APPENDIX

Regular Expressions

This appendix introduces regular expressions in a quick and practical manner. This introduction is only the beginning, however, as the subject is so extensive that entire books have been written on it. A good example is *Regular Expression Recipes* by Nathan A. Good (Apress, 2004). Regular expressions (regexes) are a method for precisely matching strings that fit certain selection criteria. They are an old concept from theoretical computer science and are based on finite state machines.

There are many varieties of regular expressions, and they are different in many respects. The two most frequently encountered regular expression engines are Posix regular expressions and Perl compatible regular expressions (PCRE). PHP uses the latter. Actually, it can use both, but the Posix variant is deprecated in PHP 5.3 and later. The following are the main PHP functions that implement the PCRE regular expression engine:

- `preg_match`
- `preg_replace`
- `preg_split`
- `preg_grep`

There are other functions that are part of the regular expressions machinery, but these four functions are the most frequently used ones. The "preg" in each stands for "Perl regular expression," as opposed to Posix regular expressions or "extended regular expressions." Yes, there used to be "ereg" versions of the regular expression functions, but they are deprecated as of PHP 5.3.0.

This appendix has two parts: the first explains the PCRE regular expression syntax; the second shows examples of using them in PHP scripts.

Regular Expression Syntax

The basic elements of regular expressions are meta-characters. When meta-characters are escaped by the backslash character ("\"), they lose their special meaning. Table A-1 is a list of the meta-characters.

Table A-1. Meta-Characters and Their Meanings

Expression	Meaning
.	Matches any single character.
*	Matches zero or more occurrences of the character expression preceding it.
?	Matches 0 or 1 occurrences of the character expression preceding it. It also makes regular expressions non-greedy. (An example of greedy and non-greedy regular expressions will come later.) It is also used for setting internal options.
/	Regular expression delimiter. It marks the beginning and the end of the regular expression.
+	Matches 1 or more occurrences of the character expression preceding it.
[ ]	Character classes: [a-z] are all lowercase letters. [Ab9] would match any of the characters 'A','b' or '9'. It is also possible to negate the character class with the "^" at the beginning. [^a-z] would match any character except lowercase letters.
^	Beginning of a line.
$	End of a line.
( )	Match groupings. That will be explained in detail later.
|	This is the "or" expression, separating two sub-expressions.
{}	Quantifiers. \d{3} means "3 digits", \s{1,5} means "one, two, three, four or five space characters", Z{1,} means "one or more letters Z." That is synonymous with Z+.

In addition to meta-characters, there are also some special character classes, which are listed in Table A-2.

Table A-2. Special Character Classes

Class Symbol	Meaning
\d,\D	Lowercase symbol "\d" matches a digit. Uppercase symbol "\D" is a negation, and matches a non-digit character.
\s,\S	Lowercase "\s" matches a space character or a tab character. Uppercase is a negation that matches any non-space character.
\w,\W	Lowercase "\w" matches a "word character", that is a letter or a digit. As with the previous example, "\W" is a negation of "\w" and matches "any non-word character."

Regular expression `.*` will match any character. Regular expression `^.*3` will match characters from the beginning of the line until the last digit “3” on the line. This behavior can be changed; we’ll cover this later in the section about greediness. For now, let’s see some more examples of regular expressions.

Regular Expression Examples

First, let’s see dates. Today, it is Saturday, April 30, 2011. The first pattern to match the date in this format would look like this:

```
/[A-Z][a-z]{2,},\s[A-Z][a-z]{2,}\s\d{1,2},\s\d{4}/
```

The meaning of that is “A capital letter, followed by at least two lowercase letters and a comma, followed by space, uppercase letter, at least two lowercase letters, space, 1 or 2 digits, comma, space and, finally, precisely four digits for a year.” Listing A-1 is a small PHP snippet to test regular expressions:

