Regular Expressions

An Essential Skill for System Administrators

Presented at BayLISA – April 21, 2005

William R. Ward

Bay View Training

william.ward@bayview.com

http://www.bayview.com

Has This Ever Happened to You?

- Your employer has changed its name and your boss tells you to edit all the Web pages to show the new name.
- A very common scenario nowadays. This has happened to me three times so far in my career:
  - Silicon Graphics » SGI
  - CellMania.com » Cellmania, Inc.
  - Oracle Corporation » Oracle USA, Inc.
- Regular Expressions to the Rescue!

Outline

- Introduction
- History
- Syntax
- Performance Tips
- Regular Expressions in Perl
- My Favorite Regular Expression Tricks

Introduction

What are Regular Expressions Good For?

- Searching for substrings and patterns
- Modifying files (search & replace)
- Transforming data from one format to another
- Extracting key information from log or data files
- ...and much more

Tools with Regular Expression Support

- Command line tools
  - grep, egrep, sed, etc.
- Editors
  - vi, vim, Emacs
- Programming languages
  - Perl, Java, .NET, C, etc.
- Databases
  - MySQL, Oracle 10^g
- Other tools
  - Apache, exim, postfix, procmail, KDE

What do Regular Expressions Look Like?

```
    ? [a-z]
 ? \b   ^  ?
?
  ([0-9]{4})?
 +?   {0,42}
?
  (?i:)  ?? $
```

- To the untrained eye, regular expressions look like a lot of random characters. But each one has meaning!
- The trouble is being able to remember what they all are for...
- Do you know what all of these do?

Regular Expression Examples

- Searching with grep and egrep:
  - `find . -type f -print | xargs grep '^From:'`
  - `egrep '(gif|jpg)$' images.txt`
- Searching with Perl:
  - `print "$1\n" if /^subject:\s*(.*)/i;`
- Search & Replace with vi or vim:
  - `:%s/hello/goodbye/g`
- Search & Replace with Emacs:
  - `M-x query-replace-regexp RET foo RET bar RET`
- Sorting mail with Procmail:
  - `* ^Return-Path: <owner-.*baylisa`

Regular Expressions in Programs

- Perl:
  - `print "Found\n" if ($text =~ /ing\b/);`
- Microsoft .NET / Visual Basic:
  - ```
    If Regex.IsMatch(Text, "ing\b")
        Console.WriteLine("found");
    End If
    ```
- Java:
  - ```
    String re = "ing\\b";
    java.util.regex.Pattern p =
        java.util.regex.Pattern.compile(re);
    java.util.regex.Matcher m = p.matcher(text);
    if (m.find()) {
        System.out.println("found");
    }
    ```

Wildcards are not Regular Expressions

- You've probably used “wildcards” to specify files:
  - * represents any number of characters: `*.txt`
  - ? represents one character: `c?t`
  - `[]` represents a character in the given character class. Examples:
    - `[a-z]` = lowercase letter
    - `[aeiouAEIOU]` = vowel
    - `[0-9]` = decimal digit
    - `[0-9a-fA-F]` = hexadecimal digit
- But these are *not* Regular Expressions!

SQL LIKE is not a Regular Expressions

- Similarly, SQL supports the LIKE operator:
  - % matches any number of characters (like the * wildcard): `'Bay%'`
  - _ matches one character (like the ? wildcard): `c_t`
- With these wildcard rules you can select a set of records in a database query.
  - For example: "`SELECT NAME FROM USER_GROUP WHERE NAME LIKE 'Bay%'`"
- But just like filename wildcards, these are *not* Regular Expressions either!

History

History of Regular Expressions: The Early Years – Theoretical Modeling

- *Early 1940's:* Warren McCulloch and Walter Pitts develop models for understanding the nervous system.

