Powerful Tools for Perl and Other Tools

Mastering

Regular Expressions

Jeffrey E. F. Friedl

Mastering Regular Expressions (mini version)

Jeffrey E. F. Friedl

1st Edition January 1997

1-56592-257-3, 366 pages

Regular expressions, a powerful tool for manipulating text and data, are found in scripting languages, editors, programming environments, and specialized tools. In this book, author Jeffrey Friedl leads you through the steps of crafting a regular expression that gets the job done. He examines a variety of tools and uses them in an extensive array of examples, with a major focus on Perl.

Release Team[oR] 2001

Preface

This book is about a powerful tool called "regular expressions."

Here, you will learn how to use regular expressions to solve problems and get the most out of tools that provide them. Not only that, but much more: this book is about *mastering* regular expressions.

If you use a computer, you can benefit from regular expressions all the time (even if you don't realize it). When accessing World Wide Web search engines, with your editor, word processor, configuration scripts, and system tools, regular expressions are often provided as "power user" options. Languages such as Awk, Elisp, Expect, Perl, Python, and Tcl have regular-expression support built in (regular expressions are the very heart of many programs written in these languages), and regular-expression libraries are available for most other languages. For example, quite soon after Java became available, a regular-expression library was built and made freely available on the Web. Regular expressions are found in editors and programming environments such as *vi,* Delphi, Emacs, Brief, Visual C++, Nisus Writer, and many, many more. Regular expressions are very popular.

There's a good reason that regular expressions are found in so many diverse applications: they are extremely powerful. At a low level, a regular expression describes a chunk of text. You might use it to verify a user's input, or perhaps to sift through large amounts of data. On a higher level, regular expressions allow you to master your data. Control it. Put it to work for you. To master regular expressions is to master your data.

Why I Wrote This Book

You might think that with their wide availability, general popularity, and unparalleled power, regular expressions would be employed to their fullest, wherever found. You might also think that they would be well documented, with introductory tutorials for the novice just starting out, and advanced manuals for the expert desiring that little extra edge.

Sadly, that hasn't been the case. Regular-expression documentation is certainly plentiful, and has been available for a long time. (I read my first regular-expression-related manual back in 1981.) The problem, it seems, is that the documentation has traditionally centered on the "low-level view" that I mentioned a moment ago. You can talk all you want about how paints adhere to canvas, and the science of how colors blend, but this won't make you a great painter. With painting, as with any art, you must touch on the human aspect to really make a statement. Regular expressions, composed of a mixture of symbols and text, might seem to be a cold, scientific enterprise, but I firmly believe they are very much creatures of the right half of the brain. They can be an outlet for creativity, for cunningly brilliant programming, and for the elegant solution.

I'm not talented at anything that most people would call art. I go to karaoke bars in Kyoto a lot, but I make up for the lack of talent simply by being loud. I do, however, feel very artistic when I can devise an elegant solution to a tough problem. In much of my work, regular expressions are often instrumental in developing those elegant solutions. Because it's one of the few outlets for the artist in me, I have developed somewhat of a passion for regular expressions. It is my goal in writing this book to share some of that passion.

Intended Audience

This book will interest anyone who has an opportunity to use regular expressions. In particular, if you don't yet understand the power that regular expressions can provide, you should benefit greatly as a whole new world is opened up to you. Many of the popular cross-platform utilities and languages that are featured in this book are freely available for MacOS, DOS/Windows, Unix, VMS, and more. Appendix A has some pointers on how to obtain many of them.

Anyone who uses GNU Emacs or vi, or programs in Perl, Tcl, Python, or Awk, should find a gold mine of detail, hints, tips, and understanding that can be put to immediate use. The detail and thoroughness is simply not found anywhere else. Regular expressions are an idea—one that is implemented in various ways by various utilities (many, many more than are specifically presented in this book). If you master the general concept of regular expressions, it's a short step to mastering a particular implementation. This book concentrates on that idea, so most of the knowledge presented here transcend the utilities used in the examples.

1.0 Introduction to regular expressions

1.1 What are regular expressions used for

Here comes the scenario: Your boss in the documentation department wants a tool to check double words e.g. "this this", a common problem with documents subject to heavy editing. Your job is to create a solution that will:

- Accept any number of files to check, report each line of each file that has double words.
- Work across lines, find word even in separate lines.
- Find double words in spite of the capitalization differences "The", "the", as well as allowing whitespace in between the words.
- Find doubled words that might even be separated by HTML tags. "it s very <I>very</I> important"

That is not an easy task! If you use such a tool for existing documents, you may surprisingly find similar spelling mistakes in various sources. There are many programming languages one could use to solve the problem, but one with regular expression support can make the job substantially easier.

Regular Expressions are the key to powerful, flexible, and efficient text processing. Regexps themselves, with a general pattern notation, almost like a mini programming language, allow you to describe and parse text. With additional support provided by the particular tool being used, regular expressions can add, remove, isolate, and generally fold, spindle all kinds of text and data. It might be as simple as text editor's search command or as powerful as a full text processing language. You have to start thinking in means of *Regexps*, and not the the way you have used to with your previous programming languages, because only then you are taking the full magnitude of their power.

The host language (Perl, Python, Emacs Lisp) provides the peripheral processing support, but the real power comes from regular expressions. Using the Regexps right will make it possible to identify the text you want and bypass the portions that you are not interested in.

1.2 Solving real problems

Checking text in files

As a simple example, suppose you need to check slew of files (70-150) to confirm that each file contained `SetSize` exactly as often as contained `ResetSize`. To complicate matters, you should disregard the capitalization and accept `SETSIZE`. The total count of lines in those files could easily end up to 30000 or more and checking them by hand would give you a headache. Even using normal "find this word" with text processor would have been truly arduous, what with all the files and all the possible capitalizations. Regexps come to rescue. Typing just a single short command your make the work in seconds and confirm what you want to know.

```
% perl -One "print qq($ARGV\n) if s/ResetSize//ig != s/setSize//ig" *
```

Summary of Email mailbox

If you wanted to create a summary of the messages in your mailbox, it would be tedious to read all your 1000 mails and store the important lines to a separate lines by and (like `From:` and `Subject:`). What if you were behind dial-up? The on-line time spend in making such summary easily eats your pocket if you had to do it multiple times. In addition, you couldn't do that to some other person, because you would see the contents of his mailbox. Regexps come to rescue again. A very simple command could display summary of those two lines immediately.

```
% perl -ne "print if /*(From|To):/" ~/Mail/*
```

What if someone asked about that summary? It would be non-needed to send the 5000 line results, when you could send that little one-liner to the friend and ask him to run it for his mailbox.

1.3 Regular expressions as a language

Unless you have had some experience with regular expressions, you wouldn't understand the above commands. There really is no special magic here, just set of rules that must be digested. once you learn how to hide a coin behind your hand, you know there is not much magic in it, just lot of practice and learning new skills. Like a foreign language, it will start stopping sound like "gibberish" after a while.

1.4 The filename analogy

If you have only experience on the Win32/Windows environment, you have a grasp that following refers to multiple files:

```
*.txt
```

With such filename patters like this (called file *globs*) there are few characters that have a special meaning.

```
*   => means: "MATCH ANYTHING"
?   => means: "MATCH ONE CHARACTER"
```

The complete example above will be parsed as

```
*.txt
|||
|||
||Match three characters in order "t" "x" "t"
|Match a "dot"
Match anything [A special character]
```

And the whole patters is thus read as "Match files that start with anything and end with `.txt`"

Most systems provide a few additional special characters, but in general these filename patterns are limited in expressive power. This is not much of a shortcoming because the scope of the problem (to provide convenient ways to specify group of files) is limited to filenames.

On the other hand, dealing with general text is a much larger problem. Prose and poetry, program listings, reports, lyrics, HTML, articles, code tables, word lists ...you name it. over the years a generalized pattern language has developed which is powerful and expressive for wide variety of uses. Each program implements and uses them differently, but in general this powerful pattern language and the patterns themselves are called *Regular Expressions*.

1.5 The language analogy

Full regular expressions are composed of two types of characters. The *special* characters (like "*" in files) are called **meta-characters**, while everything else are called **literal** or normal text characters. What sets regular expressions apart from the filename patterns is the scope of power their meta-characters provide. Filename patterns provide limited patterns, but regular expression "Language" provides rich and expressive power to advanced users.

1.6 The regular expression from in mind

Complete regular expressions are built up from small building block units. Each building block is in itself quite simple, but since they can be combined in an infinite number of ways, knowing how to combine them to achieve a particular goal takes some experience. While some regular expressions may seem silly, they do really represent the kind of tasks that are done in real - you just might not realize it yet.

Just as there are difference between *playing* musical piece well and *making music*, there is a difference between understanding regular expressions and *really understanding* them.

1.7 Searching text files

Finding text is the simples uses of regular expressions - many text editors and word processors allow you to search a document using some kind of pattern matching. Let's return to the original example of finding some relevant lines from a mailbox, we study it in detail:

```
                Perl command part
                |
                |        Regular expression part
                +------+ +-----------+
% perl -ne "print if /*(From|To):/" file.txt
| |     |  |                        |
| |     |  |                        Read from file
| |     |  Start of the code Win32/Unix. Unix also accepts single(')
| |     |
| |     Give some options
| |     -n  Do not print unless requested
| |     -e  Read expression or code immediately
| |
| Call program "perl"
|
command shell's prompt. In Unix % or $ and in Win32 typically >
```

Even more simple example would be searching every line containing word like "cat":

```
% perl -ne "print if /cat/" Mail/*.*
```

But things are not that simple, because how do you know which word is plain "cat", you must consider how "catalog", "caterpillar", "vacation" differs semantically from the animal "cat". The matched results do not show what was really matched and made the line selected, the lines are just printed. The key point is that regular expressions searching is not done a "word" basis, but in general only *character* basis without any knowledge about e.g. English language syntax.

1.8 Grep, egrep, fgrep, perl, say what?

There is a family of products that started the era of regular expressions in Unix tools know as `grep(1)`, `egrep(1)`, `frep(1)`, `sed(1)` and `awk(1)`. The first of all was *grep*, soon followed by *extended grep* `egrep(1)`, which offered more patterns in regular expression syntax. The final evolution is `perl(1)` which enhanced the regular expressions way further that could be imaginable. Whenever you nowadays talk about regular expression, the foundation of new inventions lies in Perl language. The Unix man page about regular expression it in ´regexp(5)'.

