35

# Regular Expressions – Extra Elements

*As well as matching a pattern, a regular expression can be used to filter out part of a string. This is a very powerful feature, but then you'll also need to learn about capture brackets, sparse and greedy matches, modifiers and more.*

## 35.1 Capturing matches

It's fabulous that you can use a regular expression to check whether a string includes a particular pattern, perhaps an ISBN number with four groups of digits totalling 10 digits in length. But once you know that the match has been achieved, you'll want to go on and make use of that ISBN number. It would be a shame to go to all the sophistication of a regular expression match to identify that you had a match if you then had to revert to low-level string handling to extract the part of the string that matched, or the various parts of the match string that are of interest to you.

If you round brackets around a part of your regular expression, they have two significances. Firstly, any count that follows the brackets is applied to the brackets, and secondly the matching part is stored into matching variables that can be used later in the matching process and perhaps (depending on the particular language and function you're using) be saved for you.

Our sample program reports matches identified by brackets. If I look for words ending with "fish", I get back true/false values:

```
Please enter a regular expression: \w+fish
Good ... let's match ....
Match, line 99: 192.168.200.71   swordfish
returned: 1
Match, line 104: 192.168.200.76  dogfish
returned: 1
Match, line 105: 192.168.200.77  catfish
returned: 1
Match, line 197: 192.168.200.168 macdonald donald cuttlefish scanhost #whc Original Mac laptop
returned: 1
Total of 4 matches to \w+fish
```

But if I use brackets, I can get back the letters before "fish":

```
Please enter a regular expression: (\w+)fish
Good ... let's match ....
Match, line 99: 192.168.200.71    swordfish
returned: sword
Match, line 104: 192.168.200.76   dogfish
returned: dog
Match, line 105: 192.168.200.77   catfish
returned: cat
Match, line 197: 192.168.200.168  macdonald donald cuttlefish scanhost #whc Original Mac laptop
returned: cuttle
Total of 4 matches to (\w+)fish
```

or even the whole word:

```
Please enter a regular expression: (\w+fish)
Good ... let's match ....
Match, line 99: 192.168.200.71     swordfish
returned: swordfish
Match, line 104: 192.168.200.76    dogfish
returned: dogfish
Match, line 105: 192.168.200.77    catfish
returned: catfish
Match, line 197: 192.168.200.168   macdonald donald cuttlefish scanhost #whc Original Mac laptop
returned: cuttlefish
Total of 4 matches to (\w+fish)
```

If you want to rematch a captured string , you can refer to **\1**, **\2**, etc. for the first and second bracket, and so on:

```
Please enter a regular expression: (\w)o\1
Good ... let's match ....
Match, line 72: 192.168.200.44    bob blyth
returned: b
Match, line 96: 192.168.200.68  cockle
returned: c
Match, line 116: 192.168.200.88    cocker
returned: c
Match, line 175: 192.168.200.146   cocoa eddie
returned: c
Match, line 292: 192.168.200.248 dodo
returned: d
Total of 5 matches to (\w)o\1
```

In Perl, captured matches are also saved into variables **$1**, **$2** and so on, after the completion of the match, and in matches in PHP you may give an extra parameter to your match function call. Note that if you specify brackets within brackets, the variables are assigned in the order of the **(** characters, and if you apply counts to brackets, the last match to a bracket set will be saved. Thus:

```
Please enter a regular expression: ((m(\w)+)don\w+)
Good ... let's match ....
Match, line 197: 192.168.200.168   macdonald donald cuttlefish scanhost #whc Original Mac laptop
returned: macdonald mac c
Total of 1 matches to ((m(\w)+)don\w+)
```