Listing A-1. Testing Regular Expressions

```
<?php
$expr = '/[A-Z][a-z]{2,},\s[A-Z][a-z]{2,}\s\d{1,2},\s\d{4}/';
$item = 'Saturday, April 30, 2011.';
if (preg_match($expr, $item)) {
    print "Matches\n";
} else {
    print "Doesn't match.\n";
}
?>
```

Note that the `$item` variable in Listing A-2 has a dot at the end, while our regular expression finishes with `\d{4}` for the year and doesn’t match the dot at the end. If we didn’t want that, we could have “anchored” the regular expression to the end of line, by writing it like this: `/[A-Z][a-z]{2,},\s[A-Z][a-z]{2,}\s\d{1,2},\s\d{4}$/`. The newly added dollar sign at the end of the expression means “end of the line,” which means that the regular expression will not match if there are any trailing characters on the line after the year. Similarly, we could have anchored the regular expression to the beginning of the line by using the meta-character “^”. The regular expression that would match the entire line, regardless of the content, is `/^.*$/`.

Now, let’s take a look at a different date format, YYYY-MM-DD. The task is to parse the date and extract components.

■ **Note** This can easily be done with the date function and it is a pretty good illustration of the internal workings of some PHP functions.

We need not only to verify that the line contains the valid date; we also need to extract the year, month, and date. In order to do that, we’ll need to match groupings or sub-expressions. Match groupings can be thought of as sub-expressions, ordered by the sequence number. The regular expression that would enable us to perform the task at hand looks like this:

```
/(\d{4})-(\d{2})-(\d{2})/
```

The parentheses are used to match groupings. Those groupings are sub-expressions, and can be thought of as separate variables. Listing A-2 shows how to do that with the built-in `preg_match` function.

Listing A-2. Matching Groupings with the Built-In `preg_match` *Function*

```
<?php
$expr = '/(\d{4})-(\d{2})-(\d{2})/';
$item = 'Event date: 2011-05-01';
$matches=array();
if (preg_match($expr, $item,$matches)) {
    foreach(range(0,count($matches)-1) as $i) {
        printf("%d:-->%s\n",$i,$matches[$i]);
    }
    list($year,$month,$day)=array_splice($matches,1,3);
    print "Year:$year Month:$month Day:$day\n";
} else {
    print "Doesn't match.\n";
}
?>
```

In this script, the function `preg_match` takes the third argument, the array `$matches`. Here is the output:

```
./regex2.php
0:-->2011-05-01
1:-->2011
2:-->05
3:-->01
Year:2011 Month:05 Day:01
```

The 0th element of the array `$matches` is the string that matches the entire expression. That is not the same as the entire input string. After that, each consecutive grouping is represented as an element of the array. Let's see another, more complex example. Let's parse a URL. Generally the form of a URL is the following:

```
http://hostname:port/loc?arg=value
```

Of course, any part of the expression may be missing. An expression to parse a URL of the form described above would look like this:

```
/^https?:\/\/[^:\/]+:?\d*\/[^?]*.*/
```

There are several new elements worth noticing in this expression. First is the `s?` part in the `^http[s]?:`. That matches either `http:` or `https:` at the beginning of the string. The caret character `^` anchors the expression to the beginning of the string. The `?` means "0 or 1 occurrences of the previous expression." The previous expression was letter `s` and translates into "0 or 1 occurrences of the letter s." Also, the slash characters `/` were prefixed with the backslash characters `\` to remove the special meaning from them.

PHP is extremely lenient when it comes to the regular expression delimiter; it allows changing it to any other delimiter. PHP would recognize brackets or the pipe characters `|`, so the expression would have been equally valid if written as `[^https?://[^:/]+:?\d*/[^?]*.*]`, or even using the pipe character as a delimiter: `|^https?://[^:/]:?\d*/[^?]*.*|`. The general way of stripping the special meaning from the special characters is to prefix them with a backslash character. The procedure is also known as "escaping special characters." Regular expressions are smart and can figure out the meaning of the characters in the given context. Escaping the question mark in `[^?]*` was unnecessary because it is clear from the context that it denotes the class of characters different from the question mark. That doesn't apply to the delimiter character like `/`; we had to escape those. There is also the `[^:\/]+` part of the expression, which stands for "one or more characters different from the colon or slash." This regular expression can even help us with slightly more complex URL forms. See Listing A-3.