- *1943:* Stephen Kleene invents a notation called *regular sets* to describe neural activity. Later renamed to *regular expressions.*

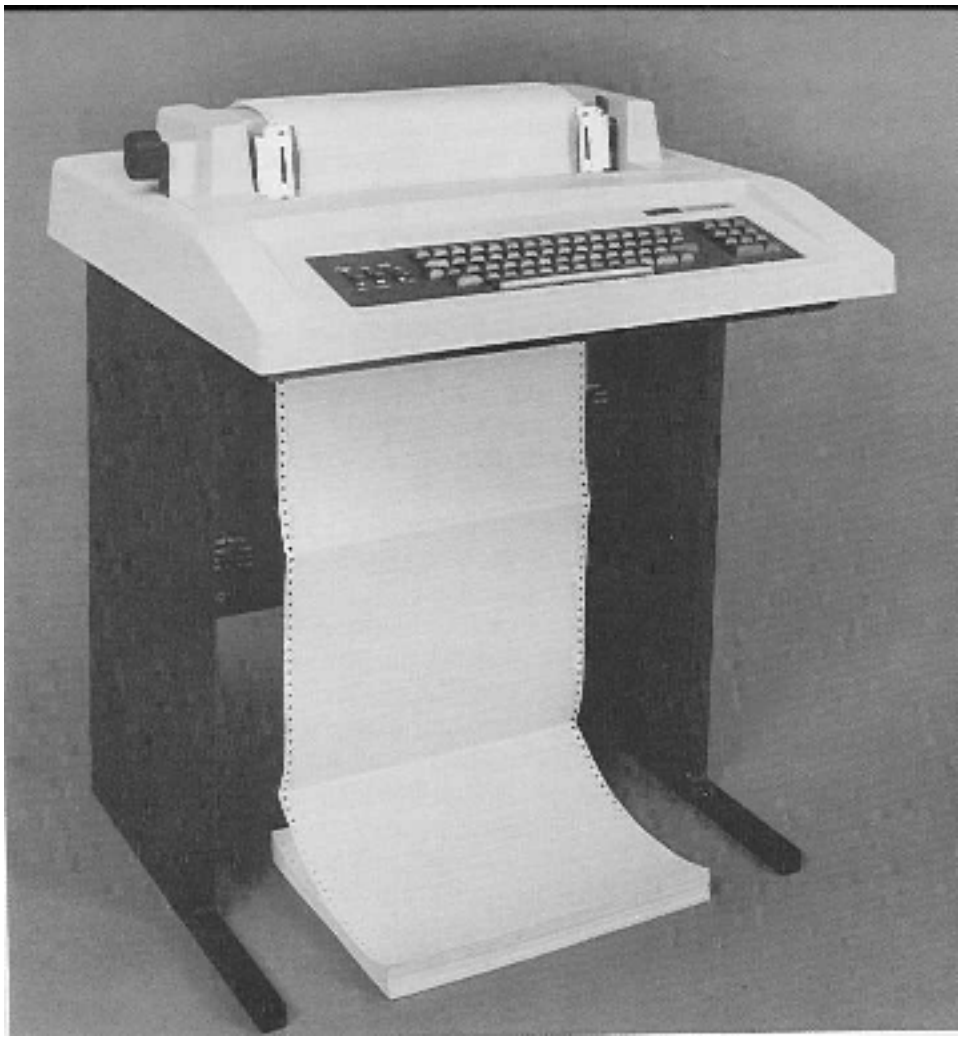
- *1950's-1960's:* Widely used in theoretical mathematics.

History of Regular Expressions: First use in Software – Text Editors

- *1968:* Ken Thompson writes article about regular expressions for searching in computer software.
- *1970:* Thompson's port of the QED editor to CTSS was the first editor to have regex support.
- *1969:* Thompson, along with Dennis Ritchie, created Unix at Bell Labs. Thompson wrote the "ed" editor for Unix, based on "qed", including regular expression support.

Why Edit with Regular Expressions?

- Ever edit a file on one of these?
  - Moving the cursor means reprinting the lines you are editing!
  - Slow line speeds mean it is faster to *describe* changes by giving commands rather than moving the cursor.
  - Using a regex you can describe complex edits and search documents.



Text Editor History

- “ed” was original editor for Unix and ancestor of many other editors:
  - “ex” - enhanced version of “ed”
  - “EDLIN” - early MS-DOS editor (but without regex)
  - “vi” - visual editor for use on video displays. Built on top of “ex” - vi's “colon mode” is actually ex.
  - “vim” - VI Improved
- Other editors not related to “ed,” such as Emacs, also have had regex support added.

The Need for Grep Arises

- One day in 1973 at Bell Labs, Doug McIlroy was working on a voice synthesizer program, and needed to look for words in the dictionary.
- He was doing this by loading the file into "ed" and running the command "g/*<regex>*/p"
  - This still works in vi today: `:g/something/p`
- But the file McIlroy was using was so large, he had to break it up into chunks first.