2.0 Character classes

You must THINK that the character class notation is something of its own regular expression sub language. It has its OWN rules that are not the same as outside of character classes.

2.1 Matching list of characters

What if you want to search all colors of "grey" but also spelled like "gray" with a one character difference. You can define a list of characters to match, a *character class*. This regexp reads: "Find character `g` followed by `e` and try next character with `e` OR `a` and finally character `y` is required.".

```
/ge[ea]y/
```

As another example, suppose you want to allow capitalization of word's first letter. Remember that this still matches lines that contain `smith` or `Smith` embedded in another word as `blacksmith`. This issue is usually the source of the problem among new users.

```
/[Ss]mith/
```

You can list in the class as many characters as you like. Notice that you can list the items in any order:

```
/[0123456]/
/[6543210]/
```

Which might be a good set of choices to find HTML heading from the page with: <H1> <H2> .. <H6> (That is the maximum according to HTML 4.x specification. Refer to http://www.w3c.org/)

```
/<H[0123456]>/
```

There are few rules concerning the character class: Not all characters inside it are pure *literals*, taken "as is". A dash(-) indicates a range of characters, and here is identical example:

```
/<H[0-6]>/
```

One thing to memorize is, that regular expressions are case ***sensitive***. It is different to match "a" or "A", like if you would construct a set of alphabets for regular 7bit English text. (Different countries have different sets of characters, but that is a whole separate issue)

```
/[a-z]/             Ehm...
/[a-zA-Z]/          Maybe this is what you wanted?
```

Remember that the dash(-) applies only to a character class, in here it is just a regular dash:

```
/a-z/               Match character "a", character "-", character "z"
```

Multiple dashes can be used inside a class in any order, but the dash-order must follow the ASCII-table sequence, and not run backwards:

```
/[a-zA-Z0-9]/       ok
/[A-Za-zA-Z0-9]/    Repetitive, but no problem
/[0-9A-Za-z]/       ok
/[9-0]/             Oops, you can't say it like "backwards"
```

Exclamation and other special characters are just characters to match:

```
/[!?:,.=]/
```

Or pick your own personal set of characters. This does **not** match word "help":

```
/[help]/
```

2.2 Negated character classes

It is easy to write what charters you *want* to include, but what if you would like to match everything *except* few characters? It would be unpractical to list all the possible character and then leave out only some of them:

```
/[ZXCVBNMASDFGHJKLQWERTYUIO ...]/
```

A special character, *inside* character class tells "not to include". (this same character has different meaning outside of the class, where it means "beginning of line"):

```
  The NOT operator
  |
/[^0-9]/        Match everything, but numbers.
```

NOTE: The end-of-line marker is different is various OS platforms. The above regular expression will match a line containing only plain numbers, because there is embedded end-of-line marker at the end: In Unix "1234567\n", in win32 "1234567\r\n" and in Mac "1234567\r"

Why would following regular expression list items below?

```
% perl -ne "print if /q[^u\r\n]/" file.txt
Iraqi
Iraqian
miqra
qasida
qintar
qoph
zaqqum
```

Why didn't it list words like

```
Quantas
Iraq
```

2.3 Character class and special characters

The brackets ([])

As we have learned earlier, some of the characters in character class are special, these include range(-) and negation(^), but you must remember that the characters continuing the class itself must also be special:] and [. So, what happened if you really need to match any of these characters? Suppose you have text:

```
See [ref] on page 55.
```

And you need to find all texts that are surrounded within the brackets []. You must write like this, although it looks funny. It works, because an "empty set" is not a valid character class so in here there is not really two "empty character class sets":

```
 start of class
 |  End of class
 |  |
/[][]/              => also /[]abcdef0-9]]/
  ||
  |character "]"
  character "["
```

***Rule**:] can be anywhere in the class and [must be at the beginning of class*

The dash (-)

If the dash operator is used to delimit a range in a character class we have problem what top do with it if we want to match person names like "Leary-Johnson". The solution can be found if we remember that dash need a FROM-TO, but if we omit either one, and write FROM- or -TO, then the special meaning is canceled.

```
/[-a-zA-Z]/   OR    /[a-zA-Z-]/
```

***Rule**: dash(-) character is taken literally, when it is put either to the beginning or to the end of character class*

The caret (^)

We still have one character that has a special meaning, the negation operator, that excludes characters from the set. We can solve the conflict, to take "^" literally, as plain character when we move it out from its special position: at the beginning of the class

```
/[^abc]/    Means: all except "abc" characters
/[abc^]/    Caret has no special meaning any more. Matches
            characters "a" "b" "c" and "^"
/[a^bc]     Works too. "^" is taken literally.
```

***Rule**: caret(^) loses its special meaning, when it is not the first character in the class.*

How to put all together

Huh, do we dare to combine all these exceptions in one regular expression that would say, "I want these character: ^, - ,] and [". It might be impossible or at least time consuming task if you didn't know the rules of these characters. With trial and error you could eventually come up with right solution, but you would never understand fully why it works. Here is the answer. Can you think of more possible choices?

```
/[][^-]/
```

And now the final master thesis question: how do you reverse the question, "I want to match everything, **except** characters ^, - ,] and [" ??

2.4 POSIX locales and character classes

*POSIX, short for **Portable Operating System Interface**, is a standard ensuring portability across operating systems. Within this ambitious standard are specifications for regular expressions and many of the traditional Unix tools use them.*

One feature of the POSIX standard is the notion of *locale*, setting which describe language and cultural conventions such as the display of dates, times and monetary values, the interpretation of characters in the active encoding, and so on. Locales aim at allowing programs to be internationalized. It is not regexp-specific concept, although it can affect regular expression use. For example when working with *Latin-1* (ISO-8859-1) encoding, the character "a" has many different meanings in different languages (think adding ' at top of "a"). Perl defines `\w` to be word and as regexps `[a-zA-Z0-9_]`, but this in not the whole story, since perl respects the `use locale` directive in programs and thus allows enlarging the A-Z character range.

POSIX collating sequence

A locale can define collating sequences to describe how to treat certain characters or sets of characters, for sorting. For example Spanish `ll` as in *tortilla' traditionally sorts as if it were on logical character between `l` and `m`. These rules might be manifested in collating sequences named *span-ll* and *eszet* for German `ss`. As with *span-ll*, a collating sequence can define multi-character sequences that should be taken as **single** character. This means that the dot(.) in regular expression /torti.a/ matches "tortilla".

POSIX character class

A POSIX character class is one of several special meta sequences for use within a POSIX bracket expression. An example is `[:lower:]` which represents any lowercase letter within the current locale. For normal English that would be `[a-z]`. The exact list of POSIX character classes is locale independent, but the following are usually supported (appeared 2000-06 in perl 5.6). See more from the [perlre] manual page.

```
[:class:]         GENERIC POSIX SYNTAX, replace "class" with names below
[:^class:]        PERL EXTENSION, negated class
[:alpha:]         alphabetic characters
[:alnum:]         alphabetic characters and numeric characters
[:ascii:]
[:cntrl:]         control characters
[:digit:] \d      digits
[:graph:]         non-blank (no spaces or control characters)
[:lower:]         lowercase alphabetics
[:print:]         like "graph" but includes space
[:punct:]         punctuation characters
[:space:] \s      all whitespace characters
[:upper:]         uppercase alphabetics
[:word:]  \w
[:xdigit:]        any hexadecimal digit, [0-0a-fA-F]
```

Here is an example how to use the basic regular expression syntax and the roughly equivalent POSIX syntax: They match a word that is started with uppercase letter.

```
/[A-Z][a-z]+/
/[[:upper:]][[:lower:]]+/
```

POSIX character equivalents

Some locales define *character equivalents* to indicate that certain characters should be considered identical for sorting. The equivalent characters are lister with the [=...=] notation. For example to match Scandinavian "a" like characters, you could use `[=a=]`. Perl 5.6 2000-06 recognizes this syntax, but does not support it and according to [perlre] manual page: "The POSIX character classes [.cc.] and [=cc=] are recognized but not supported and trying to use them will cause an error: Character class syntax [= =] is reserved for future extensions"

3.0 Regular expressions syntax

3.1 Marking start and end

A good start to regular expression is to discuss how regular expression define beginning-of-line (^) and end-of-line ($). Both have special meta-characters that mark the position correctly. As we have seen ´cat' will be batched everywhere in the line, but we may want to *anchor* the match to the start of the line. Get into habit interpreting the regular expressions in a rather literal way, don't loosen up your mind or you will read the regular expression wrongly. [IMPORTANT] The ^ and $ are particular in that they match a *position*. They do match any actual characters themselves.

```
/^cat/
```

WRONG: *matches line with "cat" at the beginning* ***RIGHT:*** *Matches at the beginning of line, FOLLOWED by character "c" and character "a" and character "t".*

How would you read following expressions:

```
/^cat$/
/^$/
/$^/
/^/
/$/
//
/cat^/
/$cat/      # This is not "end-of-line" + "cat", but a variable
```

3.2 Matching any character

The *meta-character* dot(.) is shorthand for a pattern that matches an character, **except** newline. For example, if you want to search regular ISO 8601 YYYY-MM-DD dates like 2000-06-01, 2000/06/01 or 2000.06.01, you could construct the regular expression using the character classes or just allow any character in between:

```
          Note, the "/" must be escaped with \/ because
          it would otherwise terminate Perl Regexp / ..... /
          |
/2000[.\/-]06[.\/-]/    This is more accurate
/2000.06.01/            The "." accepts anything in place
```

Notice the different semantics again in the above regexps: The dot(.) **is not** a meta-character inside the character class, like in the first example. It only has the special meaning if it is used alone, outside of the class like in the second example. [IMPORTANT] Consider using dot(.) only if you know that the data is in consistent format, because it may cause trouble and match lines that you didn't want to, like lottery numbers. The first regexp is the most safest to use compared to second, which will match:

```
Lottery this week: 12 2000106 01 20
                       ====.==.==
```

3.3 Alternation and grouping

When you are inclined to choose from several possibilities, you mean word **OR**. The regular expression atom for it is |, like in programming languages. When used in regexps, the parts of the regular expressions are called *alternations*.

```
 Try "Bob" first. If not found, then try "Joe" ...
 |
/Bob|Joe|Mike|Helen/
 ===
 This part is tried completely before moving to next
 alternation after "|". The alternations are tried in order
 from left to right (but refer to DFA and NFA engines)
```