Listing A-3. *Regular Expressions with Complex URL Forms*

```
<?php
$expr = '[^https*://[^:/]+:?\d*/[^?]*.*]';
$item = 'https://myaccount.nytimes.com/auth/login?URI=http://';
if (preg_match($expr, $item)) {
    print "Matches\n";
} else {
    print "Doesn't match.\n";
}
?>
```

This is the login form for the *New York Times*. Let's now extract the `host`, `port`, `directory`, and the `argument` string, using groupings, just like we did in Listing A-2 (see Listing A-4).

Listing A-4. *Extracting the* `host`*, port, directory, and argument String*

```
<?php
$expr = '[^https*://([^:/]+):?(\d*)/([^?]*)\??(.*)]';
$item = 'https://myaccount.nytimes.com/auth/login?URI=http://';
$matches = array();
if (preg_match($expr, $item, $matches)) {
    list($host, $port, $dir, $args) = array_splice($matches, 1, 4);
    print "Host=>$host\n";
    print "Port=>$port\n";
    print "Dir=>$dir\n";
    print "Arguments=>$args\n";
} else {
    print "Doesn't match.\n";
}
?>
```

When executed, this script will produce the following result:

```
./regex4.php
Host=>myaccount.nytimes.com
Port=>
Dir=>auth/login
Arguments=>URI=http://
```

Internal Options

The value for the port wasn't specified in the URL, so there was nothing to extract. Everything else was extracted properly. Now, what would happen if the URL was written with capital letters, like this:

```
HTTPS://myaccount.nytimes.com/auth/login?URI=http://
```

That would not match, because our current regular expression specifies the lowercase characters, yet it is a completely valid URL that would be properly recognized by any browser. We need to ignore case in our regular expression if we want to allow for this possibility. That is achieved by setting the "ignore case" option within the regular expression. The regular expression would now look like this:

```
[(?i)^https?://([^:/]+):?(\d*)/([^?]*)\??(.*)]
```

For any match after `(?i)` the case will be ignored. Regular expression `Mladen (?i)g` would match both strings `Mladen G` and `Mladen g`, but not `MLADEN G`.

Another frequently used option is `m` for "multiple lines." Normally, regular expression parsing stops when the newline characters "`\n`" is encountered. It is possible to change that behavior by setting the `(?m)` option. In that case, parsing will not stop until the end of input is encountered. The dollar sign character will match newline characters, too, unless the "`D`" option is set. The "`D`" option means that the meta-character "`$`" will match only the end of input and not newline characters within the string.

Options may be grouped. Using `(?imD)` at the beginning of the expression would set up all three options: ignore case, multiple lines, and "dollar matches the end only."

There is also an alternative, more traditional notation for the global options that allows the global modifiers to be specified after the last regular expression delimiter. Using that notation, our regular expression would look like this:

```
[^https?://([^:/]+):?(\d*)/([^?]*)\??(.*)]i
```

The advantage of the new notation is that it may be specified anywhere in the expression and will only affect the part of the expression after the modifier, whereas specifying it after the last expression delimiter will inevitably affect the entire expression.

■ **Note** The full documentation of the global pattern modifiers is available here: `www.php.net/manual/en/reference.pcre.pattern.modifiers.php`

Greediness

Normally, regular expressions are greedy. That means that the parser will try to match as large a portion of the input string as possible. If regular expression `'(123)+'` was used on the input string `'123123123123123A'`, then everything before the letter `A` would be matched. The following little script tests this notion. The idea is to extract only the `img` tag from an HTML line and not any other tags. The first iteration of the script, which doesn't work properly, will look like Listing A-5.