Grep is Born

- So McIlroy asked Ken Thompson to create a command line tool to print lines from a file or pipe that match a given regular expression.
- Thompson named his program "`grep`" (short for "`g/re/p`") and we still call it that today.
- Later, Lee McMahon (also at Bell Labs) wrote a catch-all tool to mimic other "ed" commands: "sed" (stream editor).

The Grep Family Tree

- The “grep” family of utilities was created to enable searching files from the command line.
  - grep = “Global Regular Expression Print”
  - egrep = “Enhanced grep” (enhanced regex syntax)
  - fgrep = “Fixed-length grep” (optimized for fixed-length)
- Nowadays we mainly use GNU grep, which includes features of all three, plus more.

From “grep” to Embedding in Programs

- *1977:* At Bell Labs Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan write “awk”
  - awk is a rule-based language for transforming text (named for the authors' initials)
- *1986:* Henry Spencer's “regex” package
  - Standard library for regular expressions in C
- *1987:* Larry Wall releases Perl version 1
  - Regular expressions embedded into the language

Perl Regular Expressions Take Over

- *1994:* Perl version 5 released
  - Massive overhaul in regular expression engine
- *1994-present:* Perl regex extensions
  - Addition of `(?:)` syntax & non-greedy operators
  - Unicode support
- *Now:* "Perl-compatible" regex support available for most modern languages:
  - C, C++, Java, JavaScript, .NET, Python, Ruby, etc.

Syntax

Primary Flavors of Regular Expressions

1. grep: The original regular expression language
   - Most restrictive regular expression rules
   - Supported by vi/vim and Emacs (with extras added)

2. egrep: Extended grep.
   - Adds +, ?, alternation, enhanced grouping , and other features

3. Perl-Compatible: Current state of the art.
   - Most languages' regex support is based on Perl's
   - “pcre” (Perl-Compatible RE) library used in many tools

Core Regular Expression Syntax

- Any regular expression (of any flavor) consists of a string of characters of two types:
  - *Metacharacters,* that affect the behavior of the regex in various ways, such as:
    - Require pattern to be at the start and/or end of the line
    - Match a variable number of occurrences of something
    - Specify a set or range of allowable characters
  - *Literals,* specific characters that are searched for.
    - Search for a metacharacter as literal text by adding a backslash. For example, `\.` matches a period.

Common Regular Expression Metacharacters

- All flavors of regular expressions support:
  - `.` (period) – match any character except newline.
  - `*` (asterisk) – match zero or more of the previous thing. Example: `A*` matches zero or more A's.
  - `^` (caret or circumflex) – at the beginning of the expression, indicates pattern must match at the beginning of the line/string.
  - `$` (dollar sign) – at the end of the expression, indicates pattern must match at the end.
  - `[]` (square brackets) – match one character in the range or set of characters given in the brackets.

Common Metacharacter Examples

- Some examples, and strings that they will match:
  - `c.t` – “cat” “cot” “cxt” “ascot” “catharsis”
  - `c.*t` – “cat” “caught” “sect”
  - `^c.t` – “cat” ~~“ascot”~~ “catharsis”
  - `c.t$` – “cat” “ascot” ~~“catharsis”~~
  - `c[aeiou]t` – “cat” “cot” ~~“cxt”~~

Metacharacters for “grep” flavor regexes

- The original regex flavor also supports:
  - `\<` and `\>` indicate the begin or end of a word, respectively.
  - `\(...\)` doesn't affect what is matched, but stores what was matched by the enclosed part of the expression.
  - `\1` through `\9` recall what was matched by `\(...\)`, so you can look for the same string occurring more than once.
  - `\{`*min*`,`*max*`\}` (where *min* and *max* are numbers) lets you specify a specific number of the previous thing.
    - * is equivalent to `\{0,\}`

Examples for “grep” Flavor Metacharacters

- Some examples, and strings that they will match:
  - `\<c.t` – “the cat box” ~~“an ascot tie”~~ “catharsis”
  - `c.t\>` – “the cat box” “an ascot tie” ~~“catharsis”~~
  - `\<c.t\>` – “the cat box cat” ~~“an ascot tie”~~ ~~“catharsis”~~
  - `\(\<[a-z][a-z]*\>\) \1` – “the the” ~~“the cat”~~
  - `ca\{1,5\}t` – “cat”, “caaaaat” ~~“caaaaaaaat”~~ ~~“ct”~~