Looking back to color matching with gr[ea]y, it could have been written using the alternation

```
/grey|gray/         Both of the regexps are effectively
/gr[ea]y/           ..the same, but gr[ea]y is faster.
```

The alternation can't be written like this, which would mean different thing, because both sides, around `|`, are tried as they are read.

```
 Choice one: "gre"
 |
/gre|ay/
     |
     OR choice two: "ay"
```

In order to write the regexp this way, you need to say which characters belong together and restrict the range of effect, in this case, the alternation only to "e" and "a". You need *grouping* `()` like this:

```
/gr(e|a)y/          Again the same as /gr[ea]y/
```

This is not a substitute for character class. To match range of numbers, you would need to write a long alternation, compared to elegant character class notation:

```
/<H(1|2|3|4|5|6)>/      => better is to use /<H[1-6]>/
```

What would you think that the following regular expression would match?

```
"Book is here"
"A novel is interesting"
"The pocket book si is A6"
```

Fix the regular expression below to match the books correctly:

```
/(an|the) (book|novel|pocket)/
```

3.4 Alternation and anchors

The alternation is usually put inside parenthesis, and you should use it like that too if you are even a bit insure. Consider what would happen if we added anchors, beginning-of-line and end-of-line to the regexp:

```
/^grey|gray/
/grey|gray$/
```

They are **not** interpreted by the regular expression engine as "find at the beginning of line choice `grey` OR choice `gray`". The key point to understanding is that each part is a *complete* self contained subexpression and read by the engine as two separate regexps:

```
First try this:  /^grey/
Then try this:   /gray/
```

As you see, the effect of the beginning-of-line anchor only affects the first, not the second OR choice. You need to group the choices, so that the anchor effect will touch both.

Notice that the character class and the alteration does have a equivalence, but it is important to remember that character classes are meant to match only **one** character, whereas alteration is meant to separate **many** choices. If we try to convert the following regular expression to alteration method, which one would you think is correct: A or B? See [Friedl] page 42.

```
/[ \t]*/
/ *|\t*/            Is it like this (A) ?
/( |\t)*/           Is it like this (B) ?
```

Exercise: Write smallest regular expression possible to match lines below:

```
First Street
1st street
```

Exercise: Write smallest regular expression possible to match lines below:

```
Jul 4
July 4th
Jul  fourth
```

Exercise: What is the correct answer from the list below?

```
/^(grey|gray)/
/(grey|gray)$/
/^(grey|gray)$/
```

An email or a news article message usually has the following format. Suppose we want to extract few key information from the headers of it, like From, Subject and Date fields.

```
Message-ID: <39A962F5.E0C4E75C@NOSPAM.yahoo.com>
Date: Sun, 27 Aug 2000 20:50:29 +0200
From: gatto@linuxfreak.com (Matthew Gatto)
X-Mailer: Mozilla 4.05 [en] (Win95; I)
MIME-Version: 1.0
Newsgroups: comp.emacs
Subject: writing fancy documents?
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
NNTP-Posting-Host: upp5.hhs.se
X-Trace: 27 Aug 2000 20:48:47 +0200, upp5.hhs.se
Organization: Stockholm School of Economics
Lines: 12
Path: newsfeed1.funet.fi!news.hhs.se!upp5.hhs.se
I'm writing some formal letters, and I'd like to do it with
emacs. I need to be able to adjust printing fonts, text size,
etc. Is that possible? Also, is there some way to do footnotes
in emacs?
```

Would you be able to use regexp that just gives a list of choice, like this?

```
/^From|Subject|Date:/
```

No, you forgot to include the all so important parenthesis, because the the effect of anchor ^ and character ":" will be applied to all choices. Can you write equivalent regular expression **without** using the parenthesis?

```
/^(From|Subject|Date):/
```

Exercise: What are the alternatives in below?

```
/a|b/
/[a|b]/
/[a]|[b]/
```

Exercise: How would you match subjects that are replied or forwarded multiple times?

```
Subject: Hi There!
Subject: Re: Hi There!
Subject: Re: Re: Hi There!
Subject: fw: Hi There!
Subject: fw: fw: Hi There!
Subject: fw: Re: Hi There!
Subject: Re: fw: :Re: Hi There!
```

Exercise: Which of the above choices is equivalent to regular expression /^this|that$/ ?

```
/^(this|that)$/
/(^this)|(that$)/
/^(this)|(that)$/
```

FINALLY, BE VERY CAREFUL NOT TO LEAVE THE ALTERATION LAST. Try and see what would this regular expression match and think about it for a moment.

```
/(this|that|)/
           ^
           ^
           Oops !!
```

3.5 Word boundaries

A common problem is that regular expressions that matches the word you want can also match where the "word" is embedded within a larger word. In regular expression syntax, there usually is a meta-character notation to mark a word boundary. The meta-character used to mark the **word boundary** is not standard:

```
\<   and  \>        egrep(1), possibly new grep(1) too
\\<  and  \\>       Emacs, all \ character must be doubled
\b   and  \b        perl(1)
```

Here are few examples. In Emacs, you would run the command over the current buffer: the regular expression search command is named `re-search-forward`.

```
% perl -ne "print if /\bcat\b/" file.txt
% egrep "\<cat\>" file.txt
M-x re-search-forward RET <\cat\>
C-x ESC ESC                          << Look at the command
```

Where exactly is the word boundary? Regular expression does not know anything about the spoken or artificial language semantics and "words" in a sense that we have learned. We have to know the definition of a word from the perspective of regular expression engine:

```
[a-zA-Z0-9_]   A character belonging to this class is a Perl word
```

Let's see how the words are seen in real text according to the class above. [IMPORTANT] The word meta-character `\b` is like the anchor, and belongs to the set of *zero width assertions*, which means that it **never consumes** a character, it matches a position **before** or _after a word, effective in the middle of two words if needed:

```
 \b    \b         \b \b \b    \b  \b    \b        \b    \b  \b
 |     |          |  |  |     |   |     |         |     |   |
That solar-flare, 2.11.1999, was darn' fabulous! Don't you think?
   |     |    |          |   |     |         |    |    |      |
   \b    \b   \b         \b  \b    \b        \b   \b   \b     \b
```

Notice the location of words (\b), which are really not exactly over the characters itself, while they are marked so in this page (Hear the instructor to explain this in detail).

3.6 Quantifiers (basic, greedy)

*Remember that quantifiers affect the **immediately** previous regular expression element, whatever that be. If you group the regular expression, the quantifier will affect the **whole** group, like in /(grey|gray)*/ would repeat "greygrey" or "graygray" or any of these combinations like "greygray".*

If you want to repeat something, it is possible to just repeat the character class e.g. to match an three letter alphabet word:

```
/[a-z][a-z][a-z]/
```

Usually you don't know how long a word will be, so trying to match multiple choices by adding more alternations blurrs the clear regexp fast. (Off the record for the experienced regexp programmer, yes, you would change the order of the alternation in perl, because it uses NFA regular expression engine):

```
/[a-z]|[a-z][a-z]|[a-z][a-z][a-z]|  ....  /
 |     |          |
 |     |        OR maybe three characters ...
 |    Try two characters...
Try one character
```

There are three basic meta-characters that can repeat, one or as many times as needed. They are:

- ? Match 0 or 1 times [NOTE] Will always match
- * Match 0 or more [NOTE] Will always match
- + Match 1 or more

Pay attention to the notes above. If the regular expression is not required to match, as the zero count allows, it will flag "ok" and the regular expression is happy with it. It has succeeded. Consider following examples and their differences for a while:

```
/a/    /a+/    /a*/    /a?/
```

Let's take a real example from the HTML specification, which says that "whitespace are allowed inside tags". The term *whitespace* here refers to both **space** and the **tab** character, when it in Perl would mean a character class [\t\n\r\f]. Here are some examples. Refer to HTML 4.01 section 5.5 specification at http://www.w3c.org/

```
<H1>
<H1 >
<H1            >
<      H1>
< H1 >
```

The start * quantifier is suited for this kind of repeating, when there can be any number of whitespace including none:

```
/< *H[1-6] *>/
```

Exploring further, the HR, horizontal rule, element can contain a specifier SIZE and written like this, if we wanted 14 pixels thick line across the screen.

```
<HR SIZE=14>
```

This tag requires a bit more intelligence for crafting the regular expression, because there **must** be a separation between HR and SIZE, they are not allowed to read "<HR SIZE=14>". We use quantifier +, which allows any number of repetitions, but at least **1** must be present. There is still room for improvement in this regexp, can you make it even better?

```
/< *HR +SIZE=14 *>/
       |
       There must be preceding space. The + will require at least 1
```

The expression is still inflexible with respect to the is given in the tag. It only finds the is 14, when it should find HR elements with any size. We need a number that can be of varying size. A number is one digit or more digits. Notice the use of + quantifier in the number class:

```
/< *HR +SIZE=[0-9]+ *>/
```

All is well now, but now this regular expression won't find the basic <HR> any more, because it looks for SIZE keyword as well. To make it complete, we must remember what the HR specifications says: the SIZE part is *optional*, usually expressed in notation like this

```
<HR [SIZE=N]>       THIS IS NOT A REGEXP, BUT A NOTATION
    ========
    This part inside [] is marked as optional
```

3.7 Quantifiers (basic, additional)

In addition to basic quantifiers `?`, `*` and `+` that are part of every regular expression engine, there can also be more meta-syntaxes which offer more precise control of *how many exactly* is matched. I perl you will find following extra syntaxes:

- `{n}` exactly n times
- `{n,}` at least n times
- `{n,m}` at least n times, but no more than m times

A few examples, let's say that you want to match decimal number of varying size, like 1.1 or 1.12 or 12.12, but nothing else. You know how long the numbers are and you can use the extended quantifiers:

```
/[0-9]{1,2}\.[0-9]{1,2}/
           |
          Notice this, remember what dot(.) would have been !!
```

You see a little new construct in this regular expression, and escaped dot `\.` The dot in regular expression notation means "any character", but we don't want that, we really want the character "." that makes a decimal number. In order to prevent dot(.) meta-character, we must *escape*, thus disable, its special meaning when parsed by the regular expression engine. You can always write backslash(\) in front of character to take them *literally*.

