2. Regular Expressions – AS.410.698.81.SU16 Bioperl

Basics of Regular Expressions

Regular expressions are a very powerful (and sometimes very confusing) way of searching for **pattern matches** in text. They are one of Perl's strongest and best known features. Mastering regular expressions is essential to mastering Perl.

Basics are very simple: you use the '`=~`' operator to associate a regular expression with a string. This construct returns true if the regular expression matches the string, and false if it doesn't.

The interesting part is in how you specify the regular expression.

The most basic rule: a character in a regexp matches that same character:

```
#! /usr/bin/perl

use strict;
use warnings;

my $string = "foobarbaz";

if( $string =~ m/bar/ ) {
  print "match\n";
}
else { print "no match\n" }
```

```
$ ./regexp1.pl
match
```

Metacharacters

If all **regexps** let you do was specify exact matches, they wouldn't be very exciting. Fortunately, they let you specify much much ***MUCH*** more than that. Unfortunately, the syntax and terminology used gets a bit thick.

In order to be able to specify that something in a regexp isn't supposed to literally match something, you need a way to let Perl know that you're trying to do something fancy. Enter **metacharacters**.

```
\   # meta meta
|   # alternation
()  # grouping
[]  # character classes
.   # wildcard
*   # quantifier
+   # quantifier
?   # quantifier
{}  # quantifiers
^   # anchor at beginning of string
$   # anchor at end of string
```

The effect of the `\` metacharcter depends on what it proceeds:

- Placed in front of another metacharacter, it **disables** the meta effect.
- Placed in front of certain letters, it **enables** a new meta effect.
- Placed in front of any other character, it has no effect.

Alternation

Alternation works like it did with the `case` statement in shell: you specify one or more alternatives, separated with pipes. If any one of the alternates matches, the regular expression matches. In order to specify the beginning and end of the alternation pattern, you use parens:

```
/b(ar|az)/ # matches 'bar' or 'baz'
```

Character Classes

Character classes are like alternation, but for just one character and one position at a time, using square brackets to hold the possible matches:

```
/ba[rz]/ # matches 'bar' or 'baz'; same as /ba(r|z)/
/[fg]oo/ # matches 'foo' or 'goo'
```

```
/[fg]ooba[rz]/ # matches foobar, foobaz, goobar, goobaz
/[a-z]oo/
/[a-zA-Z]oo/
/[^f]oo/
```

Related to the character classes are some of the special metacharcters formed by putting '\' in front of certain letters -- they're like "mega" character classes:

```
\d  == match any digit ( [0-9] )
\D  == match any nondigit ( [^0-9] )
\s  == match any whitespace character (newline, space, tab, etc.)
\S  == match any non-whitespace character
\w  == match any "word" character
\W  == match any "non-word" character
```

Position Matching

The **position operators** let you **anchor** a match to the beginning or end of a string.

```
^   # beginning of string
$   # end of string

/foo/   # matches 'foo' , 'foobar' , 'barfoo'
/^foo/  # matches 'foo' , 'foobar' , NOT 'barfoo'
/foo$/  # matches 'foo' , 'barfoo' , NOT 'foobar'
/^foo$/ # matches 'foo' , NOT 'barfoo', NOT 'foobar'
```

Wildcards

The **wildcard** is like, well, wild cards in card games: it matches anything you want. It's basically a character class that includes all possible characters.

```
.   # wildcard
```

A single '.' matches a single position in the string. To get more sophisticated matches, you must use **quantifiers**.

Quantifiers

If you want to match more than one character at a time, then you need to use **quantifiers**, which specify how many times to match the **preceding character:**

```
*   # quantifier
+   # quantifier
?   # quantifier
{}  # quantifiers

/a*/         == match 'a' zero or more times
/a+/         == match 'a' one or more times
/a?/         == match 'a' zero or one times
/a{MIN,MAX}/ == match 'a' at least MIN and not more than MAX times
/a{MIN,}/    == match 'a' at least MIN times
/a{COUNT}/   == match 'a' exactly COUNT times
```

Note that quantifiers can apply to **character classes** and to **things grouped with parens**:

```
/f[oa]*/      # matches 'f', 'fo' , 'fa' , 'foo' , 'foa', etc
/f[oa]+/      # matches 'fo' , 'fa' , 'foo' , 'foa', etc
/b(ar|az)?/   # matches 'b' , 'bar' , 'baz'
/f(oo){1,3}/  # matches 'foo' , 'foooo' , 'fooooo'
```

Modifiers

You can also supply **modifiers** that affect how the match is calculated:

```
/i == ignore letter case when figuring matches
/g == globally find all matches
/m == let ^ and $ match next to embedded \n
/s == let . match \n
```

There are others, but they will be introduced later, after we've covered a bit more ground. (See *Programming Perl*, chapter five, if you can't wait.)

Capturing Matches

Suppose you want to **capture** a match -- find out exactly what in the string was matched. You can do this with the paren grouping characters.

After a successful match, a special variable is set, containing the characters that were matched by each pair of parentheses. The first one is `$1`, the second is `$2`, and so on. Observe:

```
my $string = "this is an example";
$string =~ /this (\w+) an (\w+)/;
print "$1\n"; # prints 'is'
print "$2\n"; # prints 'example'
```

This technique is frequently used to **parse** data.

Substitution

There are other things you can do with regexps, other than simple matching. One of them is **substitution** -- match a section of a string, and then substitute some other text for the matched portion:

```
my $string = "foobar";
$string =~ s/bar/baz/;
# $string is now "foobaz"

# can use $1, $2 in right-hand side of s/// expression:
$string =~ s/(.{3})(.{3})/$2$1/;
# $string is now 'bazfoo'
```

Modifiers also apply to the `s///` type of matches:

```
/i == ignore letter case when figuring matches
/g == globally find all matches and apply substitution to all
/m == let ^ and $ match next to embedded \n
/s == let . match \n
```

Debugging Regular Expressions

Getting a complicated regular expression to work properly can be frustrating -- especially when you're trying to figure out how to parse a complicated data format.

Generally, the best way to approach this situation is to write a test script, just checking the regexp. Start out with a very simple form of the expression, and make sure you can get it to work. Once you've got that working, add the next piece, and test. Repeat until you have the whole expression, and you're satisfied that the regular expression is working the way you want it to.

One good way of making sure you didn't miss anything is to process your whole input file through the regular expression, just checking that your match always works:

```
open( my $IN , '<' , $file ) or die( "can't open $file ($!)" );
while( <$IN> ) {
  chomp;
  die "failed to match:\n$_\n" unless /REGEXP/;
}
close( $IN );
print "everything matched.\n";
```

Exercises

Open a shell session to the server and write some simple scripts that use regular expressions. A good file to test regular expressions on is `/usr/share/dict/words`, which is used by spellchecking programs. It's about 230,000 words, one word per line. (Is this a good file to **slurp** or not?)

How would you write a script that prints out every line in a file that matches a particular regular expression? How would you write a script to print out every line that *doesn't* match a regexp?

What happens if you **nest** groups of parens in a regexp? What gets captured by each set of parens, and which variable does it end up in?

Table of Contents