The Answer Girl

EGALITARIANISM

Have your Web links got split ends, simply because upper and lower case notation was ignored when they were produced? Patricia Jung shows us how to resolve this problem using a Perl script

At last, the commissioned Web site is finished. A sigh of relief in the office is closely followed by the sobering realisation that the Web designer has been working under Windows and has not been all that careful with upper and lower case notation of filenames, because on her Microsoft test computer index.html, Index.html and INDEX.HTML are all identical notations for a single file. Unix file systems on the other hand, such as the ones mainly used under Linux, ext2fs and ReiserFS, insist that a capital A and a lower case a are completely different things – even in file names.

The site's relaunch site still has to take place on time, and who wants to sit down and correct all the wrong **A HREF** details by hand across several dozen files? So the question arises, as to how the whole deal can be dealt with automatically?

Defining the task

The task is by no means trivial because there is quite a bit to do. The first thing is to find all the references, search out all those which relate to local files, extract the corresponding filenames together with **path** and check whether there is a file of this name at the appropriate place in the file system.

If the designations of the file in the link and in the file system match, we need do nothing. If they are completely different, the best thing to do is to add a comment, to the effect that this point will need re-processing by hand. If the details differ only in the upper and lower case lettering, we can adapt the filename in the link details.

This does not look like something that can be solved simply with a couple of command line tools and a few **pipes**. Instead we will have to stick out our necks, do it properly and write a little script.

A shell script, a sed-script, an awk-script... – there are a number of options – but so that this article does not become overlong, let's agree on a Perl script. This is a good idea anyway as Perl's **regular expressions** lighten the load a bit when it comes to search and replace operations. This would also work with sed, but since we have to check the presence of the files in the file system, sed could only cope with the aid of other shell tools. Perl has advantages here, since as a "real" programming language it also has functions for accessing the file system and is faster than a shell script.

Awk comes into its own especially when working with columns, which in this case we do not want. It must be emphasised at this point that the use of a specific tool for a specific task always depends on one's personal tastes. If you prefer **Python** or **Tcl**, that's perfectly all right.

Unfortunately, Perl also has some drawbacks. Although, or more precisely because, there is masses of documentation – with manpages and tutorials on the Web as well as paper books – finding help on a specific task is a highly time-consuming job. Since Perl also wants to be "human", by allowing several notations for a syntax that is normally fixed in other programming languages, writing Perl code looks simpler at first glance, but the reading is then made more difficult if a Perl script originates from people with different Perl customs. Of course this versatility does not make learning Perl any easier.

The Answer Girl

The fact that the world of everyday computing, even under Linux, is often good for surprises, is a bit of a truism: Time and again things don't work, or at least not as they're supposed to. The Answer-Girl in Linux Magazine shows how to deal elegantly with such little problems.

Perls in action

All this lamentation is useless if the release date for the new Web site is imminent. So turn to your favourite editor and create a new file. Let's call it, cgks as an abbreviation for "change upper to lower case notation" – who would want to call up a program starting with an "A" – as in "Alter"?.

As in every script, the first line comes easily: It consists of a special comment, stating which interpreter is to do its job here. Using which perl, we can find out in which path the Perl interpreter is located (provided it is installed and the corresponding directory is entered in the **search path**).
Then we should have

```
#!/usr/bin/perl
```

sitting there. When developing programs it makes life easier if the interpreter points out the snares a bit more, rather than merely griping about real syntax errors. The manpage should come up with some information on this. First, though, man perl explains to us that the Perl Manual "is split up to make accessing individual sections easier", which are reached as special manpages.

```
man perlrun       # Perl execution and options
                  manpage
```

looks for the section we need, in order to find out more about options which make debugging easier. As a matter of fact, man perlrun explains an option -w (as in "warn"), which appears suitable for our protection.

Simply take everything

Now we want to edit lots of files, ideally, all those in the current directory. Yet this is something someone else should have done before us at some point. There are many Perl scripts in this world and on the Web but those with comprehensible documentation, on the other hand, are much scarcer. Searching the Web we find a script that converts international character entities in all HTML files in the current directory into real ISO characters (Figure 1) and with

```
$^I = ".bak";
```

in front, it even makes a backup file with the ending .bak.

This last feature is one we first mark with a # at the start of the line so that the interpreter ignores it. By decommenting the line, this produces the nice side effect that in the meantime we are not even writing any files, but are being shown the result on the standard output. In any case, it's much better for testing!