3.8 Quantifiers (extended, non-greedy)

The normal quantifies are *greedy*, meaning that they get as much as they can from the regular expression But this is not always desirable. Consider trying to match all the text between the bold tags in HTML:

```
use English;
$ARG = "<B>Intel</B> and <B>AMD</B> are competitors.";
print  "Matched: $1\n"  if /<B>(.*)<\/B>/;
-->
Matched: Intel</B> and <B>AMD
```

As you can see, the match does not stop to the first /B, but continues until the last /B, that is, the match is greedy. To overcome this, perl has the *non-greedy* quantifier alternatives, where an additional question mark is added to the end of the basic quantifies:

- `??` Match **0** or 1 times [NOTE] Will always match
- `*?` Match 0 or as **little** as possible [NOTE] Will always match
- `+?` Match 1 or as **little** as possible
- `{n}?` exactly n times. This is identical to plain {n}
- `{n,}?` at least n times
- `{n,m}?` at least n times, but no more than m times

Now the regular expression can match only the shortest possible portions:

```
use English;
$ARG = "<B>Intel</B> and <B>AMD</B> are competitors.";
print  "Matched: $1\n"  while /<B>(.*?)<\/B>/g;
-->
Matched: Intel
Matched: AMD
```

3.9 Ignoring case

HTML tags can be written with mixed case capitalization, so <h3> and <H3> or <HR SIZE=14> and <Hr Size=14> are all legal. Modifying a simple tag is easy, /<[Hh][0-9]>/ but making a longer tag to ignore capitalization become more troublesome in plain regular expression means:

```
/< *[Hh][Rr] +[Ss][Ii][Zz][Ee]=14 *>/
```

Fortunately most programs that deal with regular expression can perform a match in *case insensitive* manner. This is not part of the regular expression language, but is a related useful feature that the tool or programming language provides:

```
% perl -ne    "print if /\bcat\b/" file.txt
% perl -ne    "print if /\b[Cc][Aa][Tt]\b/" file.txt
% perl -i -ne "print if /\bcat\b/" file.txt
         |
        This command line options means "ignore case"
% perl -ne    "print if /\bcat\b/i" file.txt
                                  |
                                 Regular expression modifier
                                 to ignore case
```

Perl also provides set of *modifiers* that you can apply to regular expression to achieve the same effect.

3.10 Parentheses and back references

Parentheses can *remember* text matched by the sub-expression they enclose. For example to remove double words like " the ", or " one " you could match the first one and refer to next with it. There is only one problem with simple matches: it would also find "the theory" and "one one-list". Referring to existing match is known as *backreferencing*.

```
/\bthe +the\b/          -- first try for double "the" search
/\b([a-zA-Z]+) \1\b/    -- Using the backreferencing \1
                |
               Special meta sequence to refer to grouped ( )
```

A more general solution to find double words might be

```
"Here is the the book"
"Here is the theory"
/\b(\w+)\b\s+\b\1\b/        See explanation of \w and \s later
 =========    =====
 find "word"  Is the word same?
```

You can have more than one set of parentheses, and you can refer to them in order \1 \2 \3 \4 .. \9 which is the maximum. The parentheses are numbered by counting open parentheses from left to right:

```
( ( ( ) ( ) ) ) ( )
1 2 3   4       5
```

NOTE: the syntax \1 \2 .. \9 is used only **inside** the regular expression. To the **outside**, the captured values are available from perl variables $1, $2 .. $N and there is no maximum limitation of 9 levels. You can e.g. refer to $22 but not \22. This is also called *capturing parentheses*. You can print the captured text in perl like this:

```
                        Regular expression to capture "Mike" or "Dan"
                                                        |
% perl -MEnglish -e "$ARG=qq(Dan is fishing); print $1 if /(Mike|Dan)/i"
        |         |  |                       |       |
        |         |  |                       |       captured value
        |         |  |                      command end
        |         |  set variable ARG to "Dan is fishing"
        |        evaluate perl code
       load perl module "English.pm"
```

3.11 Problems with parentheses

You can print the parentheses match with perl variable $N, but counting the possible N becomes interesting when you have marked the parentheses optional:

```
/([0-9][0-9]):([0-9][0-9]) *(am|pm)?/
 |          |           |
 1st        2nd         3rd, BUT OPTIONAL, notice "?" at the end
% perl -MEnglish -e "$ARG=qq(11:12); print $3 if ...."
                                       |
                                      won't print anything
```

3.12 The escape character - backslash

The backslash is used normally in the meaning "take next character as-is not matter what is it's meaning in regular expression syntax.". There is only one exception and it has been listed in backreferencing (\1). To remove e.g the meaning of "match any character except newline" from the dot(.), you would add the backslash

```
ega.att.com             /ega.att.com/       WRONG!
megawatt.computing                          SURPRISE!
                        /ega\.att\.com/     ok
```

The backslash can be used anywhere. Consider the difference between these two. What do they match?

```
/find (word)/
/find \(word\)/
```

3.13 Backslash character notation in regular expressions

You can match space and tab with regular expression /[ ]/ but it is quite difficult to read that there is SPACE and TAB in inside the class notation. Perl, like other languages has alternative notation for few special characters. Here is the list:

```
\t          tab                   (HT, TAB)
\n          newline               (LF, NL)
\r          return                (CR)
\f          form feed             (FF)
\a          alarm (bell)          (BEL)
\e          escape (think troff)  (ESC)
\033        octal char (think of a PDP-11)
\x1B        hex char
\x{263a}    wide hex char         (Unicode SMILEY)
\c[         control char
\N{name}    named char
```

In addition, THESE ARE COMMANDS INSIDE STRING. See [perlop] manpage and the substitute s/// command for full explanation.

```
\l          lowercase next char (think vi)
\u          uppercase next char (think vi)
\L          lowercase till \E (think vi)
\U          uppercase till \E (think vi)
\E          end case modification (think vi)
\Q          quote (disable) pattern metacharacters till \E
```

Using these sequences, a regular expression including space and tab can be written as:

```
/hello[ \t]there/
```

3.14 Line endings \r \n and operating systems

You remember that the dot(.) matches all other characters, except newline, but what if you wanted to match the whole line including the terminating newline, you might want to write:

```
/^(.*)\n/
```

However, In **Win32** operating system, the line ending (like found from C:\autoexec.bat) is combination of two characters CR and LF and to refer to them, you need to explicitly list the characters. Notice that in the regular expression there is no end-of-line anchor $, but only a beginning-of-line anchor:

```
/^(.*)[\r\n]/
      |
      The last character can be either CR or LF
```

To summarize the end of line terminator in various operating systems:

- **\r\n** are used in Windows, usually referred as Win32 operating systems.
- **\n** is used in Unix or Linux systems, such as HP, SunOS, RedHat, Debian, Caldera, Madrake, Slackware, SOT Linux.
- **\r** is used in Mac operating system.

3.15 Perl shorthand regular expressions

In addition to previous backslash notations, some basic regular expression classes have short notations: Keep in mind that perl **word** \w is more a "programming language word", than a spoken language word. The Perl word won't find e.g. Finnish word "linja-auto", because the class doesn't include character "-".

```
\w    Match a "word" character (alphanumeric plus "_")    [a-zA-Z0-0_]
\W    Match a non-word character                          [^a-zA-Z0-0_]
\s    Match a whitespace character                        [ \t\n\r\f\v]
\S    Match a non-whitespace character                    [^ \t\n\r\f\v]
\d    Match a digit character                             [0-9]
\D    Match a non-digit character                         [^0-9]
\pP   Match P, named property.
      Use \p{Prop} for longer names.
\PP   Match non-P
\X    Match eXtended Unicode "combining character sequence",
      equivalent to C<(?:\PM\pM*)>
\C    Match a single C char (octet) even under utf8.
```

The above set is usually used instead of the basic regular expressions, because it makes the expressions shorter and easier to understand. Below there are some equivalent basic regular expressions and their equivalents with short notation:

```
/[0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]/  <=>  /\d\d\d\d-\d\d-\d\d/
/void[ \t][a-zA-Z_]+\(.*\)/                   <=>  /void\s+\w+\(.*\)/
/time is [0-9][0-9]:[0-9][0-9]/               <=>  /time\s+is\s+\d\d:\d\d/
```

If we return to the HTML tag matching, writing a regular expression to match HTML tag can now be written as:

```
<B>          /<B>/
< B>         /<\s*B>/
<B >         /<B\s*>
< B >        /<\s*B\s*/>
<            /<\s*B\s*>/                NOTE: COMPARE TO ABOVE !!
  B          /<[ \t\r\n]*B[ \t\r\n]*>/   ... AND THE OLD STYLE
>
```

4.0 Perl zero width assertions

The term zero-width means that the regular expression element does not ***consume*** any character. It is just a marker, or a spot, where regular expression takes place. In fact the position is ***between*** characters and this is important to understand.

4.1 Beginning of line (^) and (\A)

^	Match the beginning of the line
\A	Match only at beginning of string

The beginning of **line** assertion works between lines, meaning that it can find a place where there is \n or empty position. Consider this multi line string. The possible matches are marked with numbers:

```
"Hello\nthere\nand"          /^./m
 1       2     3
```

There is also more specialized assertion, that matches only at the beginning of **string** (\A) and the only possible match for it even in multi line string is as follows.

```
"Hello\nthere\nand"          /\A./m
 1
```

Exercise: Spend a moment thinking what would plain anchor match? Does anything appear in the perl variable $1

```
/(^)/
```

Exercise: Can you figure out what do the following mean and can they match if line has a trailing newline?

```
/.$/
/.\z/
/\n$/
```

4.2 End of line ($) and (\Z) and (\z)

\Z	Match only at end of string, or before newline at the end
\z	Match only at end of string

The \A and \Z won't match multiple times when the /m modifier is used. The "^" and "$" will match at every internal line boundary. To match the actual end of the string and include trailing newline, use \z.

```
"Hello\nthere\nand\n"
                  1 2
                  \Z\z
```

4.3 Word (\b) and non-word (\B) boundaries

A word boundary (\b) is a spot between two characters that has a \w on one side of it and a \W on the other side of it (in either order), counting the imaginary characters off the beginning and end of the string as matching a \w. Within character classes \b represents backspace rather than a word boundary, just as it normally does in any double-quoted string. You can think \b as as a synonym for "Text must be separated" and \B as synonym for "keep together".

```
/\bcat\b/       "The cat is here."
                "That cat"
                "cat?"
```

You can think \B as "Text must be together"

```
/\Bcat/         "Small work machine: bobcat"
/cat\B/         "The caterpillar"
```

Exercise: Is these regular expressions sensible?

```
/\Bion/
/\w\Bion/
/\wion/
```

4.4 Match continue from last position (\G)

The \G assertion can be used to chain global matches (using m//g), as described in the section on "Regexp Quote-Like Operators" in the [perlop] manpage. It is also useful when writing lex-like scanners, when you have several patterns that you want to match against consequent substrings of your string, see the previous reference. The actual location where \G will match can also be influenced by using pos() as an lvalue. See the pos entry in the [perlfunc] manpage.

