/) {
     $unwanted_file = 1;
    }
} # End of foreach unwanted files

if ($unwanted_file) {
    next;
} # End of Case 4 Unwanted File
```

Case 5: The File Needs to Be Searched

In the last case, the script finds that the file is a file that we want to search for keywords. The `-r` flag is used to check whether the file is readable; if it is, the `last` command is issued to force a break out of the `while(1)` loop. Breaking out of this loop allows the script to move on and search through the file.

```
     if (-r $fullpath) {
         last;
     } # Make sure the file is readable

    } # End of While (1)
```

Searching the HTML File for Keywords

After the `while(1)` loop, we check again for the `$end_of_all_files` flag. If it is not equal to 1, the script can continue the file search.

```
if (!($end_of_all_files)) {
```

When we search a file, we initially set `@not_found_words` to the array of keywords we want to search. This corresponds to the idea that, initially, none of the words is found. As we later search the file and find keywords, they are deleted from the `@not_found_words` array. When this array has no elements left, we know that all the keywords were found in the file and that we have found a hit (a successful match).

```
     @not_found_words = @keyword_list;
```

In addition to searching for the keyword, we attempt to parse out the name of the HTML file. The `$are_we_in_head` flag is set to zero initially. If it is zero, we know that we are still in the header of the HTML file. Upon reaching a `</HEAD>` or `</TITLE>` flag, the script knows that it is finished reading the header. The header is read into the `$headline` variable.

```
     $are_we_in_head = 0;
     open(SEARCHFILE, $fullpath);
     $headline = "";
     while(<SEARCHFILE>) {
         $line = $_;
         $headline .= $line if ($are_we_in_head == 0);
         $are_we_in_head = 1
          if (($line =~ m!</head>!i) ||
                ($line =~ m!</title>!i));
```

The `&FindKeywords` subroutine performs the search of the keywords in each line as it is read from the file. When `&FindKeywords` finds a match, it deletes the keyword from the `@not_found_words` array.

```
         &FindKeywords($exact_match, $line, *not_found_words);
     } # End of SEARCHFILE
     close (SEARCHFILE);
```

Printing a Successful Keyword Match

If the `@not_found_words` array is less than 1, the script knows that all the keywords were found, so it prints the matched files. Part of the routine that prints the match parses the title of the document out of the HTML code stored in `$headline`.

```
     if (@not_found_words < 1) {
```

The first thing the routine does is to replace all newlines with spaces in `$headline`. Then it sets up a match against the regular expression `<title>(.*)</title>`. This expression matches for zero or more characters between the `<TITLE>` HTML tags. In Perl, the successful match will make the variable `$1` equal to the characters between the `<TITLE>` tags. The `i` at the end of the match expression indicates that the match is performed without regard to case. If the title turns out not to exist in this document, the `$title` variable is set to `No Title Given`.

```
$headline =~ s/\n/ /g;
$headline =~ m!<title>(.*)</title>!i;
$title = $1;

     if ($title eq "") {
         $title = "No Title Given";
     }
```

NOTE

We use a special form of the match operator. Most of the time we use forward slashes (/) to indicate the endpoints of a search. Here, we use the `m` (match) operator followed by a different character to use as the matching operator. In this case, we use the exclamation point (!) to delimit the search. We do this because we are including forward slashes inside the actual expression to search, and escaping them with the backslash (\) would look messy. The same technique can be used with the `s` (substitute) operator, which is used in the next code sample.

The program then strips out the `$root_web_path`, because it contains information we do not want to pass to the user about the internal directory structure of the Web server. Finally, the script prints the HTML code related to the hit and increments the hit counter.

```
$fullpath =~ s!$root_web_path/!!;
&PrintBodyHTML($fullpath, $title);
$number_of_hits++;
} # If there are no not_found_words
} # If Not The End of all Files
} # End of While Not At The End Of All Files
```

Printing Different HTML in Case No Keywords Were Found

If no keywords were found, the HTML for “getting no hits” is printed.

```
if ($number_of_hits == 0) {
&PrintNoHitsBodyHTML;
}
```

End of the Main Program

When the program is finished, the footer is printed.

```
&PrintFooterHTML;
```

The rest of the program consists of subroutines that were called in the main program.

The FindKeywords Subroutine

The FindKeywords subroutine is the core routine of the entire search engine. It accepts a line of a file and the keywords to search for in that line. If a keyword is found, the routine deletes it from the keyword array (@not_found_words). Thus, when the @not_found_words array no longer has any elements, the script knows that all the keywords in the file have been found.

```
sub FindKeywords
{
```

There are three parameters. The first one, $exact_match, is equal to on if the pattern match is based on an exact one-to-one match of each letter in the keyword to each letter in a word contained in the HTML document. The second parameter, $line, is a line in the HTML file that is currently being searched for the keywords. The final parameter, *not_found_words, is a reference to the array @not_found_words, which contains a list of all the keywords not found so far. As keywords get found in the searched file, words are removed from this array. When the array is empty, we know that the file contained all the keywords. In other words, there are no “not found words” if the search is successful.

```
    local($exact_match, $line, *not_found_words) = @_;
    local($x, $match_word);
```

If the exact match is on, the program matches all the words in the array by surrounding the keywords with \b. This means that the keyword must be surrounded by word boundaries to be a valid match. Thus, the keyword “the” would not match a word such as “there,” because the characters are only part of a larger word.

```
if ($exact_match eq "on") {
     for ($x = @not_found_words; $x > 0; $x--) {
         $match_word = $not_found_words[$x - 1];
         if ($line =~ /\b$match_word\b/i) {
```

The splice routine can be used to cut out the words if they are satisfied by the search. The splice command is a Perl routine that accepts the original array, the element in the array to splice, the number of elements to splice, and a list or array to splice into the original array. Because we are omitting the fourth parameter of the splice, the routine by default splices "nothing" into the array as the element number. This technique deletes the element in one convenient little routine.

```
          splice(@not_found_words,$x - 1, 1);
         } # End of If
     } # End of For Loop
```

If the exact match is not on, the program will report a match if the letters in the keyword exist anywhere on the line, whether or not the keyword is part of a larger word. All the searches are case-insensitive, as indicated by the `i` following the slashes that define the search term.

```
} else {
     for ($x = @not_found_words; $x > 0; $x--) {
         $match_word = $not_found_words[$x - 1];
         if ($line =~ /$match_word/i) {
          splice(@not_found_words,$x - 1, 1);
         } # End of If
     } # End of For Loop
    } # End of ELSE

} # End of FindKeywords
```

The GetNextEntry Subroutine

The `GetNextEntry` subroutine reads the directory handle for the next entry in the directory. The routine accepts as parameters the current directory handle and the current directory path.

```
sub GetNextEntry {
    local($dirhandle, $directory) = @_;
```

If the next entry is a file, the program checks to see whether the file has an **.htm** or **.html** extension. This is accomplished by using the regular expression "`/htm.?/i`." The "`.?`" matches any character once after the `htm`. The `i` after the search terms tells the program to treat uppercase and lowercase characters equally.

```
    while ($filename = readdir($dirhandle)) {
     if (($filename =~ /htm.?/i) ||
```

If the next entry is a directory, the routine returns the directory name if it is not a directory that is "." or "..".

```
         (!($filename =~ /^\.\.?$/) &&
          -d "$directory/$filename")) {
```

If the program satisfies one of these two conditions, the `while` loop that reads subsequent directory entries is exited using the `last` command, and the found filename or directory name is returned from the subroutine.

```
         last;
     } # End of IF Filename is a document or directory
    } # End of while still stuff to read

    $filename;
} # End of GetNextEntry
```