Regular Expressions

Perl's powerful ability to manipulate text comes in part from its inclusion of a computing concept known as *regular expressions*. A regular expression (often shortened to *regex* or *regexp*) is a *pattern* which describes characteristics of a string of text. A *regular expression engine* interprets a pattern and applies it to strings of text to identify those which match.

Perl's core documentation describes Perl regular expressions in copious detail; see `perldoc perlretut`, `perldoc perlre`, and `perldoc perlreref` for a tutorial, the full documentation, and a reference guide, respectively. Jeffrey Friedl's book *Mastering Regular Expressions* explains the theory and the mechanics of how regular expressions work. Even though those references may seem daunting, regular expressions are like Perl--you can do many things with only a little knowledge.

Literals

The simplest regexes are simple substring patterns:

```
my $name = 'Chatfield';
say "Found a hat!" if $name =~ B</hat/>;
```

The match operator (`//` or, more formally, `m//`) contains a regular expression--in this example, `hat`. Even though that reads like a word, it means "the h character, followed by the a character, followed by the `t` character, appearing anywhere in the string." Each character is an *atom* in the regex: an indivisible unit of the pattern. The regex binding operator (=~) is an infix operator (*fixity*) which applies the regular expression on its right to the string produced by the expression on its left. When evaluated in scalar context, a match evaluates to a true value if it succeeds.

The negated form of the binding operator (`!~`) evaluates to a false value if the match succeeds.

The qr// Operator and Regex Combinations

Regexes are first-class entities in modern Perl when created with the `qr//` operator:

```
my $hat = B<qr/hat/>;
say 'Found a hat!' if $name =~ /$hat/;
```

The `like()` function from `Test::More` works much like `is()`, except that its second argument is a regular expression object produced by `qr//`.

You may interpolate and combine them into larger and more complex patterns:

```
my $hat   = qr/hat/;
my $field = qr/field/;

say 'Found a hat in a field!' if $name =~ /B<$hat$field>/;

# or

like( $name, qr/B<$hat$field>/, 'Found a hat in a field!' );
```

Quantifiers

Regular expressions are far more powerful than previous examples have demonstrated; you can search for a literal substring within a string with the `index` operator. Using the regex engine for that is like flying your autonomous combat helicopter to the corner store to buy spare cheese.

Regular expressions get more powerful through the use of *regex quantifiers*, which allow you to specify how often a regex component may appear in a matching string. The simplest quantifier is the *zero or one quantifier*, or ?:

```
my $cat_or_ct = qr/caB<?>t/;

like( 'cat', $cat_or_ct, "'cat' matches /ca?t/" );
like( 'ct',  $cat_or_ct, "'ct' matches /ca?t/"  );
```

Any atom in a regular expression followed by the ? character means "match zero or one of this atom." This regular expression matches if there are zero a characters immediately following a `c` character and immediately preceding a `t` character. It also matches if there is one and only one a character between the `c` and `t` characters.

The *one or more quantifier*, or +, matches only if there is at least one of the preceding atom in the appropriate place in the string to match:

```
my $one_or_more_a = qr/caB<+>t/;

like( 'cat',    $one_or_more_a, "'cat' matches /ca+t/"    );
like( 'caat',   $one_or_more_a, "'caat' matches /ca+t/"   );
like( 'caaat',  $one_or_more_a, "'caaat' matches /ca+t/"  );
like( 'caaaat', $one_or_more_a, "'caaaat' matches /ca+t/" );

unlike( 'ct',   $one_or_more_a, "'ct' does not match /ca+t/" );
```

There is no theoretical limit to the number of quantified atoms which can match.