Metacharacters for “egrep” style regexes

- The “egrep” flavor adds the following:
  - + (one or more) and ? (zero or one) quantifiers are supported in addition to * (zero or more).
  - | (pipe) symbol for alternation: “`hello|goodbye`”
  - \< and \> same as grep, but GNU egrep also allows \b to be used instead of either \< or \>.
  - (…) works similar to \(…\) in grep.
  - \1 through \9 also work as in grep.
  - Some versions of egrep support {*min*,*max*} while others use \{*min*,*max*\}

Examples for “egrep” Flavor Metacharacters

- Some examples, and strings that they will match:
  - `cat|dog` – “the cat box” “doggie bag”
  - `c(o|augh)t` – “cot” “caught”
  - `(\<[a-z][a-z]*\>) \1` – “the the” ~~“the cat”~~
  - `ca+t` – “cat”, “caaaaat” “caaaaaaat” ~~“ct”~~
  - `ca?t` – “cat”, ~~“caaaaat”~~ ~~“caaaaaaat”~~ “ct”
  - `ca{1,5}t` – “cat”, “caaaaat” ~~“caaaaaaat”~~ ~~“ct”~~

Perl-style regular expressions

- Based on the “egrep” flavor with a few changes.
  - Added several new backslash+alphanumeric codes
    - `\d`: shortcut for `[0-9]`
    - `\w`: shortcut for `[0-9a-zA-Z_]`
    - `\s`: shortcut for `[ \n\r\t\f]`
    - Uppercase versions of the above for negated classes
  - Use `\b` instead of `\<` and `\>`
  - Add non-greedy searching: `*? ?? +? {…}?`
  - `(?:…)` for non-capturing parentheses; other `(?…)` type codes for lookahead/-behind, mode modifiers, comments, etc.

Examples for Perl Flavor Metacharacters

- Some examples, and strings that they will match:
  - \d+ – “123” “abc123” ~~“abc”~~
  - \D+ – ~~“123”~~ “abc123” “abc”
  - (\b[a-z][a-z]*\b) \1 – “the the” ~~“the cat”~~

What Metacharacters Need a Backslash?

- Here's how you know when to use backslash (\):
  - In grep, yes on \(\) \{ \} \< \> but no on [ ]
  - In egrep, yes on \< \> but no on ( ) [ ]
    - Some versions of egrep use \{ \} and others use { }
  - In Perl, non-alphanumeric metacharacters *never* use a backslash. No on ( ) { } [ ]
    - It is *always* safe to add a backslash to a non-alphanumeric to search for the character literally.
    - Alphanumeric metacharacters use a backslash, of course. (Remember, use \b instead of \< or \>)

Regular Expression Flavor in Oracle 10^{g}

- Very similar to Perl style but with some changes
  - No `\s`, `\d`, `\w`, etc,
  - Instead, use POSIX character classes such as `[:space:]` for spaces (`\s` in Perl) or `[:digit:]` for numeric digits (`\d`).
  - Does not support Perl's advanced `(?...)` operators

Regular Expression Functions in Oracle 10^{g}

- REGEXP_LIKE()
  - Similar to LIKE but using regex
- REGEXP_INSTR()
  - Find position where pattern found
- REGEXP_SUBSTR()
  - Return substring where found
- REGEXP_REPLACE()
  - Replace matched text with replacement pattern, which may have backreferences!