Let's see a real example. This is a Perl program that matches ISBN numbers and performs a series of checks. If the ISBN number has valid components, each is printed out. Regular expressions are not well suited to calculate the check character, so that's done with other code:

```
#!/usr/bin/perl

print "Please enter ISBN number: ";
chop ($isbn = <STDIN>);

if ($isbn !~ /^(\d-?){9}[0-9xX]$/) {
        die "Incorrect digit count for an ISBN\n";
        }
if (! (($gid,$pub,$book,$check) =
        ($isbn =~ /^(\d+)-(\d+)-(\d+)-([0-9xX])$/))) {
        die "Incorrect group count for an ISBN\n";
        }

print "Group id (country and region): $gid\n";
print "Publisher                      $pub\n";
print "Individual book number         $book\n";
print "Check character entered        $check\n";

$compressed = "$gid$pub$book";
$runtot = 0;
for ($k=0,$f=10; $k<9; $k++,$f--) {
        $runtot += $f * substr($compressed,$k,1);
        }
```

```
$check_calc = (11 - $runtot % 11) % 11;
$check_calc = "X" if ($check_calc == 10);

print "\nCheck character calculated:    $check_calc\n";
if ($check_calc eq uc($check)) {
        print "Test PASSED - a valid ISBN number\n";
} else {
        print "Test FAILED - Correct format, but checksum wrong\n";
        }

[localhost:~/q802] graham% perl isbn
Please enter ISBN number: 12345678
Incorrect digit count for an ISBN
[localhost:~/q802] graham% perl isbn
Please enter ISBN number: 0-596-00289-0
Group id (country and region): 0
Publisher                      596
Individual book number         00289
Check character entered        0

Check character calculated:    0
Test PASSED - a valid ISBN number
[localhost:~/q802] graham% perl isbn
Please enter ISBN number: 0-596-00415-x
Group id (country and region): 0
Publisher                      596
Individual book number         00415
Check character entered        x

Check character calculated:    X
Test PASSED - a valid ISBN number
[localhost:~/q802] graham% perl isbn
Please enter ISBN number: 0-595-55002-6
Group id (country and region): 0
Publisher                      595
Individual book number         55002
Check character entered        6

Check character calculated:    9
Test FAILED - Correct format, but checksum wrong
[localhost:~/q802] graham%
```

```
|
Non greedy
Look aheads and look behinds
\A and \Z
```

## 35.2 Which match?

Let's suppose that you're using the regular expression:

**`\S+@\S+`**

to extract an email address from a line of text such as

`You could try graham@wellho.net or wellho@aol.com perhaps.`

What is the email address you want? *graham@wellho.net*, or *wellho@aol.com*? Why not *graham@w* or *o@ao*, both of which match the pattern specified. In fact, there are 102 possible ways that the regular expression could match.

While you're simply asking "does this string contain an email address?", the question of which match is academic. But when you want to use the saved variable it becomes significant. The rules to remember are:

d) The matching starts as far to the left as possible

e) The counts you have encountered so far are greedy (in other words "*****" means "0 or more", but has a subtext "As many as possible")

So taking our email address match, of

**`\S+@\S+`**

to

**`You could try graham@wellho.net or wellho@aol.com perhaps.`**

f) Match to **`\S+`** attempt starts at Y-o-u, and fails to find any suitable match to the space after the "u". It backtracks character-by-character, trying to match the @ instead, but fails

g) Match to **`\S+`** attempts to start at c-o-u-l-d and similarly fails, and attempts to start at t-r-y and fails yet again

h) Match to **`\S+`** attempts to start at g-r-a and it matches all the way to n-e-t, at which point it fails because the following space character does not match either a non-space or an @

i) This time, the backtracking steps back character-by-character until it gets to the @, which matches against the @ in the regular expression as well as against the **`\S+`**

j) The @ matches exactly

k) Matching to the second **`\S+`** proceeds from the w-e-l-l-h-o right through to the t of n-e-t

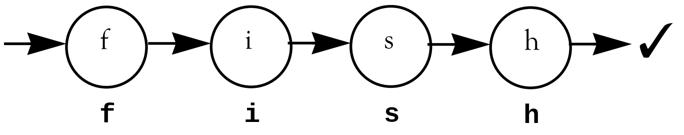
l) Matching fails to match any further characters at this point, as the next character, a space, fails to match against **`\S`**

m)The regular expression engine notes that it has matched every element of the regular expression correctly but cannot successfully match any further, therefore it accepts the match it has

| | | |
|---|---|---|
| **`\S+`** | matched | graham |
| @ | matched | @ |
| **`\S+`** | matched | wellho.net |

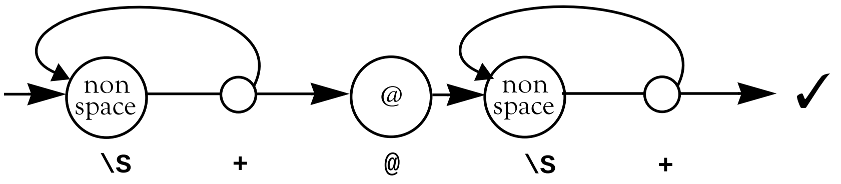
If you want to work out how a regular expression matches, you may wish to interpret the expression into a diagram, a drawing of what is known as a "regular expression engine" for processing your string. Here's a diagram that matches against the regular expression "fish":