Listing A-5. Insert Listing Caption Here.

```
<?php
$expr = '/<img.*>/';
$item = '<a><img src="file">text</a>"';
$matches=array();
if (preg_match($expr, $item,$matches)) {
    printf( "Match:%s\n",$matches[0]);
} else {
    print "Doesn't match.\n";
}
?>
```

When executed, the result would look like this:

```
./regex5.php
Match:<img src="file">text</a>
```

■ **Note** Some browsers, most notably Google Chrome, will attempt to fix bad markup, so both greedy and non-greedy output will exclude the stray `</a>`.

We matched more than we wanted, because the pattern "`.*>`" matched as many characters as possible until it reached the last "`>`", which is a part of the `</a>` tag, not of the `<img>` tag. Using the question mark will make the "*" and "`+`" quantifiers non-greedy; they will match the minimal number of characters, not the maximal number of characters. By modifying the regular expression into `'<img.*?>'`, the pattern matching would continue until the first "`>`" character was encountered, producing the desired result:

```
Match:<img src="file">
```

Parsing HTML or XML is a typical situation in which non-greedy modifiers are utilized, precisely because of the need to match the tag boundaries.

PHP Regular Expression Functions

So far, all we've done is to check whether the given string matches the specification, written in a convoluted form of PCRE regular expression, and extract elements from the string, based on the regular expression. There are other things that can be done with regular expressions, such as replacing strings or splitting them into arrays. This section is devoted to the other PHP functions that implement the regular expression mechanisms, in addition to the already familiar `preg_match` function. The most notable one among these is `preg_replace`.

Replacing Strings: preg_replace

The `preg_replace` function uses the following syntax:

```
$result = preg_replace($pattern,$replacement,$input,$limit,$count);
```

Arguments `$pattern`, `$replacement`, and `$input` are self explanatory. The `$limit` argument limits the number of replacements, with -1 meaning no limit; -1 is the default. The final argument, `$count`, is populated after the replacements have been made with the number of replacements actually performed, if specified. It may all look simple enough, but there are further ramifications. First of all, the pattern and replacement can be arrays, as in the Listing A-6.

Listing A-6. Insert Listing Caption Here.

```
<?php
$cookie = <<<'EOT'
    Now what starts with the letter C?
    Cookie starts with C
    Let's think of other things that starts with C
    Uh ahh who cares about the other things

    C is for cookie that's good enough for me
    C is for cookie that's good enough for me
    C is for cookie that's good enough for me

    Ohh cookie cookie cookie starts with C
EOT;
$expression = array("/(?i)cookie/", "/C/");
$replacement = array("donut", "D");
$donut = preg_replace($expression, $replacement, $cookie);
print "$donut\n";
?>
```

When executed, this little script produces a result that probably wouldn't appeal to the Cookie Monster from *Sesame Street*:

```
./regex6.php
    Now what starts with the letter D?
    donut starts with D
    Let's think of other things that starts with D
    Uh ahh who cares about the other things

    D is for donut that's good enough for me
    D is for donut that's good enough for me
    D is for donut that's good enough for me

    Ohh donut donut donut starts with D
```

The important thing to notice is that both the pattern and replacement are arrays. The pattern and replacement array should have an equal number of elements. If there are fewer replacements than patterns, then the missing replacements will be replaced by null strings, thus effectively destroying the matches for the remaining strings specified in the patterns array.

The full strength of the regular expressions can be seen in Listing A-7. The script will produce SQL-ready truncate table commands from the list of provided table names. This is a rather common task. For the sake of brevity, the list of tables will already be an array, although it would typically be read from a file.