You can intermix m//g matches with m/\G.../g, where \G is a zero-width assertion that matches the exact position where the previous m//g, if any, left off. The \G assertion is not supported without the /g modifier. (Currently, without /g, \G behaves just like \A, but that's accidental and may change in the future.)

An example demonstrates this modifier best. You can think the **\G** as the "try next full package match". Here we try to find series of *five* digits from the full line, all of which start with "11". The sequences has been broken for clarity in the comment. See [Friedl] page 236.

```
use English;
$ARG = "112231121155";
#   112 231 121 155;
print "Matched: $1\n" while /(1\d\d)/g;
-->
Matched: 112
Matched: 112
Matched: 115
```

The above is the obvious regular expression that comes to the mind but it does not work right, because the regular expression engine bumps along **one** character every time, resulting wrong answers.

The way to solve this is to **force** regular expression always to match three digits, so that next match will continue from the point of last try position. In the regexp below, the *? is not attempted if the sequence starts with number 1. But if it did not, then it occupies the three digits allowing smooth transition in the string:

```
print "Matched: $1\n" while /\G(?:\d\d\d)*?(1\d\d)/g;
-->
Matched: 112
Matched: 121
Matched: 155
```

5.0 Perl Regular expression modifiers

5.1 Perl match operator

In Perl, the regular expression is enclosed in slashes /REGEXP/, but there is also a formal match command m/REGEXP/ where the delimiter can be chosen. For example when matching filenames that frequently contain slashes, the standard delimiter causes escaping all the "inside" path delimiters:

```
use English;
$ARG = $PROGRAM_NAME;       # internal Perl variable
print "Path: $1 Bin: $2\n" if /^\/(.*\/)(.*)/;
```

It can be written in more readable form by using some other delimiter like:

```
m,(.*/)(.*),;
```

The choice of the delimiter is up to the user, but it is recommended that a visually appealing and clear delimiter is chosen. In above, the colon gives a good **contrast** on the uprising slash. Another common possibilities include:

```
m!(.*/)(.*)!;
m{(.*/)(.*)};       # The standard functional braces {  }
```

Avoidable choices are those would make the code less clear. E.g it is frequently seen in newsgroup news://comp.lang.perl.misc that the comment delimiter is used:

```
m#(.*/)(.*)#;       # Hmmmm, This comment here
m&(.*/)(.*)&;       # Not that better either
m<(.*/)(.*)>;       # Not that bad, sometimes you see this
```

Some of the choices are not supported, try these and it gives you an error (the @ is a perl list data type symbol). You will get error "Useless use of a constant in void context".

```
m@something@;       # Error
m_something_;       # Error
m3something3;       # no numbers allowed
mYsomethingY;       # no characters allowed
```

5.2 Perl substitute command

The very close counterpart to matching is the substitution function, which **replaces** the matched text. The command syntax is almost identical to the match operator. The interesting thing that in theory you could change the delimiter for each portion. That is however **not** recommended:

```
s/REGEXP/SUBSTITUTION/cgimosxe;
s!REGEXP!SUSBTITUTION!;
s,REGEXP,SUBSTITUTION,;
s{REGEXP}{SUBSTITUTION};
s<REGEXP>{SUBSTITUTION};    # possible, but NOT recommended
```

If the regular expression is long, it can be divided in separate lines as just like in the m// by using the /x options. However, the substitution prt must be exactly like that, because all the text in it will constitute a replacement, including any spaces.

```
s                   # Splitting with /x is allowed
{                   # in REGEXP portion
    REGEXP
}
{SUBSTITUTION}x;
```

The captured text like ´$1' is available in the substitution part. The special literal commands are also available, which are listed below. Refer to [perlop] manpage.

```
\l          lowercase next char (think vi)
\u          uppercase next char (think vi)
\L          lowercase till \E (think vi)
\U          uppercase till \E (think vi)
\E          end case modification (think vi)
\Q          quote (disable) pattern metacharacters till \E
```

Here are some examples of the s/// command, be sure to count **three** delimiters in the command.

```
s/\bred\b/blue/g;           # Change all colors
s/^(.*)/\U$1/;              # Whole line to uppercase
s/0x(\d\d)(\d\d)/0x$2$1/g;  # Swap 0x1122 HI-LO => 0x2211 LO-HI
s/n\.a\.t\.o/N.A.T.O/       # All in big letters
s/\Q(n.a.t.o)/\U$1/         # Same but with \U command, note \Q
s/\b([a-z])(\w+)/\u$1$2/    # joe => Joe
```

Exercise: See [Friedl] page 46 and think what does the following substitution do?

```
s/\bJeff/\bJeff/i;
```

5.3 Modifiers in matches

Matching operations can have various modifiers. Modifiers that relate to the interpretation of the regular expression inside are listed below. Modifiers that alter the way a regular expression is used by Perl are detailed in the section on "Regexp Quote-Like Operators" in the perlop manpage and the section on "Gory details of parsing quoted constructs" in the perlop manpage. The modifiers are placed after the regular expression syntax:

```
/REGEXP/modifiers       General syntax
m/REGEXP/cgimosx        The (m)atch command as delimiter change
/REGEXP/cgimosx
/cat/i                  Example: Do case-insensitive "cat" matching
m,/,                    Match the "/" character in string
```

The list of modifiers as of Perl 5.6 are:

- **c** Do not reset search position on a failed match when /g is in effect.
- **g** Match globally, i.e., find all occurrences.
- **i** Do case-insensitive pattern matching.
- **m** Treat string as multiple lines.
- **o** Compile pattern only once.
- **s** Treat string as single line.
- **x** Use extended regular expressions.

5.4 Do not reset position or continue (c)

A failed match normally resets the search position to the beginning of the string, but you can avoid that by adding the /c modifier (e.g. m//gc). Modifying the target string also resets the search position. The perl function pos() tells the position of the last match in string.

```
use English;
#       01234567            <= positions in a string
$ARG = "123 12 4";
print "match $1 " while /(12)/gc;
print "\nMatch pos = ", pos(), "\n";
-->
match 12
Match pos = 6
```

5.5 Global matching (g)

The /g modifier specifies global pattern matching--that is, matching as many times as possible within the string. How it behaves depends on the context. In list context, it returns a list of the substrings matched by any capturing parentheses in the regular expression. If there are no parentheses, it returns a list of all the matched strings, as if there were parentheses around the whole pattern.

```
"123 12 4"
 |    |
 |    /12/g          The "g" finds all matches in string
 /12/                Without the "g", the first match is found
```

A simple perl program demonstrates traversal of all matches:

```
print "position ", pos(), " = $1\n"  while /(12)/g;
--> position 2 = 12
--> position 6 = 12
```

5.6 Ignore case (i)

To ignore character case in matching and treat all the same, you would add this options to the regular expression. If `use locale' is in effect, the case map is taken from the current locale. See the [perllocale] manpage.

```
/<HR>/      Won't find if written in lowercase letters: <hr>
/<HR>/i     Now  will find <hr> <Hr> <hR> <HR>
```

5.7 Lock regular expression (o)

Using a variable in a regular expression match forces a re-evaluation (and perhaps re-compilation) each time through. The /o modifier locks in the regexp the first time it's used. This always happens in a constant regular expression, and in fact, the pattern was compiled into the internal format at the same time your entire program was. Use of /o is irrelevant unless variable interpolation is used in the pattern, and if so, the regexp engine will neither know nor care whether the variables change after the pattern is evaluated the very first time.

/o is often used to gain an extra measure of efficiency by not performing subsequent evaluations when you know it won't matter (because you know the variables won't change), or more rarely, when you don't want the regexp to notice if they do.

```
$regexp = '(\w+)';
print "Matched: $1\n" while  "Hello there" =~ /$regexp/go;
-->
Matched: Hello
Matched: there
```

5.8 Span multiple lines (m)

Treat string as multiple lines. That is, change "^" and "$" from matching the start or end of the string to matching the start or end of any line anywhere within the string.

```
"123 abc\n13 abc\n"
 |       |
 /^1./    /^1./gm       "global" + "multi line" match
```

The following short program demonstrates that the beginning of line anchor **moves** in the string. Pay attention how the match is bound to the string in question with operator =~.

```
#   Alternative way:
#
#   use English
#   $ARG = "12 abc\n13 abc\n";
#
#   print "Matched $1" while /^(1.)/gm;
print "Matched $1" while  "12 abc\n13 abc\n" =~ /^(1.)/gm;
-->
Matched 12
Matched 13
```

5.9 Single line matches and dot (s)

Treat string as **single** line. That is, change "." to match any character whatsoever, even a newline, which normally it would not match.

The /s and /m modifiers both override the $* setting. That is, no matter what $* contains, /s without /m will force "^" to match only at the beginning of the string and "$" to match only at the end (or just before a newline at the end) of the string. Together, as /ms, they let the "." match any character whatsoever, while yet allowing "^" and "$" to match, respectively, just after and just before newlines within the string.

```
"123 abc\n456 def\n"
/c./            Won't find' any characters after "c"
/c./s           Will find "c\n"
/c.s/gs         Will find "f\n"  (and previous "c\n" )
```

5.10 Extended writing mode (x)

Extend your pattern's legibility by permitting whitespace and comments.

These are usually written as "the /x modifier", even though the delimiter in question might not really be a slash. Any of these modifiers may also be embedded within the regular expression itself using the (?...) construct. See below.

The /x modifier itself needs a little more explanation. It tells the regular expression parser to ignore whitespace that is neither backslashed nor within a character class. You can use this to break up your regular expression into (slightly) more readable parts. The # character is also treated as a meta character introducing a comment, just as in ordinary Perl code. This also means that if you want real whitespace or # characters in the pattern (outside a character class, where they are unaffected by /x), that you'll either have to escape them or encode them using octal or hex escapes. Taken together, these features go a long way towards making Perl's regular expressions more readable. Note that you have to be careful not to include the pattern delimiter in the comment--perl has no way of knowing you did not intend to close the pattern early.