In Perl, simple (scalar) variables always begin with a dollar symbol, and a funny variable such as $^I must simply be something pre-defined. As a matter of fact man perlvar explains that this means the Inplace editing, thus the editing of a file which is currently being edited is switched on or off.
And the next line,

```
@ARGV = <*.html>;
```

looks like a pre-defined variable and an array because of the preceding @, thus a one-dimensional or multidimensional value field. @ARGV, the "Argument vector", is one-dimensional and according to the perlvar manpage contains the command line arguments of the script. We can use this to define within the program which arguments it should actually be called up with – obviously with all files ending in .html.

Perl makes provisions for the argument files to be opened and to enable access, via the handle <>, to the data contained therein. So all we have to do is measure off the content line by line, until there are no more lines:

```
while( $line = <> ) { }
```

This is clearly a loop, which will run continuously as long as the condition in the round brackets applies. In Perl, it makes no difference whether we declare the variable $line needed for buffering $line first, with the my() function, or only allow it to arise where we need it. It's only in connection with object-oriented Perl programming that my() really becomes important, although it does no harm to get used to this right from the start. If we again output the content of $line inside the curly brackets with

```
print $line;
```

our script should simply output the content of the .html files in the current directory line by line. We can test this in a directory that contains (not too many) HTML files. Since cgks does not presumably lie in the search path, we also state the path (perhaps with the dot as an abbreviation for the current directory).

```
pjung@chekov:~/answergirl$ ./cgks
bash: ./cgks: No such file or directory
```

No file or directory of this name? There's something

What does perl -w do?

-w prints warnings about dubious constructs, such as variable names that are mentioned only once; scalar variables that are used before being set; redefined subroutines; references to undefined file handles or file handles opened read-only that you are attempting to write on; values used as numbers that don't look like numbers; using an array as though it were a scalar; if your subroutines are nested more than 100 deep; and innumerable other things.

fishy going on here. We have in fact forgotten to assign ourselves executability rights with chmod u+x cgks.

Pattern recognition

Since the script outputs the file content so nicely for us, we can now look for the links. Perl has the nice construct of "data to be edited by default", the data hiding behind the variable $_ . When you are looking for something, there is no need even to state where to look, as long as you mean the content of $_ . We want to edit the content of $line and therefore file it with

```
$_ = $line;
```

in the default. Are there links in this line? If so, follow them to an <A HREF= within double quotes ("). The end is shown by a >. (So as not to complicate the script unnecessarily, we shall assume that there are no line breaks in a character string). As a regular expression this looks as follows:

```
<A HREF=\"(.*)\">
```

We have to escape the double quotes with the backslash, since Perl uses these to delimit string contents. In round brackets, we note the reference (either a URL or a local file specification), which is a sequence of characters of any length or .* for short.

Unfortunately, regular expressions have the habit of wanting to cover as much as possible. If, after the HREF, several "> appear on the line, the above regexp will save everything in the round brackets until the last occurrence. We wean it off this greedy habit by placing the one-time or no-times character ? after .*:

```
<A HREF=\"(.*?)\">
```

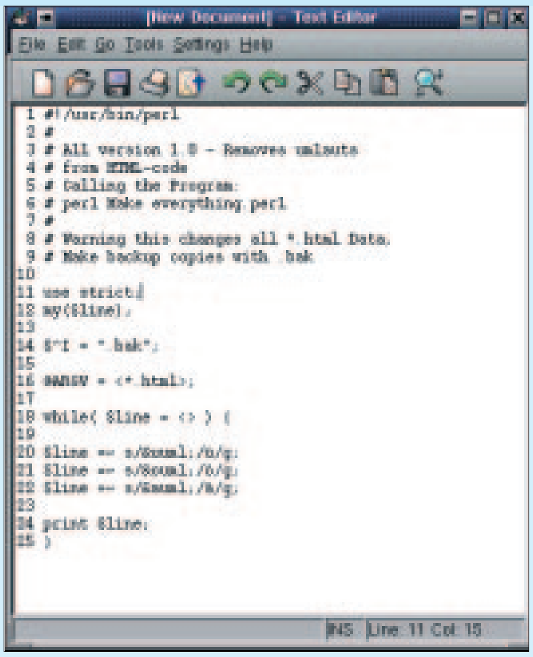
In order to look for it in the content of $_ , we make use of the 'match' operator m/pattern/. We can tell that an href can also be written in lower case ("case-insensitive search") by the i-flag. We also want to collect all the links occurring on the line and to do so we use the flag g (global):

```
@files = m/<A HREF=\"(.*?)\">/ gi;
```