Oracle 10^{g} Regular Expression Examples

- Conventional SQL allows expressions such as:
  - ```
    SELECT NAME FROM EMPLOYEE
    WHERE EMAIL LIKE '%oracle.com';
    ```
- Using regular expressions this would be:
  - ```
    SELECT NAME FROM EMPLOYEE
    WHERE REGEXP_LIKE(EMAIL, '.*oracle.com');
    ```
- But unlike conventional LIKE, you can use the full power of Regular Expressions!
  - ```
    SELECT NAME FROM EMPLOYEE
    WHERE REGEXP_LIKE(EMAIL,
        '.*(peoplesoft|oracle).com');
    ```
    ```

Writing Efficient Regular Expressions

Writing a Good Regular Expression

- A regular expression is like any piece of software
  - It can be fast; it can be slow
  - It can do a lot; it can do not very much
  - It can have bugs; it can work perfectly
- So when you write a regular expression, take the same care you do when you write software:
  - Know the language well
  - Know quirks of your implementation of the language
  - Consider both the positive (successful match) and negative (no match) cases when writing the regex
    - Ask not only “Does it work?” but “Can I break it?”

Types of Regular Expression Engines

- There are two main types of regex algorithms
  - NFA (Nondeterministic Finite Automaton)
  - DFA (Deterministic Finite Automaton)
- Different regular expression libraries implement one or the other, or a combination
- Optimizations may be added as well to speed up common tasks
- Know what algorithm your tool uses for best results!

NFA vs. DFA

- NFA (Nondeterministic Finite Automaton)
  - Returns “first leftmost” match
  - Can be slow if written poorly
  - Examples: grep, Perl & compatibles, vi, Emacs, less
- DFA (Deterministic Finite Automaton)
  - Returns “longest of the leftmost” possible match
  - Limited capabilities – no capturing parentheses, etc.
  - Examples: awk, egrep, MySQL, Procmail

Combining NFA and DFA Types

- POSIX NFA – NFA with DFA type semantics
  - Returns “longest of the leftmost” – but can be very slow as it must consider every possible match
  - Examples: mawk, MKS utilities
- Hybrids that include features of both
  - Examples: GNU versions of grep, egrep, awk; Tcl
- In this presentation we focus on traditional NFA as used in Perl-type regex engines.

How NFA Works: Backtracking

- An NFA executes the regular expression like it was a program.
- It uses “backtracking” to try each possible match until a successful one is found.
  - Each element tries to match the next thing in the string. If it is unable to match, it rewinds to the last successful point and tries something else.
- Example:
  - Text: “She sells seashells by the seashore”
  - Regex: /sea(scape|urchin|shore)/

Speeding up Regular Expressions

- When writing for an NFA engine, there are some tricks you can use to get the best performance.
- In the examples that follow, the string being matched is:
  - “She sells seashells by the seashore.”

Improving Alternation

- Alternation can be slow because each case requires trackbacking if it fails to match.
  - If possible, factor out common parts of alternation. The common part can then be easily skipped when it doesn't match.
    - `/seashell|seashore/` » `/seash(ell|ore)/`
  - Put the most frequent case first. This way the rarer case is often skipped.
    - `/seash(anty|ell)/` » `/seash(ell|anty)/`

Perl-specific Optimizations

- In Perl use these tricks to speed things up:
  - Use non-capturing parentheses if you can. Reduces storage overhead, and also simplifies backtracking.
    - `/(shell|shore)/` » `/(?:shell|shore)/`
  - Use the case-insensitive option, not character classes, when allowing either upper or lowercase letters.
    - Old versions of Perl were very slow with case-insensitive regexes, but this is no longer true with modern Perl.
    - `/[Ss]he/` » `/she/i`

Anchoring Regular Expressions

- The more you can limit the part of the string that needs to be scanned, the faster it will be.
  - Use leading anchors whenever starting with /.*/ to avoid unnecessary backtracking when match fails
    - `/.*sea/ » /^.*sea/`
  - Any type of anchor usage, whether ^ or $ or \b, will avoid looking in the wrong place or backtracking when not matching successfully.

Splitting up Regular Expressions

- Sometimes a complicated regex is slower than multiple simpler ones:
  - `/seashells|rocks|…|driftwood/`
    - The engine would backtrack for each word not found.
  - `/seashells/ || /rocks/ ||…|| /driftwood/`
    - Each simple pattern is very quick to process separately.

Splitting up Regular Expressions II

- Here, a complicated expression requires a lot of trackbacking when it doesn't match.
  - `/s?he.*sea(?:shells|shore|breeze)/`
- If most lines don't match, this is a lot of work for nothing. But since only lines that match `/sea/` can ever match the whole pattern:
  - `/sea/ && /s?he.*sea(?:shells|shore|breeze)/`
  - Now, the big pattern is not attempted if it's unlikely to match.

Testing Performance in Perl

- Use the Benchmark module to test performance of any Perl code, including regular expressions.
  - ```
    #!/usr/bin/perl -w
    use strict;
    use Benchmark;
    my $s = "abacababacababacababacababacababacababacababba";
    timethese(1000000, {
        a => sub { $s =~ /ab(ba|bc)/   },
        b => sub { $s =~ /(abba|abbc)/ } });
    ```
  - Sample output:
    - ```
      Benchmark: timing 1000000 iterations of a, b...
               a: 12 wallclock secs (12.10 usr +  0.00 sys
      = 12.10 CPU) @ 82644.63/s (n=1000000)
               b: 23 wallclock secs (23.21 usr +  0.00 sys
      = 23.21 CPU) @ 43084.88/s (n=1000000)
      ```

Regular Expressions in Perl

Integrating Regular Expressions into Perl

- Perl's biggest strength is in processing text.
- Its regular expression support is deeply embedded into the language itself.
- Just put the pattern between forward slashes: `//`
  - In contrast, the Java equivalent to the Perl statement "`print if /hello.*world/`" is at least 4 lines long!
- Here's "`grep 'hello.*world'`" in Perl:
  - `while(<>) { print if /hello.*world/ }`

What Variable is it Searching, Anyway?