The thing to remember is that ***whitespace*** is ignored when you use the */x* modifier

```
/\bcat[a-z]+/;     <=>   / \b  cat [a-z]+ /x;
                           =  ==   =      =   <== spaces ignored
```

In complex regular expressions, you can't write the whole regular expression in one line conveniently, so breaking it into separate lines makes the regular expression more manageable. Notice also the use if */i* modified with the */x* modifier

```
# "Book is here"'
#  ====
# "A novel is interesting"
#  =======
# "The pocket book is A6"
#  ==========
print() if
/
    #  The book name can be at the start of string
    ^(book|novel|pocket)
    |            #    OR-MARKER
    #  There can also be "a" "an" or "the" before the book
    \b(an?|the)[ ](book|novel|pocket)\b
/ix;
```

5.11 Evaluate perl code (e)

This modifier is used with the substitution operator ´s///' and it allows to run arbitrary perl code in the SUBSTITUTION portion. Examples best demonstrate the usage:

```
s/\b(\d+)/$1 + 10/eg;      # increase all values by 10
s/\b(\d+)/$1 * 1.10/eg;    # increase all values by 10%
```

The following will format all numbers to have only exactly two decimals:

```
s/\b(\d+)/sprintf "%.02f", $1/eg;
                  |    |
                  |    Format as "float" values
                  Format specifier starts with %
20.222      =>  20.22
1.12        =>  1.12
200         =>  200.00
```

6.0 Perl Extended regular expression patterns

6.1 Comment (?#text)

A comment. The text is ignored. If the /x modifier enables whitespace formatting, a simple # will suffice. Note that Perl closes the comment as soon as it sees a), so there is no way to put a literal) in the comment.

```
"Test string with abc"
/(?#this-is-comment)abc(?#yet another comment)/
```

6.2 Modifiers (?imsx-imsx)

One or more embedded pattern-match modifiers. This is particularly useful for dynamic patterns, such as those read in from a configuration file, read in as an argument, are specified in a table somewhere, etc. Consider the case that some of which want to be case sensitive and some do not. The case insensitive ones need to include merely (?i) at the front of the pattern. For example:

```
$pattern = "foobar";
if ( /$pattern/i )
{
    # do something
}
# .................. more flexible
$pattern = "(?i)foobar";
if ( /$pattern/ )
{
    # do something
}
```

6.3 Non-capturing parenthesis (?:pattern)

This groups subexpressions like "()", but doesn't make back references as "()" does.

```
use English;
$ARG = "123 456 ";
print "[$1] [$2]"  if  /(123)\s+(456)/;         # [123] [456]
print "[$1] [$2]"  if  /(?:123)\s+(456)/;       # [456] []
```

With the non-capturing parenthesis, you can control exactly, which submatch will contain the answer.

```
(?:  () () (   (?:)    )  )
1    2  3  4   5              Regulatr count of submatches
     1  2  3                  The correct count of submatches
```

A simple example will demonstrate this:

```
use English;
$ARG = "123";
print "[$3]"  if /(((12)))/;        # [12]
print "[$3]"  if /(?:(?:(12)))/;    # []
```

6.4 Zero-width positive lookahead (?=pattern)

A zero-width positive look-ahead assertion. For example, `/\w+(?=\t)/` matches a word followed by a tab, without including the tab in `$MATCH`. Here are more examples:

```
/Bill(?=\s+The\s+Cat|\s+Clinton)/       Match "Bill"
/(?=IBM|HAL)(\w+)/g;                   IBM900   HAL20001
```

Exercise: What lines are matched below?

```
Joe has car
Mike has motorcycle
Bill has Truck
/\b(\w+)\b.*(?=car|truck))(\w+)/
```

6.5 Zero-width negative lookahead (?!pattern)

A zero-width negative look-ahead assertion. For example `/foo(?!bar)/` matches any occurrence of "foo" that isn't followed by "bar". Note however that look-ahead and look-behind are NOT the same thing. You cannot use this for look-behind. Few examples:

```
/\d+(?!\.)/                 Match number, not decimal
/(?!000)\d+/                Exclude 000
/(?!000|255)+\d{3}/         Exclude 000 and 255
```

Be aware that the lookahead also **moves** if the first position is not correct:

```
/(?!cat)\w+/        "cattle"
                      ======
```

You need to say, that it should stay put at the the beginning of word like this:

```
/\b(?!cat)\w+/
```

Exercise: What lines are matched below?

```
/\b(?!mike|joe)\w+\s+\w+\s+\b(?!tall)\w+/i;
Bill is tall
Henry is tall
Jordan is huge, literally
Kenny is weighty
joe is big
mike is ok
```

6.6 Zero-width positive lookbehind (?<=pattern)

A zero-width positive look-behind assertion. For example, `/(?<=\t)\w+/` matches a word that follows a tab, without including the tab in `$&`. Works only for fixed-width look-behind.

6.7 Zero-width negative lookbehind (?<!pattern)

A zero-width negative look-behind assertion. For example `/(?<!bar)foo/` matches any occurrence of "foo" that does not follow "bar". Works only for fixed-width look-behind.

6.8 Zero-width Perl eval assertion (?{ code })

This zero-width assertion evaluate any embedded Perl code. It always succeeds, and its `code` is not interpolated. Currently, the rules to determine where the `code` ends are somewhat convoluted.

6.9 Postponed expression (??{ code })

This is a "postponed" regular subexpression. The `code` is evaluated at run time, at the moment this subexpression may match. The result of evaluation is considered as a regular expression and matched as if it were inserted instead of this construct. The `code` is not interpolated. As before, the rules to determine where the `code` ends are currently somewhat convoluted.

6.10 Independent subexpression (?>pattern)

An "independent" subexpression, one which matches the substring that a *stand-alone* `pattern` would match if anchored at the given position, and it matches *nothing other than this substring*. This construct is useful for optimizations of what would otherwise be "eternal" matches, because it will not backtrack (see the section on "Backtracking"). It may also be useful in places where the "grab all you can, and do not give anything back" semantic is desirable.

For example: `^(?>a*)ab` will never match, since `(?>a*)` (anchored at the beginning of string, as above) will match *all* characters `a` at the beginning of string, leaving no `a` for `ab` to match. In contrast, `a*ab` will match the same as `a+b`, since the match of the subgroup `a*` is influenced by the following group `ab` (see the section on "Backtracking"). In particular, `a*` inside `a*ab` will match fewer characters than a stand-alone `a*`, since this makes the tail match.

An effect similar to `(?>pattern)` may be achieved by writing `(?=(pattern))\1`. This matches the same substring as a stand-alone `a+`, and the following `\1` eats the matched string; it therefore makes a zero-length assertion into an analogue of `(?>...)`. (The difference between these two constructs is that the second one uses a capturing group, thus shifting ordinals of backreferences in the rest of a regular expression.)

6.11 Conditional pattern (?(condition)yes-pattern|no-pattern)

Conditional expression. `(condition)` should be either an integer in parentheses (which is valid if the corresponding pair of parentheses matched), or look-ahead/look-behind/evaluate zero-width assertion.

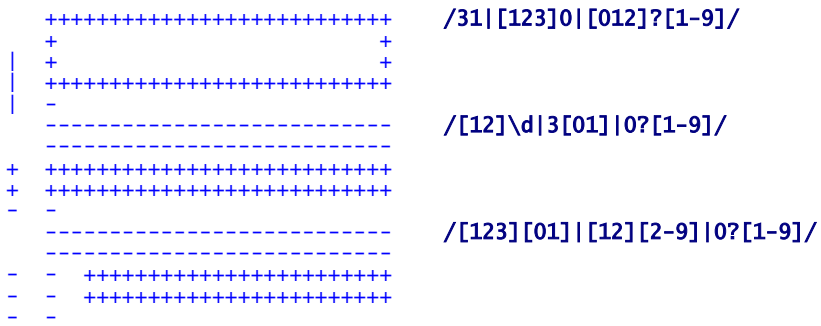
7.0 Regular expression discussion

7.1 Matching numeric ranges

Regular expressions do not about numeric ranges, they only match one number at the time. A number range 0-9 is seen and one character range, whereas 10-99 is interpreted as two digits. Suppose we want to define a month day range 1-31, we could conceptualize the range of numbers in a series:

```
    1  2  3  4  5  6  7  8  9
   01 02 03 04 05 06 07 08 09
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31
```

The possibilities to define the numeric ranges can be visualized by series of rectangles



Exercise: A similar table can be constructed to decide regular expressions for a 24-hour clock. Here is the initial table for the hour part. Define your own regular expression "areas"

```
 0  1  2  3  4  5  6  7  8  9
00 01 02 03 04 05 06 07 08 09
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24
```

7.2 Pay attention to the use of .*

Be very careful when using the "match everything" regular expression `.*`. If you don't save the match anywhere with parenthesis, the regular expression is useless:

```
/this.*/        is same as  /this/
/.*(.*)/        is same as  /.*/    is same as  //
```

The second thing to remember is that it always goes to the **end** of the lines ad only after that it starts backtracking if needed:

```
number 12345            /(.*)(\d+)/;
==========.
|         |
(.*)      (\d+), must have at least one
```

The third thing to remember is that you should use anchor to prevent infinite number of matches from happening. Consider what would happen if string does not contain a digit:

```
this line here              /(.*\d)/
1=============  1st try
1============    2nd try
1===========     3nd try
1==========      ..
1..
1               Still No match, then moves forward 1 character
 2============  1st try
 2===========   2nd try
 2==========    3rd try
 2..
 2              Still No luck, Transmission starts again
  3===========
  3..
  3..           .. And So on ..
```

But using an anchor, you only let regular expression to try only the **First** part of the choices and the transmission doesn't move the regular expression character by character forward. You should use:

```
/^(.*\d)/
```

7.3 Variable names

Many programming languages have identifiers (variables and function names) that are allowed to contain only alphanumeric characters and underscores, but which may not begin with numbers. Here is solution for it. You could limit the length with `{1,31}`

```
/[a-zA-Z_][a-zA-Z_]*/
```

7.4 A String within double quotes

The quotes at either end are to match the open and close quote of the string. Between them there can be anything .. except another quote character.

```
use English;
$ARG = qq(Find this "word" in the line\n);
/\"([^\"]*)\"/              # NOTE: count the double quotes "
  | ======  |
  | |       End
  | Anything in between
  Start
```

7.5 Dollar amount with optional cents

From top level perspective this is a simple regular expression with three parts: " a literal dollar sign, a bunch of one thing, and perhaps finally another thing". This expression is a bit naive, because the optional part doesn't need to match. But if you add beginning of line and end of line anchors, it becomes important. One type of value that it does not take into account is `$.49`.

```
/\$[0-9]+(\.[0-9][0-9])?/
```

7.6 Matching range of numbers

Matching time can be taken to varying levels of strictness. This matches `9:17 am` and `12:30 pm` but also allows `9:99 am`.

```
/[0-9]?[0-9]:[0-9][0-9] +(am|pm)/
```

Looking at the hour part, it must be two-digit number if the first digit is one. But `1?[0-9]?` still allows number `19` and `0`, so it's better to break it into more details: one digit hours and two digit hours:

```
/([1-9]|1[012])/
```

The minute part is easier: the time range is 1-59 or 01-09 and 10-59 or we think of term of individual characters in matches.

```
/[0-5][0-9]/
```

Using the same logic can you extend this to handle 24-hour clock, allow a leading zero in the hour part, like `09:41`.