We store whatever ends up in the round brackets in an array variable named @files and go through it step by step:

```
foreach $file ( @files ){ }
```

To do this we file the respective current reference in the variable $file, which as "run variable" of the foreach loop lands automatically in $_ . In order to change something solely when it is a reference to a



[New Document] - Text Editor
File Edit Go Tools Settings Help
INS Line: 11 Col: 15

```
#!/usr/bin/perl
#
# All version 1.0 - Removes umlauts
# from HTML-code
# Calling the Program:
# perl Make everything.perl
#
# Warning this changes all *.html Data.
# Make backup copies with .bak

use strict;
my($line);

$^I = ".bak";

@ARGV = <*.html>;

while( $line = <> ) {

$line =~ s/&auml;/ä/g;
$line =~ s/&ouml;/ö/g;
$line =~ s/&uuml;/ü/g;

print $line;
}
```

A small Perl script

local file, we check that its content does not begin with a protocol such as ftp or http (other protocols such as gopher can be ignored):

```
if ( ! /(ftp|http):\/\//i ){ }
```

The m of Match operator can be left out, and "ftp:// or http://" can be shortened to (ftplhttp)://. In this case, the pipe symbol l serves as a logical Or. Since the forward slashes are already framing the pattern, we must escape them, and in order to ignore upper and lower case notation, we make use of the i-flag of the match operator. Lastly, the exclamation mark ensures that the condition is met precisely when no match is found.

Local only

If there is no protocol hiding in the reference, we try to open the file:

```
open( FILE, $file );
```

The first argument, FILE, is a so-called handle, so is a stand-in for the file whose name is hiding in $file. If we get the file open, we have nothing to correct for this link and can close the file again:

```
close FILE;
```

If, on the other hand, the opening goes wrong...

```
if ( ! open( FILE, $file ) ){ }
```

...we must try to find out the right file name. If that in the link specification starts with a / , as absolute pathname it does not relate to the root directory of the file system but to the corresponding document root on the Web server. We must first pin down the directory in a variable, which serves as the start directory for the files on the Web site:

```
$rootDir = "/home/pjung/LM/LM1001/answergirl";
```

This circumstance makes our task a bit harder: To find a file specified in the link, which begins with /, it's best to look for it, not in /, but in $rootDir. If there is anything to correct though, $rootDir/corrected_name must not be written back into the link, but only /corrected_name.

So what could be more obvious than saving the corrected name and the prefix we need to find the file in the file system separately?

If one combines the contents of both variables, $prefix and $corrfil with the dot operator . , we obtain the file details made to measure for the file system. If we take only $corrfil, we have the specification matching the link. So we first write a / in the variable intended to contain the corrected link:

```
$corrfil = "/";
```

We also note the root directory as prefix:

```
$prefix = $rootdir;
```

Using this preparation for the later combination in the corrected file name, we can do without the slash at the beginning (^) of the link details save in $file. In order to formulate the condition under which the $corrfil and $prefix, as just written, should be set we therefore do not use the match but the substitute operator s. We simply tell the script: "If, at the start of $file, you can replace a / by nothing, set $prefix and $corrfil as just discussed".

Unfortunately, / is another case for the escape character. Luckily there is also an option of separating the patterns to be searched for and replaced from each other, not only by /, but also by other special characters. For example, take the dollar sign. This turns "Search / at start of string, and replace it with nothing" into not so much an escape orgy, but rather a simple s$^/$$. In order to make this replacement directly in the variable $file, we say

```
$file =~ s$^/$$;
```

Formulated as a condition, this looks as follows:

```
if ( $file =~ s$^/$$ ){ }
```

If the link was not an absolute one, $corrfil remains empty. In the prefix the dot is saved as a stand-in for the current directory, together with separator slash:

```
else {
$prefix = "./";
}
```

Bit by bit

If the link is ever broken, this can be due to the file name itself or else to a directory specified in the path. Consequently, we must swallow the bitter pill and check every component separated by / from root to tip. To do this, we divide the content of $file, which

A HREF In order to use a hyperlink on a Web site to point to another file or Internet resource, you have to write an anchor into the text. This is supplemented by the hyperreference, which states where the link is pointing, for example A HREF="http://www.linux-magazine.co.uk/". To prevent the reader of the page from seeing this specification in the text, it is placed in pointed brackets (<A HREF="http://www.linux-magazine.co.uk/">). This is followed by the text, which should be clicked on in order to get to the reference. Last of all comes the end tag </A>, with which the anchor is completed: <A HREF="http://www.linux-magazine.co.uk/">Linux Magazine</A>.