- To search a specific variable use the =~ operator:
  - `print $str if $str =~ /hello.*world/;`
- But you don't have to list a variable. Many things in Perl use $_ when no variable is specified. These are equivalent:
  - `print $_ if $_ =~ /hello.*world/;`
  - `print if /hello.*world/;`

Leaning Toothpick Syndrome

- Since / is the default delimiter, when using a regular expression that contains / you have to escape it:
  - `print if /\/usr\/local\/bin/;`
- Or you can avoid this by choosing a different delimiter. But you must add “m” (match) so Perl knows what you're trying to do:
  - `print if m!/usr/local/bin!;  # using !`
  - `print if m{/usr/local/bin};  # using { }`

Variables Inside Regular Expressions

- You can have a Perl scalar or array variable inside a regular expression.
  - Contents of variable are treated as part of the pattern
  - Any metacharacters in the variable are interpreted:
    - ```
      $pattern = "seash(ore|ells)";
      print if /she sells $pattern/i;
      ```
  - To force Perl to treat the variable's contents as literal text use \Q before and \E after.
    - ```
      $pattern = "seashore.";
      print if /by the \Q$pattern\E/i;
      ```
    - Here, it will look for a literal "." at the end.

Accessing Captured Text

- If the pattern contains parentheses the portion of the string matched by that part of the pattern is "captured" and can be accessed in two ways:
  - Using numbered variables $1, $2, etc.:
    - `print "Found $1\n" if /(sea\w+)/;`
  - Assigning to a list:
    - `($y, $m, $d) = m[^(\d+)/(\d+)/(\d+)$];`

Greed

- The quantifying operators (*, +, ?, {}) are by default *greedy*.
  - This means they grab as much as they can, provided the pattern can still match. For example:
    - `($a, $b) = m!^(.+)/(.+)$/;`
  - How will it parse the string "`/usr/bin/perl`"?
    - The first `.+` grabs as much as it can, provided there is still a "/" and some more text after, so it gets "`/usr/bin`" and the other `.+` gets what's left: "`perl`"

Hazards of Greed

"Greed is Good!"
– Gordon Gekko

From the film *Wall Street* (1987)
Actor: Michael Douglas

- As in real life, greed is not always good.
  - Consider the string "Cupertino, CA 95014" and the regex "(.*)([0-9]+)"
  - You might think that .* gets "Cupertino, CA " and that the [0-9]+ gets "95014"
  - But no: .* gets "Cupertino, CA 9501" leaving only "4"!

Non-Greedy

- Non-greedy versions of those operators exist. Just add a ? after them (*?, +?, ??, {}?).
  - To parse the string “Cupertino, CA 95014” the way we want, make the .* be non-greedy: “(.*?)([0-9]+)”
  - The result is now as you would expect: the first part gets “Cupertino, CA ” and the second part gets “95014”
  - But change + to +? and you get just “9” - do you see why?

The Problem with Non-Greedy Operators

- The problem is performance.
- To do non-greedy searching, we have to backtrack a lot.
  - In “Cupertino, CA 95014” the .*? part would prefer to match *nothing*.
  - However since the [0-9]+ part can't match on “C” it has to backtrack and have .*? take the “C” after all.
  - It repeats the process for “u” and the other characters until finally it reaches the “9” character.
- So don't use non-greedy unless you have to.