7.7 Matching temperature values

Temperatures can have both negative and positive values, where the positive value is usually implicit and not written out with "+".

```
/[-+]?[0-9]+/
```

To allow optional decimal part, you must add a decimal point followed by the numeric part:

```
/(\.[0-9]*)?/ right /\.?[0-9]*/     WRONG, why not this?
```

Putting this all together we get:

```
/[-+]?[0-9]+(\.[0-9]*)?/
```

This will allow numbers like 32, -2, -3.723 and +54.4, it is actually not perfect because number `.234` is not found and solving this leads to quite complex regular expression. To actually show in detail what is matched, you could add parentheses;

```
% perl -ne "print qq($1-$2-$3\n) if /([-+]?)([0-9]+)(\.[0-9]*)?/" file.dat
```

7.8 Matching whitespace

To match tabs and spaces you could write:

```
/[     ]*/              There is TAB pressed inside the regexp
```

Perl can also use the standard escape character notation \t to represent the tab and the above can be more cleanly written as:

```
/[ \t]*/
```

Now, an interesting question, how are the two different from each other?

```
/( *|\t *)/
/( |\t)*/
```

7.9 Matching text between HTML tags

Matching HTML is not an easy task and sometimes required lot of expressive and carefully crafted regular expression. Here is simplified regular expression that matched text between the <B> and </B>. You can use the same idea to match other tags. It is simplified, because it doesn't take into account whitespace inside tags, like < B >

```
/<B>[^<]*<\/B>/
    |      |
    |      The "/" must be quoted, backslashed "\/" like in </B>
    |      This is because regular expression is delimited by / .. /
   match any character as long as it is NOT "<"
```

Notice that this almost like *non-greedy* matching, where you could say like below, but this would be slower.

```
/<B>(.*?)<\/B>/
```

The difference between negated and non-greedy construct is best demonstrated in this example. See Friedl page [226].

```
$ARG = "very <very> angry. <Angry!> that is.\n";
                          ====== Negation
                 ==================== Non-Greedy
print "Negation: $1\n"    if /<([^>]*!)>/;
print "Non-Greedy: $1\n"  if /<(.*?!)>/;
-->
Negation: Angry!
Non-Greedy: very> angry. <Angry!
```

7.10 Matching something inside parenthesis

This example is almost same as the HTML tag matching example. We examine mail header field "From" which can have following syntaxes according to Internet mail standards RFC 822 and 1036 at http://www.cis.ohio-state.edu/hypertext/information/rfc.html

```
From: mark@cbosgd.ATT.COM
From: mark@cbosgd.ATT.COM (Mark Horton)
From: Mark Horton mark@cbosgd.ATT.COM
```

We concentrate on the second alternative and try to pick up person's name inside the parenthesis. The regexp looks complicated and you have to read it several times in order to grasp the right "what parentheses mean what" idea:

```
                        find anything else but NOT "()"
            group start  |    group end
                      |  |    |
/^From:\s+(\S+)\s+\(([^()]*)\)
                  |          |
              literal    literal
```

7.11 Reducing number of decimals to three (substituting)

See [Friedl] pages 47, 110-11 for full explanation. Suppose you want to grab two first decimals from the number and optionally third if it is non-zero. That is in case of value 12.375000 you would strip it to 12.375 and in case of value 37.500 number 37.50 would be fine.

```
s/(\.\d\d[1-9]?)\d*/$1/g;
    |
    . 3 7 5
```

Now to the discussion, what if the value is already well-formed to number 27.625. It still replaces something because the match succeeds, but the effect is no-op, because \d* does not match any additional numbers. The number stays in 27.625.

Hm, wouldn't it be a bit more efficient to bypass this ineffective replace and run the command only if there is more than three decimal numbers? okay, we can change the last atom to \d+ to require an additional number. Looks like it's working fine with long numbers still. **But**, wait, what happens to the number 27.625 which has exactly 3 digits? *SURPRISE*, the question mark is not required to get any digit, but the plus atom is. The total effect is that the number is in fact converted to 2.62!!

```
s/(\.\d\d[1-9]?)\d+/$1/g;
    |
    . 6 2  -     5
```

*The lesson here is that match **always** takes precedence over the atoms that not necessarily require any characters.*

Is there a solution for the above? Can we make it match only if there is 3 or more digits? Yes, you can use the lookahead that just peeks more numbers ahead.

```
s/(\.\d\d[1-9]?)(?=\d\d)\d+/$1/g;
                 |
                 make sure that at least 2 digits follow
                 (requiring total of 4 always)
```

8.0 Different regular expression engines

8.1 Regexp engine types

There are two fundamentally different types of engines: one called **DFA**, Deterministic finite automaton, and one called **NFA**, non-deterministic finite automaton. Both engine types have been around for a long time, but the **NFA** type seems to be used more often.

```
NFA         Tcl, Perl, Python, Emacs, ed(1), sed(1), vi(1),
            mawk(1)
DFA         grep(1), egrep(1), awk(1), lex(1), flex(1)
            expr(1)
```

8.2 NFA engine relies on regexp (Perl, Emacs)

Let's consider one way an engine might match `to(nite|knight|night)` against the text `tonight`. Starting with the `t`, the regular expression is examined one component at the time. If the component matches, the next component is checked until all components have been checked. Faced with three possibilities, the engine just *tries each one in turn*. The writer of the regular expression has considerable opportunity to craft just what he wants to happen.

```
after tonight          to(nite|knight|night)
      |                |
after tonight          to(nite|knight|night)
      ||               ||
after tonight          to(nite|knight|night)
      |||              || |
after tonight          to(nite|knight|night)  NOT OK, back
      ||||             || |<>
after tonight          to(nite|knight|night)  NOT OK, back
      |||              ||    <>
after tonight          to(nite|knight|night)
      |||              ||              |
after tonight          to(nite|knight|night)   ..And so on
      ||||             ||              ||
```

As noted above, the writer can exercise fair amount of control simply by changing how the regular expression is written. Perhaps less work would have been wasted had the regexp been written differently such as:

```
to(ni(ght|te)|knight)
to(k?night|nite)
```

8.3 DFA engine reads text

The DFA *tracks all matches* and all possible choices during the evaluation of each component, until it can conclude that there is a match to return.

```
after tonight          to(nite|knight|night)
      |                |
after tonight          to(nite|knight|night)
      ||               ||
after tonight          to(nite|knight|night)
      |||              || |            |
after tonight          to(nite|knight|night)
      ||||             || ||           ||
after tonight          to(nite|knight|night)
      |||||            ||              |||
```

Because DFA keeps track of text position and matches, it is generally faster because each element is checked only once. The NFA engine wastes time attempting to match different subexpressions against the same text. In DFA engine, the way the regular expression is written does not matter at all, because everything is evaluated at the same time.

8.4 Crafting regular expressions

following regular expressions yield completely different results if run on DFA or NFA engine, because NFA takes "what comes first", whereas DFA must try "all matches" and pick the **longest**.

```
/\d\d|\d\d\d/       NFA: "12"  DFA: "123"
```

Another example, see [Friedl] page 112.

```
Three tournaments won?      /tour|to|tournament/
                              |       |
                             NFA     DFA choice
```

The NFA engine happily accepts the first match "tour" and never bothers to look into the other possible matches. What if the regexp were written differently:

```
/to(ur(nament)?)?/
```

Before you try to answer to the question above, you should know that both regular expressions are logically the same: there are three possibilities.

Exercise: The question is: "is question mark DFA *greedy* or like NFA *first served*"

DFA - the longest leftmost match

Consider the following example and remember the rule, that the longest match always wins in DFA:

```
oneselfsufficient       one(self)?(selfsufficient)?
```

The DFA will try all possibilities and conclude that it can match the whole string. The POSIX requires that if there is multiple possible matches that start at the same position, the one matching the most text must be returned. NFA would only find "oneself". The *longest* requirement is demonstrated well if we see and example with parenthesis, see [Friedl] page 117. The regular expression itself could match in numerous ways, including no matches in the middle and still satisfying the whole condition.

```
topological         /(to|top)(o|polo)?(gical|o?logical)/
```

But a POSIX DFA engine has no choice, but always math the longest, whereas the NFA would stick to taking the first one that does the job:

```
top o logical   DFA
to polo gical   NFA
```

If efficiency is an issue, the POSIX NFA engine really does have to try all possible permutation of the regexp every time. A poorly written regexps can suffer extremely severe performance penalties.

Text directed DFA is way around the efficiency penalty that DFA backtracking requires. The DFA engine spends extra time and memory before a match attempt to analyze the regular expression more throughly than an NFA. Once it starts looking at the string, it has an internal map describing all the possible character positions. Building the map, also called *compiling regular expression*, can sometimes take a fair amount of time and memory, but once done for regexp, the results can be applied to unlimited amount of text. In contrast NFA compilation is generally faster and takes less memory.

8.5 Differences in capabilities

An NFA engine can support many things that a DFA cannot. Among them are:

- *Capturing text* with parenthesized subexpression. An information saying where in the text each parentheses matches is available.
- *Lookahead* and *Lookbehind*. "This expression must match before others can continue, but do not consume any of the text"
- *Non-greedy* quantifies and alteration. A DFA could also implement the shortest overall match. They just haven't done it yet.

Some comparison of the size of the typical engines:

- 1979 NFA ed(1) 350 lines of C code.
- 1986 NFA Version 8 general regexp library 1900 lines of C code.
- 1992 DFA sed(1) 9700 lines of C code. o 1992 DFA grep(1) 4500 lines of C code.
- 199? DFA and NFA GNU egrep(1) 2.0 8300 lines of C code.