The *zero or more quantifier* is *; it matches if there are zero or more instances of the quantified atom in the string to match:

```
my $zero_or_more_a = qr/caB<*>t/;

like( 'cat',    $zero_or_more_a, "'cat' matches /ca*t/"    );
like( 'caat',   $zero_or_more_a, "'caat' matches /ca*t/"   );
like( 'caaat',  $zero_or_more_a, "'caaat' matches /ca*t/"  );
like( 'caaaat', $zero_or_more_a, "'caaaat' matches /ca*t/" );
like( 'ct',     $zero_or_more_a, "'ct' matches /ca*t/"     );
```

This may seem useless, but it combines nicely with other regex features to indicate that you don't care about what may or may not be in that particular position in the string to match. Even so, *most* regular expressions benefit from using the ? and + quantifiers far more than the * quantifierFriedl's book explains why..

Finally, you can specify the number of times an atom may match with *numeric quantifiers*. `{n}` means

that a match must occur exactly *n* times.

```
# equivalent to qr/cat/;
my $only_one_a = qr/caB<{1}>t/;

like( 'cat', $only_one_a, "'cat' matches /ca{1}t/" );
```

`{n,}` means that a match must occur at least *n* times, but may occur more times:

```
# equivalent to qr/ca+t/;
my $at_least_one_a = qr/caB<{1,}>t/;

like( 'cat',    $at_least_one_a, "'cat' matches /ca{1,}t/"    );
like( 'caat',   $at_least_one_a, "'caat' matches /ca{1,}t/"   );
like( 'caaat',  $at_least_one_a, "'caaat' matches /ca{1,}t/"  );
like( 'caaaat', $at_least_one_a, "'caaaat' matches /ca{1,}t/" );
```

`{n,m}` means that a match must occur at least *n* times and cannot occur more than *m* times:

```
my $one_to_three_a = qr/caB<{1,3}>t/;

like(   'cat',    $one_to_three_a, "'cat' matches /ca{1,3}t/"
       );
like(   'caat',   $one_to_three_a, "'caat' matches /ca{1,3}t/"
      );
like(   'caaat',  $one_to_three_a, "'caaat' matches /ca{1,3}t/"
     );
unlike( 'caaaat', $one_to_three_a, "'caaaat' does not match /ca{1,3}t/"
);
```

Greediness

The + and * quantifiers by themselves are *greedy quantifiers*; they match as many times as possible. This is particularly pernicious when using the tempting-but-troublesome "match any amount of anything" pattern `.*`:

```
# a poor regex
my $hot_meal = qr/hot.*meal/;

say 'Found a hot meal!' if 'I have a hot meal' =~ $hot_meal;
say 'Found a hot meal!'
     if 'I did some one-shot, piecemeal work!' =~ $hot_meal;
```

The problem is more obvious when you expect to match a short portion of a string. Greediness always tries to match as much of the input string as possible *first*, backing off only when it's obvious that the match will not succeed. Thus you may not be able to fit all of the results into the four boxes in 7 Down if you go looking for "loam" with:

```
my $seven_down = qr/l${letters_only}*m/;
```

You'll get `Alabama`, `Belgium`, and `Bethlehem` for starters. The soil might be nice there, but they're all too long--and the matches start in the middle of the words.

Regex anchors force a match at a specific position in a string. The *start of string anchor* (`\A`) ensures that any match will start at the beginning of the string:

```
# also matches "lammed", "lawmaker", and "layman"
my $seven_down = qr/\Al${letters_only}{2}m/;
```

Similarly, the *end of line string anchor* (\Z) ensures that any match will *end* at the end of the string.

```
# also matches "loom", which is close enough
my $seven_down = qr/\Al${letters_only}{2}m\Z/;
```

If you're not fortunate enough to have a Unix word dictionary file available, the *word boundary metacharacter* (\b) matches only at the boundary between a word character (\w) and a non-word character (\W):

```
my $seven_down = qr/\bl${letters_only}{2}m\b/;
```

Like Perl, there's more than one way to write a regular expression. Consider choosing the most expressive and maintainable one.