Listing A-7. *Insert Listing Caption Here.*

```
<?php
$tables = array("emp", "dept", "bonus", "salgrade");
foreach ($tables as $t) {
    $trunc = preg_replace("/^(\w+)/", "truncate table $1;", $t);
    print "$trunc\n";
}
```

When executed, the result would look like this:

```
./regex7.php
truncate table emp;
truncate table dept;
truncate table bonus;
truncate table salgrade;
```

The use of `preg_replace` shows several things. First, there is a grouping `(\w+)` in the regular expression. We saw groupings in the previous section, when extracting date elements from a string in Listing A-2. That grouping also appears in the replacement argument, as "`$1`". The value of each sub-expression is captured in the variable `$n`, where `n` can range from 0 to 99. Of course, as is the case with `preg_match`, `$0` contains the entire matched expression and the subsequent variables contain the values of sub-expressions, numbered from left to right. Also, note the double quotes here. There is no danger of confusing variable `$1` for something else, as the variables of the form `$n`, `0<=n<=99` are reserved and cannot be used elsewhere in the script. PHP variable names must start with a letter or underscore, that is a part of the language specification.

Other Regular Expression Functions

There are two more regular expression functions to discuss: `preg_split` and `preg_grep`. The first of those two functions, `preg_split`, is the more powerful relative of the `explode` function. The `explode` function will split the input string into array of elements, based on the provided delimiter string. In other words, if the input string is `$a="A,B,C,D"`, then the `explode` function, used with the string "`,`" as the separator, would produce the array with elements "`A`", "`B`", "`C`" and "`D`". The question is how do we split the string if the separator is not of a fixed format and the string looks like `$a='A, B,C .D'`? Here we have space characters before and after the separating comma and we also have a dot as a separator, which makes using just the `explode` function impossible. The `preg_split` has no problems whatsoever. By using the following regular expression, this string will be flawlessly split into its components:

```
$result=preg_split('/\s*[,.]\s*/',$a);
```

The meaning of the regular expression is "0 or more spaces, followed by a character that is either a dot or a comma, followed by 0 or more spaces." Of course, adding regular expression processing is more expensive than just comparing the strings, so one should not use `preg_split` if the regular `explode` function suffices, but it's really nice to have it in the tool chest. The increased cost comes from the fact that regular expressions are rather complex beasts under the hood.

■ **Note** Regular expressions are not magic. They require caution and testing. If not careful, one can also get unexpected or bad results. Using a regular expression function instead of a more familiar built in function doesn't, by itself, guarantee the desired result.

The `preg_grep` function should be familiar to all those who know how to use the command line utility called `grep`. That's what function was named after. The syntax of the `preg_grep` function looks like this:

```
$results=preg_grep($pattern,$input);
```

The function `preg_grep` will evaluate regular expression `$pattern` for each element of the input array `$input` and store the matching output in the resulting array results. The result is an associative array with the offsets from the original array provided as keys. Listing A-8 shows an example, based on the file system `grep` utility:

Listing A-8. *Insert Listing Caption Here.*

```
<?php
$input = glob('/usr/share/pear/*');
$pattern = '/\.php$/';
$results = preg_grep($pattern, $input);
printf("Total files:%d PHP files:%d\n", count($input), count($results));
foreach ($results as $key => $val) {
    printf("%d ==> %s\n", $key, $val);
}
?>
```

The dot in the .php extension was escaped by a backslash character, because dot is a meta-character, so to strip its special meaning, it had to be prefixed by "\". The result on the system used for writing this appendix looks like this:

```
./regex8.php
Total files:35 PHP files:12
4 ==> /usr/share/pear/DB.php
6 ==> /usr/share/pear/Date.php
8 ==> /usr/share/pear/File.php
12 ==> /usr/share/pear/Log.php
14 ==> /usr/share/pear/MDB2.php
16 ==> /usr/share/pear/Mail.php
19 ==> /usr/share/pear/OLE.php
22 ==> /usr/share/pear/PEAR.php
23 ==> /usr/share/pear/PEAR5.php
27 ==> /usr/share/pear/System.php
29 ==> /usr/share/pear/Var_Dump.php
32 ==> /usr/share/pear/pearcmd.php
```

The result may look differently on a different system, which has different PEAR modules installed. The `preg_grep` function can save us from checking regular expressions in a loop, and is quite useful.

There are several other regular expression functions that are used much less frequently than the functions described in this appendix. Those functions are very well documented and an interested reader can look them up in the online documentation, at `www.php.net`.

Index