GNU egrep(1) takes a simple but effective approach. It uses a DFA when possible (due to speed), reverting to NFA when backreferences are used. GNU awk(1) does something similar - It uses DFA engine for simple "does it match" checks and reverts to different engine for checks where the actual extent of the match must be known. AS of 1997, there has been development for adding the capturing support to the DFA engine and the author ***Henry Spencer*** has reported that the penalty is quadratic in text size, where it is exponential in NFA.

9.0 Appendix A - regular expressions

9.1 Perl regular expression syntax

Meta-characters that must be escaped

```
\ | ( ) [ { ^ $ * + ? .
```

Character representation formats

```
\033        octal character
\x1B        hexadecimal character
\cA         control character, like \0x01
0b11110000  binary number, Only in Perl 5.006 and up
```

Character Escape codes

```
\a  Alarm, bell             \x07 \cG    (BEL)
\n  newline, line feed      \x0A \cJ    (LF, NL)
\r  carriage return         \x0D \cL    (CR)
\t  tab                    \x08 \cI    (HT, TAB)
\f  form feed               \x0C        (FF)
\e  escape code             \x1B        (ESC)
```

Regular expression atoms

```
.       Any character except newline \n
[a-z]   Character class, like [\0x00-\0xff]
[^a-z]  Negated character class
|       OR clause, like /foo|bar/
()      grouping or backtracking /(foo|bar) said/
```

Character class order

```
^           Negation at front, otherwise can be anywhere
]           Must be first;last or \[
[           Can be anywhere
-           Range, must be at the beginning or at the end
\           Must be escaped, like \\
```

Perl character class shortcuts, also within a class itself.

```
\d  digit                   [0-9]
\D  non digit               [^0-9]
\w  word                    [a-zA-Z0-0_]
\W  non word                [^a-zA-Z0-0_]
\s  whitespace              [ \t\n\r\f\v]
\S  non whitespace          [^ \t\n\r\f\v]
```

Zero width Assertions (do not consume a character)

```
^           Caret, beginning of _line_. See /m
$           Dollar, end of _line_. See /m
\A          match only at beginning of _string_
\Z          Match only at end of _string_, or before _newline_ at the end
\z          Match only at end of _string_, Perl 5.005
\b          Word boundary, between \w\W
\B          Non-word boundary
\G          Match after previous m//g left off (works only with /g)
```

Assertion Extensions

```
(?= )        positive lookahead
(?! )        Negative lookahead
(?# )        comment
(?: )        Non grouping
(?imsx-imsx) local Perl modifiers, {localized to group in 5.005}
```

In Perl 5.005 new regular expression syntaxes

```
(?<=RE)     positive Look behind
(?<!RE)     negative look behind
(?>RE)      ..
(?{ CODE }) Execute Perl code
(?i:RE)     Combined ?:   and ?imsx-imsx
(?i-x)      Combined ?:   and ?imsx-imsx
(?(COND)YES_RE|NO_RE)
```

In Perl 5.006 new regular expression syntaxes

```
[[:alpha:]] POSIX character class syntax see [perlre]
```

Quantifiers

```
Greedy  Non-greedy
?       ??      0 or 1
*       *?      0 or more, same as {0,}
+       +?      1 or more, same as {1,}
{n}     {n}?    exactly n times
{n,}    {n,}?   at least n times
{n,m}   {n,m}?  at least n times, but no more than m times
```

Perl regexp modifiers

```
/g  Global
/i  ignore case, See the [perllocale]
/m  multi line       "^" and "$" match within string an \n
/o  compile regexp
/s  single line     "." matches \n
/x  extended syntax (allow spaces/comments in multi line regexp)
/e  evaluate perl code
```

Perl regexp commands

```
\Q  Quote meta
\E  End
```

These also respect locale setting. See [perllocale] manpage.

```
\l  lowercase _one_ character
\L  lowercase all until \E or end of string.
\U  Uppercase all until \E or end of string.
\u  uppercase _one_ character
```

9.2 Regular expression engines

Engine pieces are

- literal text
- Character classes, dot, and the like \w \W ..
- Anchors, \b ^ $
- simple parentheses

NFA, Nondeterministic Finite Automaton. (Perl. Emacs)

- Writer can control the hit of regexp ==> Regexp based.
- Refer to example to(nite|knight|night) when matched to "hot tonic tonight". First alternative is used, others are not tried

DFA, Deterministic Finite Automaton.

- No matter how the regexp is constructed it will yield same results ==> text based
- All alternatives are used and the greediest wins.

9.3 Regular expression rules

- First match (earliest leftmost) wins. Regexp is tried completely /fat|cat|belly|your/ --> "the dragging belly indicates your cat is too fat". left-to-right NFA rule: first OR is accepted.
- "Transmission", the regexp is shifted forward to position 0,1,2..
- Regexps are greedy, unless language gives non-greedy. Attempt to match as much as possible. Question mark is greedy, they always take if they can. Remember `happiness` rule with star:

```
    "fox" =~ /x*/
Engine backtracks after greedy
    "foo bar" =~ /.*foo/
Remember to use non-greedy version like here:
    <B>bold</B> and <B>second-bold</B>
```

- First regexp (greedy) is served first. And once they got something, they don't give up from it. Check these:

```
/(.*)(.*)/
/^.*([0-9]+)/
```

- Match always takes precedence over non-match

9.4 How to write good regular expressions

- Be as specific as you can. Use anchors.
- Fixed character `a` or `[ab]` are atomic and fast. Do you find fixed strings?

  `this|that|them` --> `th(is|at|em)`

- Do you really want it all with ".*"?? Double check that.
- Be extra careful with star "*". E.g. s/-*/--/
- Grouping is slow. Do you need grouping results? Use non-grouping.

9.5 Understanding negative lookahead

```
$x = 'ABC123';
$y = 'ABC445';
$x =~ /^(ABC)(?!123)/ and print "1: got $1\n";
$y =~ /^(ABC)(?!123)/ and print "2: got $1\n";
$x =~ /^(\D*)(?!123)/ and print "3: got $1\n";
$y =~ /^(\D*)(?!123)/ and print "4: got $1\n";
```

This prints

```
2: got ABC
3: got AB
4: got ABC
```

But See this

```
$x =~ /^(\D*)(?=\d)(?!123)/ and print "5: got $1\n";
$y =~ /^(\D*)(?=\d)(?!123)/ and print "6: got $1\n";
6: got ABC
```

10.0 Appendix B - Perl language

10.1 Perl manual pages

The main source of information is the perl manual pages that are shipped with perl installation. You can read the pages by using perl itself or with separate utilities available for Emacs.

```
% perldoc -h
% perldoc perl
% perldoc perlre
```

Options:

`-i`	Ignore case
`-t`	Display pod using pod2text instead of pod2man and nroff (-t is the default on win32)
`-u`	Display unformatted pod text
`-m`	Display module's file in its entirety
`-l`	Display the module's file name
`-v`	Verbosely describe what's going on
`-q`	Search the text of questions (not answers) in perlfaq[1-9]

You can find information on specific subject by reading some of these pages:

- *homepage* [perl]
- *Regexps* [perlre] [perlfunc] [perlop]
- *Data Structures* [perlref] [perllol] [perldsc]
- *Objects* [perltoot] [perlref] [perlmod] [perlobj] [perltie]
- *Modules* [perlmod] [perlsub]
- *Moving to perl5* [perltrap] [perl]
- *Options* [perlrun]
- *Linking with C* perlxstut, perlxs, perlcall, perlguts, perlembed

10.2 Useful Perl command line switches

- `-0[digits]` The special value 00 will cause Perl to slurp files in paragraph mode. The value 0777 will cause Perl to slurp files whole because there is no legal character with that value.

- `-a` Turns on auto split mode when used with a -n or -p. An implicit split command to the @F array is done as the first thing inside the implicit while loop produced by the -n or -p.

```
perl -ane 'print pop(@F), "\n";'
```

- `-c` Causes Perl to check the syntax of the program and then exit without executing it. Actually, it *will* execute `BEGIN`, `CHECK`, and `use` blocks, because these are considered as occurring outside the execution of your program. `INIT` and `END` blocks, however, will be skipped.

- `-e commandline` may be used to enter one line of program. If -e is given, Perl will not look for a filename in the argument list. Multiple -e commands may be given to build up a multi-line script. Make sure to use semicolons where you would in a normal program.

- `-Fpattern` Specifies the pattern to split on if -a is also in effect. The pattern may be surrounded by `//`, `""`, or `''`, otherwise it will be put in single quotes.

- `-h` prints a summary of the options.

- `-Idirectory` Directories specified by -I are prepended to the search path for modules (`@INC`), and also tells the C preprocessor where to search for include files. The C preprocessor is invoked with -P; by default it searches /usr/include and /usr/lib/perl.

- `-Mmodule` Load perl `module`, like module English.pm with `-MEnglish`

- `-n` Do not print anything unless the script says so. This switch is usually coupled with the `-e` eval switch as `-ne`

- `-p` The opposite of `-n`. Print always the lines processed. Usually coupled with the `-e` eval switch as `-pe`

- `-S` Makes Perl use the `PATH` environment variable to search for the program (unless the name of the program contains directory separators). You should always use this in Win32 environment. Perl append suffixes to the filename while searching for it. For example, on Win32 platforms, the ".bat" and ".cmd" suffixes are appended if a lookup for the original name fails

- `-v` Prints the version and patch level of your perl executable.

- `-V` Prints summary of the major perl configuration values and the current values of @INC.

- `-w` prints warnings about dubious constructs, such as variable names that are mentioned only once and 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 a number that doesn't look like numbers, using an array as though it were a scalar, if your subroutines recurse more than 100 deep, and innumerable other things.

10.3 Perl environment variables

HOME

Used if chdir has no argument.

LOGDIR

Used if chdir has no argument and HOME is not set.

PATH

Used in executing subprocesses, and in finding the program if -S is used.

PERL5LIB

A colon-separated list of directories in which to look for Perl library files before looking in the standard library and the current directory. Any architecture-specific directories under the specified locations are automatically included if they exist. If `PERL5LIB` is not defined, `PERLLIB` is used.

PERL5OPT

Command-line options (switches). Switches in this variable are taken as if they were on every Perl command line. The most usual switch placed here is the warnings: `-w`