Figure 194 Matching to the regular expression "fish"



and here's the diagram for our email address match:

Figure 195 Matching to an email address



Where there's a choice to be made on these diagrams:
1. The current situation is stored for possible backtracking
2. The uppermost path from the choice is fully explored
3. If the uppermost path fails, the next path (or lower most) is explored
4. If all possible paths fail, then the match fails as a whole

## 35.3 Alternation

Within a regular expression, you can use character groups to look for any one of a series of characters. These are fundamental regular expression elements and you can write a character group in square brackets, or using extended notations such as **\p**.

On other occasions, somewhat differently, you'll want to be able to select one string of characters (or sub-expression) or another. For example, you might want to accept http or ftp; this is known as alternation, and is available in regular expressions using the **|** (pipe) character. Thus:

```
        http|ftp
```

By default, alteration makes the entire regular expressions to the left and right of the **|** character into alternatives,thus:

```
Please enter a regular expression: ^195|bob
Good ... let's match ....
Match, line 72: 192.168.200.44    bob blyth
returned: 1
Match, line 303: 195.152.7.132  seal_h          #=4
returned: 1
Total of 2 matches to ^195|bob
```

which means "starting with 195 or containing Bob":

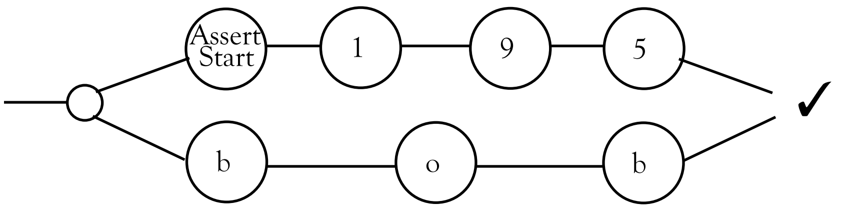


Figure 196 Matching an alteration

You'll often want to limit the scope of your alternation, which you can do using round brackets:

```
Please enter a regular expression: ^(195|bob)
Good ... let's match ....
Match, line 303: 195.152.7.132  seal_h          #=4
returned: 195
Total of 1 matches to ^(195|bob)
```

which is fine, once you appreciate that as a side effect you've also set the **\1** match variable. The diagram now looks like this:

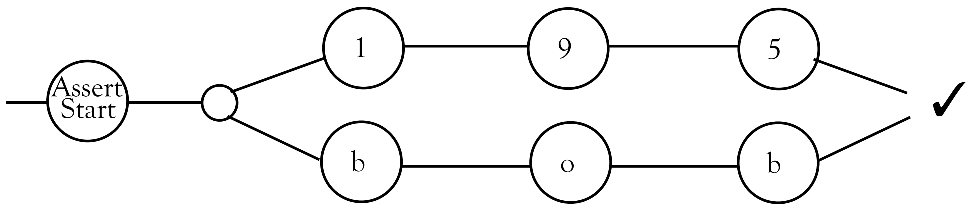


Figure 197 Matching an alteration

You could always "program around" the unwanted capture brackets if you wanted to, but most regular expression handlers also offer you a non-capturing bracket alternative, **(?:** through to **)**.

Thus:

```
Please enter a regular expression: ^(?:195|bob)
Good ... let's match ....
Match, line 303: 195.152.7.132  seal_h          #=4
returned: 1
Total of 1 matches to ^(?:195|bob)
```

## 35.4 Sparse matches

Counts default to greedy matching. There's a subtext to **+** that says "and match as many characters as possible. Most of the time that turns out to be exactly what you want. Look back to the earlier example in this module. Did you want to match the whole of an email address rather that just a part of it?