Path The road that paves the way, along directories, to a file. Absolute paths begin at the root point of the file system indicated by /.

Pipe A pipe through which the output of a command line program is transmitted. The end of the pipe serves as input for a second tool. Symbolised by a vertical stroke I : command1 I command2.

Regular expressions Option used by various standard Unix tools to express patterns. A dot stands for any symbol you like or a letter for itself. If an asterisk follows, whatever is covered by the preceding pattern can occur any number of times, or even not at all. A question mark on the other hand means that whatever it relates to occurs precisely no times, or once.

Python An object-oriented **script language**.

Script language Programs written in script languages do not have to be separately compiled, but can be executed with an interpreter direct from the source code. Often (in Perl for example) the interpreter compiles an internal binary program to increase the speed of execution, although users will not usually notice anything in normal circumstances. Since there is no need to call up a compiler, interpreted languages are especially suitable for small programs, which are quickly jotted down to solve a problem and are not intended to be used by third parties.

Tcl A script language which is usually used in connection with the GUI toolkit Tk for writing graphical applications. It can also be used without Tk.

Search path If one enters a command, the shell searches in the directories saved in the environment variable PATH, in sequence, for an executable file of the same name. The first find is used; if the shell finds nothing, it outputs the error message command not found, even if the command exists elsewhere in the file system.

may have been robbed of a leading slash, at the / points into little bits and save them in the array @parts:

```
@parts = split( /\//, $file );
```

The split() function needs two arguments: which string it should chop up, and the separator. Instead of simply specifying a delimiter, the match operator comes into play at this point. Between its two / wings, we set the slash / separating the directories, and so that this cannot be confused with the right wing / , a \ comes before it.

We now take a close look at each particle in succession:

```
foreach $part ( @parts ){ }
```

If the path component in $part is a dot for the current or (ll) double dots for the superior directory, we do not need to check any notation and add the content of $part to the string already in $corrfil:

```
if ( $part eq "." || $part eq ".." ){
$corrfil .= $part . "/" ;
}
```

Perl has two equality operators: one for numeric values and one for character strings. The latter is called eq ("equal"). $corrfil .= $part;

```
$corrfil = $corrfil . $part;
```

With the appendix operator for character strings, the dot, we also insert a slash as directory separator.

On the other hand if we have a real file or directory name sitting in $part, there is more to be done. First, we try to verify what has been in $corrfil until now: With $prefix in front, we are dealing with a directory which needs to be opened:

```
opendir (DIR, $prefix . $corrfil );
```

We will close it later using closedir(DIR);. But if we do not manage to open it, we can give up immediately and the stop processing the @parts:

```
opendir (DIR, $prefix . $corrfil ) || last ;
```

last leaves the active loop, so that we can continue with processing the next $file. If, on the other hand, we did open the directory $prefix . $corrfil and were able to "install" the handler DIR, it is best to use

```
readdir( DIR );
```

to read out all the files in it. Is there a file or directory in there with the name which is saved in $part? On the shell we would use the grep command for this – and neatly enough, it's the same in Perl:

```
grep ( /$part/i , readdir( DIR ) );
```

The pattern here is surrounded by the match operator – and of course the i-option must be there too, since the upper/lower case notation of the actual file can be completely different from $part.