Sometimes you can't anchor a regular expression. In those cases, you can turn a greedy quantifier into a parsimonious quantifier by appending the ? quantifier:

```
my $minimal_greedy_match = qr/hot.*?meal/;
```

In this case, the regular expression engine will prefer the *shortest* possible potential match, increasing the number of characters identified by the .*? token combination only if the current number fails to match. Because * matches zero or more times, the minimal potential match for this token combination is zero characters:

```
say 'Found a hot meal' if 'ilikeahotmeal' =~ /$minimal_greedy_match/;
```

If this isn't what you want, use the + quantifier to match one or more items:

```
my $minimal_greedy_at_least_one = qr/hot.+?meal/;

unlike( 'ilikeahotmeal', $minimal_greedy_at_least_one );

like( 'i like a hot meal', $minimal_greedy_at_least_one );
```

The ? quantifier modifier also applies to the ? (zero or one matches) quantifier as well as the range quantifiers. In every case, it causes the regex to match as few times as possible.

In general, the greedy modifiers .+ and .* are tempting but dangerous tools. For simple programs which need little maintenance, they may be quick and easy to write, but non-greedy matching seems to match human expectations better. If you find yourself writing a lot of regular expression with greedy matches, test them thoroughly with a comprehensive and automated test suite with representative data to lessen the possibility of unpleasant surprises.

Greediness and captures? Chas. likes the idea.

Metacharacters

Regular expressions get more powerful as atoms get more general. For example, the . character in a regular expression means "match any character except a newline". If you wanted to search a list of dictionary words for every word which might match 7 Down ("Rich soil") in a crossword puzzle, you might write:

```
for my $word (@words)
{
    next unless length( $word ) == 4;
    next unless $word =~ /lB<..>m/;
    say "Possibility: $word";
}
```

Of course, if your list of potential matches were anything other than a list of words, this metacharacter could cause false positives, as it also matches punctuation characters, whitespace, numbers, and many other characters besides word characters. The `\w` metacharacter represents all alphanumeric characters (in a Unicode sense -- *unicode*) and the underscore:

```
next unless $word =~ B</lB<\w\w>m/>;
```

The `\d` metacharacter matches digits--not just 0-9 as you expect, but any Unicode digit:

```
# not a robust phone number matcher
next unless $potential_phone_number =~ /B<\d>{3}-B<\d>{3}-B<\d>{4}/;
say "I have your number: $potential_phone_number";
```

Use the `\s` metacharacter to match whitespace, whether a literal space, a tab character, a carriage return, a form-feed, or a newline:

```
my $two_three_letter_words = qr/\w{3}B<\s>\w{3}/;
```

These three metacharacters have negated forms. To match any character *except* a word character, use `\W`. To match a non-digit character, use `\D`. To match anything but a space, use `\S`.

Character Classes

If the range of allowed characters in these four groups isn't specific enough, you can specify your own *character classes* by enclosing them in square brackets:

```
my $vowels    = qr/B<[>aeiouB<]>/;
my $maybe_cat = qr/c${vowels}t/;
```

The curly braces around the name of the scalar variable `$vowels` helps disambiguate the variable name. Without that, the parser would interpret the variable name as `$vowelst`, which either causes a compile-time error about an unknown variable or interpolates the contents of an existing `$vowelst` into the regex.

If the characters in your character set form a contiguous range, you can use the hyphen character (`-`) as a shortcut to express that range.

```
my $letters_only = qr/[a-zA-Z]/;
```

Move the hyphen character to the start or end of the class to include it in the class:

```
my $interesting_punctuation = qr/[-!?]/;
```

... or escape it:

```
my $line_characters = qr/[|=\-_]/;
```

Just as the word and digit class metacharacters (`\w` and `\d`) have negations, so too you can negate a character class. Use the caret (^) as the first element of the character class to mean "anything *except* these characters":

```
my $not_a_vowel = qr/[^aeiou]/;
```

Use a caret anywhere but this position to make it a member of the character class. To include a hyphen in a negated character class, place it after the caret or at the end of the class, or escape it.