Substitution

- Use the `s///` operator to search & replace:
  - Change “hello” to “goodbye”: `s/hello/goodbye/;`
  - The first part is a regular expression; the second part is a string. The string may contain `$1`, `$2`, etc. to include what was matched:
    - `s/(\d+)-(\d+)-(\d+)/$2-$1-$3/;`
  - You can use alternative delimiters to avoid lots of `\/`:
    - `s,/usr/local,/usr/local/bin,;`
  - If you use `(` or `[` or `{` or `<` then use its partner at the end, and one of each in the middle:
    - `s{/usr/local}{/usr/local/bin};`

Regular Expression Modifier Suffixes

- Suffixes at the end of the pattern alter behavior:
  - Case insensitive: `/i`
  - Global mode (match all occurrences): `/g`
  - Multi-line mode (affects "^"and "$"): `/m`
  - Forced-single-line mode (affects behavior of "`.`"): `/s`
  - Compile only once: `/o`
  - Extended whitespace mode (see next slide): `/x`
- Operators can be combined (in any order):
  - Case insensitive & multi-line: `/^[a-z]/mi` (or `/im`)

Extended Whitespace Mode

- Add /x at the end to make it more readable:
  - ```
    my $re = qr/^                          # Start of line
         (\d+)                             # Day of month
         \s* - \s*                         # Hyphen
         (Jan|Feb|Mar|Apr|…|Dec)           # Month
         \s* - \s*                         # Hyphen
         (\d+)$/ix;                        # Year
    ```
  - Add whitespace inside the pattern for readability.
    - Actual spaces to be searched for in the pattern need to be changed to \s or escaped with backslash.
  - Anything starting with # is a comment and is ignored.
    - Use \# to search for a literal# sign
    ```

Splitting on Regular Expressions

- split creates a list or array from a string:
  - `my @words = split(":", $sentence);`
  - But instead of a mere delimiter you can use a regex:
    - `my @words = split(/,\s+/, $sentence);`
  - If the regex has capturing parentheses, the matched text is also returned:
    - `my @tags_and_content = split(/(<[^>]+>)/, $html);`
    - Given the string "<html> Hello World </html>" the result is ("", "<html>", "", " ", "Hello World", " ", "", "</html>").

Perl Command-Line Options

- Instead of creating a Perl script file, for one-off jobs you can write the Perl code directly on the shell command line using the `-e` option:
  - `perl -e 'print "Hello, World!\n"'`
  - Combine this with the `-n` option to run the program as a loop over each line of the input file(s). This line acts just like a grep command:
    - `perl -ne 'print if /hello/i' foo.txt`
  - Here is the equivalent grep command:
    - `grep -i hello foo.txt`

How "`perl -n`" Works

- In a nutshell, "`perl -ne` *CODE*" works the same as a Perl script that contains:
  - ```
    while (<>) {
        CODE
    }
    ```
  - If files are listed on the command line (`foo.txt` in the last example) the files are read in the order listed.
  - If no files are listed then STDIN (keyboard) is read.
  - Note that this is exactly what "grep" and many other Unix stream utilities do.
    ```

Convenience Option: -p

- A large amount of the time, "`perl -ne`" scripts will end up modifying the contents of the line and printing it.
  - `perl -ne 's/hello/goodbye/g; print'`
- This is so common there's even a command line option for this: `-p`
- The following is equivalent to the above example:
  - `perl -pe 's/hello/goodbye/g'`

My Favorite Regular Expression Tricks

Matching Delimited Text
The Problem

- A very common task is to match text with delimiters. Many examples of this can be found:
  - Text inside quotes
  - HTML or XML tags
  - Text inside HTML or XML tags
  - Parsing CSV (Comma-Separated Values) files
- In general they are all done the same way:
  - Match the opening delimiter
  - Match the contents (up to the closing delimiter)
  - Match the closing delimiter

Matching Delimited Text
HTML Tag Solution

- For example consider the HTML tag:
  - `<img src=logo.gif width=100 height=100>`
  - Opening delimiter is <
  - Closing delimiter is >
  - So use one of these regular expressions:

```
/<([^>]+)>/          /<(.+?)>/
```

  - The second case (non-greedy) would normally be slower, but Perl optimizes it, since > is one character.
  - The results (returned in $1) would be:
    - `img src=logo.gif width=100 height=100`

Matching Delimited Text
HTML Tag Contents Solution

- To match the inside of the HTML tag:
  - `<p>She sells <i>seashells</i> by the seashore</p>`
  - Opening delimiter is `<p>`
  - Closing delimiter is `</p>` (not `<` due to `<i>` inside)
  - So use this regular expression:

    `/<p>(.+?)<\/p>/`
  - Here, since `</p>` is more than one character you must use the non-greedy form.