However, we have left one thing out: grep also finds matches in this version when the content of

cgks as a whole

```
#!/usr/bin/perl -w

$^I = ".bak";
@ARGV = <*.html>;

$rootDir = "/home/pjung/LM/LM1001/answergirl";

while ( $line = <> ){
$_ = $line;
@files = m/<A HREF=\"(.*)\">/ig;
foreach $file ( @files ){
 $corrfil = "";
 if ( ! /(ftp|http):\/\//i ){
  if ( ! open( FILE, $file ) ){
   if ( $file =~ s$^/$$ ){
    $prefix = $rootDir;
    $corrfil = "/";
   } else {
    $prefix = "./";
   }
@parts = split( /\//, $file );
foreach $part ( @parts ){
    if ( $part eq "." || $part eq ".." ){
     $corrfil .= $part . "/" ;
    } else {
     opendir (DIR, $prefix . $corrfil ) || last ;
     @selection = grep ( /^$part$/i , readdir( DIR ) );
     closedir( DIR );
     if ( $#selection < 0 ){
      $corrfil = ("<!-- " . $file . " not found! -->" );
      last;
     } elsif ( $#selection > 0 ){
      $corrfil = ("<!-- " . $file . " not clear! -->" );
      last;
     } else {
      $corrfil .= $selection[0];
     }
     $corrfil .= "/";
    }
   }
   $corrfil =~ s+/$++;
   $line =~ s+$file+$corrfil+;
  }
  close FILE;
 }
}
print $line;
}
```

$part is only a component of an existing file name or directory name. In this case the match must be specified precisely: We state it inclusive of beginning (^) and end ($) and save the result in an auxiliary array:

```
@selection = grep ( /^$part$/i , readdir➘
( DIR ) );
```

If @selection now contains nothing – not even a zeroed element – we cannot create a corrected version of the link and are misusing $corrfil for an HTML comment which says that $file could not be found. It continues, with the aid of last at the start of the loop, with any subsequent element of @files:

```
if ( $#selection < 0 ){
$corrfil = ("<!-- " . $file . " not found!➘
-->" );
last;
}
```

Perl is endowed with a wealth of funny special character combinations, which provide you with some involuntary memory training. If one pinches the @ from an array and replaces it with a $#, one obtains ($) a scalar variable, in which (#) the number of array members is saved.

If the yield saved in @selection was a bit too successful (we recall that directory and Directory can exist side by side on Unix systems without any problem), we add $corrfil to a comment which says that there are several options:

```
elsif ( $#selection > 0 ){
$corrfil = ("<!– " . $file . " not clear!➘
–>" );
last;
}
```

Only if we find precisely one variant can we attach the zeroed element from @selection to $corrfil:

```
else {
$corrfil .= $selection[0];
}
```

There is still a trailing slash to be attached, in order to prepare $corrfil for new sub-directory levels. To blsd – if $part now contained the filename, this too has a slash at the end, although it goes no further now. Nevertheless, despite how far as we have come with our script it's still only a script for ironing out a few upper/lower case notation errors – so we are going to take the liberty of an evil hack: We replace the slash at the end of $corrfil with nothing. And because it's so nice, we use the plus as separator for the substitute operator:

```
$corrfil =~ s+/$++;
```

Uh oh, we nearly forgot to correct the $line read out from the file, by replacing the old link $file by the corrected output $corrfil...

```
$line =~ s+$file+$corrfil+;
```

... and last of all, of course, print it:

```
print $line;
```

Now the great moment approaches: Off to the test directory and let the script loose on the files in there without additional parameters, but perhaps better "piped" by less. Looking good? Then we'll just quickly remove the comment symbol from the line

```
# $^I = ".bak";
```

and already cgks is filing the converted files under their old names, with the original version as a backup file with the ending .bak to allow for a comparison.

Debriefing

There are always more elegant ways of knocking together a script. For example a quicker variant for large quantities of data could be written which remembers each directory checked, so that it does not have to look at each directory for each link again.

Far more critical for deployment is the fact that the proposed program then gives up if the HTML anchor A HREF and the link source are on different lines. The alternative was to either take this handicap in stride or build in even more complexity, especially in the regular expressions.

Those who are only occasional users of Perl, however, will not be familiar with the fact that in the Comprehensive Perl Archive Network (or CPAN), there are any number of modules stored which, like similar libraries to those for C, C++ or Java, contain untold amounts of functions for all possible and impossible areas of use.

In the case of our script, we could have saved ourselves the work with the regular expressions, for the tedious recognition of links, if we had used the HTML::Parser-Module which is already installed by default in many distributions (in SuSE in the packet perl-HTML-Parser; in Caldera in perl-modules).

The major drawback with this module is that we are moving in object-oriented Perl areas and could thus be struggling with the problem of what a parser is. For Perl novices and programming beginners this is presumably too ambitious a project.