Capturing

It's often useful to match part of a string and use it later; perhaps you want to extract an address or an American telephone number from a string:

```
my $area_code    = qr/\(\d{3}\)/;
my $local_number = qr/\d{3}-?\d{4}/;
my $phone_number = qr/$area_code\s?$local_number/;
```

Parentheses in regular expressions are metacharacters; `$area_code` escapes them.

Named Captures

Given a string, `$contact_info`, which contains contact information, you can apply the `$phone_number` regular expression and *capture* any matches into a variable with *named captures*:

```
if ($contact_info =~ /(?<phone>$phone_number)/)
{
    say "Found a number $+{phone}";
}
```

The capturing construct can look like a big wad of punctuation, but it's fairly simple when you can recognize as a single chunk:

```
(?<capture name> ... )
```

The parentheses enclose the entire capture. The `?< name >` construct must follow the left parenthesis. It provides a name for the capture buffer. The rest of the construct within the parentheses is a regular expression. If and when the regex matches this fragment, Perl stores the captured portion of the string in the special variable `%+`: a hash where the key is the name of the capture buffer and the value is the portion of the string which matched the buffer's regex.

Parentheses are special to Perl 5 regular expressions; by default they perform the same grouping behavior as parentheses do in regular Perl code. They also enclose one or more atoms to capture whatever portion of the matched string they match. To use literal parentheses in a regular expression, you must preface them with a backslash, just as in the `$area_code` variable.

Anonymous Captures

Named captures are new in Perl 5.10, but captures have existed in Perl for many years. You may encounter *anonymous captures* as well:

```
if ($contact_info =~ /($phone_number)/)
{
    say "Found a number $1";
}
```

The parentheses enclose the fragment to capture, but there is no regex directive giving the *name* of the capture. Instead, Perl stores the captured substring in a series of magic variables starting with `$1` and continuing for as many capture groups are present in the regex. The *first* matching capture that Perl finds goes into `$1`, the second into `$2`, and so on. Capture counts start at the *opening* parenthesis of the capture; thus the first left parenthesis begins the capture into `$1`, the second into `$2`, and so on.

While the syntax for named captures is longer than for anonymous captures, it provides additional clarity. You do not have to count the number of opening parentheses to figure out whether a particular capture is `$4` or `$5`, and composing regexes from smaller regexes is much easier, as they're less sensitive to changes in position or the presence or absence of capturing in individual atoms.

Name collisions are still possible with named captures, though that's less frequent than number collisions with anonymous captures. Consider avoiding the use of captures in regex fragments; save it for top-level regexes.

Named captures are less frustrating when you evaluate a match in list context:

```
if (my ($number) = $contact_info =~ /($phone_number)/)
{
    say "Found a number $number";
}
```

Perl will assign to the lvalues in order of the captures.

Grouping and Alternation

Previous examples have all applied quantifiers to simple atoms. They can also apply to more complex subpatterns as a whole:

```
my $pork  = qr/pork/;
my $beans = qr/beans/;

like( 'pork and beans', qr/\A$pork?.*?$beans/,
     'maybe pork, definitely beans' );
```

If you expand the regex manually, the results may surprise you:

```
like( 'pork and beans', qr/\Apork?.*?beans/,
     'maybe pork, definitely beans' );
```

This still matches, but consider a more specific pattern:

```
my $pork  = qr/pork/;
my $and   = qr/and/;
my $beans = qr/beans/;

like( 'pork and beans', qr/\A$pork? $and? $beans/,
    'maybe pork, maybe and, definitely beans' );
```

Some regexes need to match one thing or another. Use the *alternation* metacharacter (`|`) to do so:

```
my $rice  = qr/rice/;
my $beans = qr/beans/;

like( 'rice',  qr/$rice|$beans/, 'Found some rice'  );
like( 'beans', qr/$rice|$beans/, 'Found some beans' );
```