In a small percentage of cases, this greedy match isn't what you want. Consider matching:

**`<(.+)>`**

against:

**`Use the i tag for <i>italic</i> text`**

and what gets captured into **`\1`**? Not simply the letter i as you would expect, but the whole segment of the incoming string after the first **<** to the *final* **>**. In other words,

**`i>italic</i`**

The first solution to such an issue would be to limit the match within the brackets to exclude the terminating character so our regular expression would become:

**`<([^>]+)>`**

This looks a bit ugly, but works very well in this case. With a more complex regular expression, rewriting matches in this way could become impractical. You may prefer to use the following sparse matches provided by most regular expression handlers:

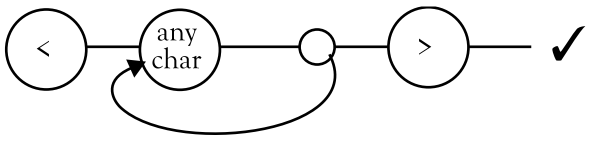
| | |
|---|---|
| **`*?`** | 0 or more, but as few as possible |
| **`+?`** | 1 or more, but as few as possible |
| **`??`** | 0 or 1, preferably 0 |
| **`{2,6}?`** | from 2 to 6, as few as possible |
| **`{5,}?`** | 5 or more, as few as possible |

Our tag match can now be re-written

**`<(.+?)>`**

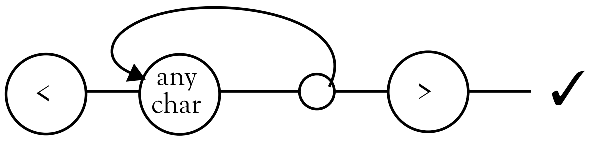
and a diagram should look like:

Figure 198 Matching to a tag



as opposed to:

Figure 199 Matching to a tag



## 35.5 Look around, ahead, behind

In a regular expression match, you can specify individual characters of character groups, and if you do, your match engine moves on past the characters concerned before it starts matching the next element. You can also specify an assertion which checks a particular condition, but does not move on.

Such assertions include:

| | |
|---|---|
| **^** | at the start of a string |
| **$** | at the end of a string |
| **\b** | at a word boundary |

There are a few occasions that you might want to assert that the following character must be one particular literal, or a character from a group, and you do not want to move on over that character. Perhaps you want to hold it back for capture, for example; such a (rare) form or match is known as a look ahead.

Example:

Let's say that I have a line of the format:

**`01225 708225 (voice)`**

and I want to save the main phone number (in this case 708225) as one capture, and the (voice) string as another capture. How to do it?

One possibility is to look for a series of digits, and then to do a look ahead assertion that they're going to be followed by space - **`(`**. Once that assertion has been met, we're then free to capture (re-match) the following characters into our special variable.

A positive look ahead assertion is written **`(?=`** through to **`)`**. In other words, we're looking ahead to check that we do match against a particular string, but that's not a consuming match:

```
Please enter a regular expression: (\d+)(?=\s\()\s(\S+)
Good ... let's match ....
Match, line 16: 01225 708225 (voice)
returned: 708225 (voice)
Match, line 17: 01225 707126 (fax)
returned: 707126 (fax)
Total of 2 matches to (\d+)(?=\s\()\s(\S+)
```

The full range of look ahead and look behind assertions:

| | |
|---|---|
| **(?=** | look ahead, must match to succeed |
| **(?!** | look ahead, must not match to succeed |
| **(?<=** | look behind, must match to succeed |
| **(?<!** | look behind, must not match to succeed |

## 35.6 Modifiers

There are times you'll wish to modify the overall behaviour of a regular expression match, and such capabilities are provided by various languages outside the regular expression itself.

### Ignoring case.

In Perl, and also in Perl-style regular expressions in PHP, add an "i" modifier after closing regular expression termination character.

In regular PHP (POSIX) matching, extra functions are provided with an additional **`i`**; thus **`ereg`** to match and **`eregi`** to match ignoring case

In MySQL, fields that are defined as text are matched ignoring case, and fields that are described as binary are matched case-significant.

### Global matching

Perl's **`g`** modifier causes the matching to resume each time at the point that the previous match left off, thus allowing a loop to iterate though all possible matches. In a list context, all possible matches are returned in a single call.

PHP provides additional functions such as **`preg_match_all`** for global matching.

### Modifiers that fine tune individual regular expression elements

In Perl-style regular expressions:

| | |
|---|---|
| **`s`** | Causes **`.`** to match any character at all (by default, it fails to match to new line) |
| **`m`** | Causes **`^`** and **`$`** to match at embedded new line characters; by default, they only match at the very beginning and very end of the string |
| **`x`** | Causes any space characters within the regular expression to be treated as comments rather than literal. |

Finally, Perl's "o" modifier signals to the Perl compiler that the regular expression will not be changed during any particular run of the program. This is useful if you're including a variable within a regular expression to which you'll be matching a very large number of times.

## Exercise