Sorting Yahoo! Groups Email with Procmail
The Problem

- Incoming email messages from Yahoo! Groups (and most other mailing list systems) have extra headers added that are usually not displayed:
  - `Mailing-List: list SVEvents@yahoogroups.com; contact SVEvents-owner@yahoogroups.com`
    `Delivered-To: mailing list SVEvents@yahoogroups.com`
    `List-Id: <SVEvents.yahoogroups.com>`
  - If we can search for one of these lines, we can find the list name and put it into the appropriate sub-folder.

Sorting Yahoo! Groups Email with Procmail The Solution

- Add this to your .procmailrc:
  - ```
    THISMONTH=`date +'%Y-%m'`
    :0:
    * ^Mailing-List: list \/[A-Za-z_-]+
    $LISTDIR/$MATCH.$THISMONTH
    ```
  - In Procmail, the text matched after the special code `\/` is stored in the variable $MATCH. We use this to generate the filename. For example:
    - ```
      Mailing-List: list SVEvents@yahoogroups.com;
        contact SVEvents-owner@yahoogroups.com
      ```
  - The message would then be stored in:
    - `Lists/SVEvents.2005-04`

Batch-renaming Files
The Problem

- Suppose you have a large number of files with regular names, and you want to rename them.
  - Example: File names contain dates in the form “MM-DD-YY” and you want it to be “YYYYMMDD”
    - `04-21-05.txt » 20050421.txt`
  - Solution: For each filename, construct a new filename and issue the command to rename the file.
  - Some options for doing this:
    - Write a Perl script to do it
    - Construct a Shell script using the editor's regex system

Batch-renaming Files
Perl Solution

- To do this in a Perl script, parse the old filename using a regular expression, and construct the new filename, like this:
  - ```
    opendir(DIR, ".") or die "opendir: $!\n";
    foreach (readdir(DIR)) {
        if (/^(\d\d)-(\d\d)-(\d\d)\.txt$/) {
            rename($_, "20$3$2$1.txt")
                or die "rename $_: $!\n";
        }
    }
    ```
  - Pros: Reusable, with good error detection
  - Con: Too much work for a one-off job
    ```

Batch-renaming Files
One-Time Shell Script Solution

- For a one-time renaming job, sometimes it's easier to just create a file containing “mv” commands, and run it as a shell script.
  - ```
    ls ??-??-??.txt > x
    vi x
    :%s/^\([0-9][0-9]\)-\([0-9][0-9]\)-\([0-9]
    [0-9]\).txt$/mv & 20\3\2\1.txt/
    :wq
    sh x
    ```
  - Pro: Quick and easy once you know the regex
  - Cons: No error checking; not reusable
    ```

Mass Editing of Text Files
The Problem

- Your employer, Acme Corporation, has changed its name to “Acme USA, Inc.” and now you have to edit all the Web pages to show the new name.
  - You can get a list of filenames using a command like:
    - ```
      find . -type f -print | \
      xargs grep -l 'Acme Corporation'
      ```
  - But how to actually modify the files?
  - Answer: Perl's edit-in-place command-line option!
    - ```
      perl -i~ -pe \
      's/Acme Corporation/Acme USA, Inc./g' \
      file.txt
      ```

Mass Editing of Text Files
Putting the Pieces Together

- Combine these lines to produce the command:
  - ```
    find . -type f -print | \
    xargs perl -i~ -pe \
    's/Acme Corporation/Acme USA, Inc./g'
    ```
  - The original contents of each file will be saved in a copy with the ~ suffix added (or whatever you put after the -i option).
  - Warning: Verify that it worked after *each* command of this type, as the backup file may be overwritten by running it more than once!
  - But what if there's a newline between “Acme” and “Corporation”? Then it won't work!
    ```

Mass Editing of Text Files
Final Solution: Allowing for Line Breaks

- It's easy, just add –00 (dash zero zero)
  - –00 puts Perl into “paragraph mode” - instead of reading the file one line at a time it reads one paragraph (defined by blank lines) at a time.
    - If there might be a blank line after “Acme” use–0 instead for “entire file mode.” But if the files are large this uses lots of memory to load the entire file into memory.
  - Also change the space after “Acme” to \s+, which also matches newline or tab characters.
    - ```
      find . -type f -print | \
      xargs perl -i~ -00 -pe \
      's/Acme\s+Corporation/Acme USA, Inc./g'
      ```
      ```

Thank You

- Visit our Web site to download a copy of this presentation.
- Be sure to take a handout which has a syntax cheat sheet & examples.

William R. Ward
Bay View Training

william.ward@bayview.com
http://www.bayview.com