The alternation metacharacter indicates that either preceding fragment may match. Be careful about what you interpret as a regex fragment, however:

```
like(   'rice',  qr/rice|beans/, 'Found some rice'  );
like(   'beans', qr/rice|beans/, 'Found some beans' );
unlike( 'ricb',  qr/rice|beans/, 'Found some weird hybrid' );
```

It's possible to interpret the pattern `rice|beans` as meaning `ric`, followed by either `e` or `b`, followed by `eans`--but that's incorrect. Alternations always include the *entire* fragment to the nearest regex delimiter, whether the start or end of the pattern, an enclosing parenthesis, another alternation character, or a square bracket.

To reduce confusion, use named fragments in variables (`$rice|$beans`) or grouping alternation candidates in *non-capturing groups*:

```
my $starches = qr/(?:pasta|potatoes|rice)/;
```

The `(?:)` sequence groups a series of atoms but suppresses capturing behavior. In this case, it groups three alternatives.

If you print a compiled regular expression, you'll see that its stringification includes an enclosing non-capturing group; `qr/rice|beans/` stringifies as `(?-xism:rice|beans)`.

Other Escape Sequences

Perl interprets several characters in regular expressions as *metacharacters*, which represent something different than their literal characters. Square brackets always denote a character class and parentheses group and optionally capture pattern fragments.

To a *literal* instance of a metacharacter, *escape* it with a backslash (`\`). Thus `\(` refers to a single left parenthesis and `\]` refers to a single right square bracket. `\.` refers to a literal period character instead of the "match anything but an explicit newline character" atom.

Other useful metacharacters that often need escaping are the pipe character (`|`) and the dollar sign (`$`). Don't forget about the quantifiers either: `*`, `+`, and `?` also qualify.

To avoid escaping everything (and worrying about forgetting to escape interpolated values), use the *metacharacter disabling characters*. The `\Q` metacharacter disables metacharacter processing until it reaches the `\E` sequence. This is especially useful when taking match text from a source you don't control when writing the program:

```
my ($text, $literal_text) = @_;

return $text =~ /\Q$literal_text\E/;
```

The `$literal_text` argument can contain anything--the string `** ALERT **`, for example. With `\Q` and `\E`, Perl will not interpret the zero-or-more quantifier as a quantifier. Instead, it will parse the regex as `\*\* ALERT \*\*` and attempt to match literal asterisk characters.

Be cautious when processing regular expressions from untrusted user input. It's possible to craft a malicious regular expression which can perform an effective denial-of-service attack against your program.

Assertions

The regex anchors (`\A` and `\Z`) are a form of *regex assertion*, which requires that a condition is present but doesn't actually match a character in the string. That is, the regex `qr/\A/` will *always* match, no matter what the string contains. The metacharacters `\b` and `\B` are also assertions.

Zero-width assertions match a *pattern*, not just a condition in the string. Most importantly, they do not consume the portion of the pattern that they match. For example, to find a cat on its own, you might use a word boundary assertion:

```
my $just_a_cat = qr/cat\b/;
```

... but if you want to find a non-disastrous feline, you might use a *zero-width negative look-ahead assertion*:

```
my $safe_feline = qr/cat(?!astrophe)/;
```

The construct `(?!...)` matches the phrase `cat` only if the phrase `astrophe` does not immediately follow.

The *zero-width positive look-ahead assertion*:

```
my $disastrous_feline = qr/cat(?=astrophe)/;
```

... matches the phrase `cat` only if the phrase `astrophe` immediately follows. This may seem useless, as a normal regular expression can accomplish the same thing, but consider if you want to find all non-catastrophic words in the dictionary which start with `cat`. One possibility is:

```
my $disastrous_feline = qr/cat(?!astrophe)/;

while (<$words>)
{
    chomp;
    next unless /\A(?<some_cat>$disastrous_feline.*)\Z/;
    say "Found a non-catastrophe '$+{some_cat}'";
}
```

Because the assertion is zero-width, it consumes none of the source string. Thus the anchored `.*\Z` pattern fragment must be present; otherwise the capture would only capture the `cat` portion of the source string.

Regex Modifiers

The regular expression operators allow several modifiers to change the behavior of matches. These modifiers appear at the end of the match, substitution, and `qr//` operators. For example, to enable case-insensitive matching:

```
my $pet = 'CaMeLiA';

like( $pet, qr/Camelia/,  'You have a nice butterfly there'       );
like( $pet, qr/Camelia/i, 'Your butterfly has a broken shift key' );
```

The first `like()` will fail, because the strings contain different letters. The second `like()` will pass, because the `/i` modifier causes the regex to ignore case distinctions. `M` and `m` are equivalent in the second regex due to the modifier.

You may also embed regex modifiers within a pattern:

```
my $find_a_cat = qr/(?<feline>(?i)cat)/;
```

The `(?i)` syntax enables case-insensitive matching only for its enclosing group: in this case, the entire `feline` capture group. You may use multiple modifiers with this form (provided they make sense for a portion of a pattern). You may also disable specific modifiers by preceding them with `-`:

```
my $find_a_rational = qr/(?<number>(?-i)Rat/;
```

How do I rephrase this paragraph?

Other modifiers include `/m`, the multiline operator. When this is active, the `^` and `$` anchors match at any start of line or end of line within the string, depending on your newline settings.

The `/s` modifier treats the source string as a single line such that the `.` metacharacter matches the newline character. Damian Conway suggests a mnemonic that `/m` modifies the behavior of *multiple* regex metacharacters, while `/s` modifies the behavior of a *single* regex metacharacter.

The `/x` modifier allows you to embed additional whitespace and comments within patterns without changing their meaning. With this modifier in effect, the regex engine treats whitespace and the comment character (#) and everything following as comments; it ignores them. This allows you to write much more readable regular expressions:

```
my $attr_re = qr{
    ^                    # start of line

    # miscellany
    (?:
      [;\n\s]*           # blank spaces and spurious semicolons
      (?:/\*.*?\*/)?     # C comments
    )*

    # attribute marker
    ATTR

    # type
    \s+
    (   U?INTVAL
      | FLOATVAL
      | STRING\s+\*
      | PMC\s+\*
      | \w*
    )
}x;
```

This regex isn't *simple*, but comments and whitespace improve its readability. Even if you compose regexes together from compiled fragments, the `/x` modifier can still improve your code.

The /g modifier performs a regex globally throughout a string. This makes sense when used with a substitution:

```
# appease the Mitchell estate
my $contents = slurp( $file );
$contents    =~ s/Scarlett O'Hara/Mauve Midway/g;
```

When used with a match--not a substitution--the \G metacharacter allows you to process a string within a loop one chunk at a time. \G matches at the position where the most recent match ended. To process a poorly-encoded file full of American numbers in logical chunks, you might write:

```
while ($contents =~ /\G(\w{3})(\w{3})(\w{4})/g)
{
    push @numbers, "($1) $2-$3";
}
```

Be aware that the \G anchor will take up at the last point in the string where the previous iteration of the match occurred. If the previous match ended with a greedy match such as .*, the next match will have less available string to match. The use of lookahead assertions can become very important here, as they do not consume the available string to match.

The /e modifier allows you to write arbitrary Perl 5 code on the right side of a substitution operation. If the match succeeds, the regex engine will run the code, using its return value as the substitution value. The earlier global substitution example could be more robust about replacing some or all of an unfortunate protagonist's name with:

```
# appease the Mitchell estate
my $contents = slurp( $file );
$contents    =~ s{Scarlett( O'Hara)?}
                 { 'Mauve' . defined $1 ? ' Midway' : '' }ge;
```

You may add as many /e modifiers as you like to a substitution. Each additional occurrence of this modifier will perform another evaluation of the result of the expression, though only Perl golfers tend to use /ee or anything more complex.