Basic Concepts in Perl

Lesson 2

Processing the data:

Search and Replace in Strings

Regular Expressions

Search and Replace without Regex

- Search with `index(string, substr, offset);`

  e.g. `$found = 1 if index ($_, "string") >= 0;`

- Replace with `substr(string, offset, length);`

  e.g. `$who='Friebel'; substr($who,4,2) = 'd';`

  negative offsets count from the end of the string

  e.g. `substr($who,-2,1) = 'be';`

- `substr` can be used on RHS (ordinary function and be assigned to (LHS)

- `index` and `substr` faster than regex (internally `index` is used for `m/constant_string/`)

Searching for Strings in Tables

- **`unpack`** may be more effective than regex for tables with fixed column widths
- `@fields = unpack pattern, string`

  string gets chopped according to pattern

  fixed width pattern, e.g. “A10xA5” (10char, ignore 1, 5char)
- `$string = pack pattern, @fields` is the inverse
- pack/unpack can also be used to convert between ASCII and various binary representations of data (see next lesson)
- for more patterns see **`perldoc -f pack`** and **`unpack`**

Examples for pack/unpack

- Unpack constant width items

```
$string = '123I56I89';
@field = unpack 'A3xA2xA5', $string;
```

- Pack it back to a string (without the separator *I*)

```
$string = pack 'A3A2A5', @field;
```

Processing text with delimiters

- `split` may be more effective than regex for data separated by constant delimiters (e.g. Csv.)
- `@fields = split /:/, $line;`
- `$line = join ':', @fields` is the inverse
- As `split` accepts regular expressions as the first argument, it will be covered in more detail later
- The lines of syslog files can be split into parts (delimiter is space), see script `02syslog.pl`, derived from `01gzread.pl`

Regular Expressions

- used for search and replace operations in strings
- one of the most powerful concepts in perl
- are used in other programs as well:

  awk, sed, vi, egrep (subset of perl regular expressions)
- For a throughout discussion see

  “Mastering Regular Expressions” by J.Friedl (O’Reilly)

Regex Operators (1)

- Search is performed with the match operator `m/regex/`

  Delimiter rules like for q, qw, ... (also `m(), m[], m!! ...`)

  Can be written as `/regex/` if the delimiter is `/`
- Replace is done with the substitute Operator `s/regex/string/`

  Delimiters: 3 times the same char or parentheses in pairs:

  `s/a/b/; s(a)(b); s[a]{b}; s/a/(b);`
- Options can modify the search/replace operation:

  `e.g. m/regex/i;` and `s/regex/string/g;`

Regex Operators (2)

- By default the search/replace target is `$_`
- Changeable by pattern binding operators `=~` and `!~`

```
$ok    = 1 if $a =~ /string/; #Search in $a
$notok = 1 if $b !~ /string/; #Search in $b
$ok    = 1 if $c =~ s/foo/bar/;#Replace in $c
```

- In scalar context result is true/false
- In list context result is list of found/replaced strings

Regex Elements (1)

- All **ASCII Chars** (Chars \|()[{^$*+?. with leading \)

  `/this is a dot in parentheses: \(\.\)/;`

- **Aliases** for special characters, e.g.:

  \a \e \n \cC \t (beep, ESC, newline, CTRL-C, tab)

- **Metachars** (Chars \|()[{^$*+?. and other chars with leading \)

  `/^\d+ items/; #Metachars ^, \d` and **`+`**

- **Character Classes** denoted by Metachars or **`[..]`**

  `[yYjJ] [a-z0-9] \d`

- **Alternation** by using the | Metachar, e.g.: /quit|exit/

Regex Elements (2)

- Begin End (^ and $) and other String **Positions**

```
/^$/; # the empty string
```

- **Grouping** (...)
- The **Quantifiers** ? * + {n} {min,max}

```
/\d{1,3}/; # a number with 1..3 digits
```

- The "**anychar**" Metachar . (not \n, but see option s)
- **Assertions** (and other extensions), start with (?

Comments in Regexes

- Regular Expressions are fairly unreadable
- Inline Commenting with

```
(?# this is a Comment)
```

- Extended Commenting using Option **x**
  - White space is ignored, Comments by using #

```
m {
        a|b  # an Alternation, a or b
   }x
```

Character classes

- Do match a **single** char contained in the class
- List of chars: [jJyY] or [^jJyY] (not jJyY)
- Character ranges: [0-9a-fA-F]
- Metachars as shortcuts for character classes:

\w	word char [a-zA-Z_0-9]	\W	nonword char
\d	digit [0-9]	\D	nondigit
\s	whitespace [\t\n\r\f]	\S	non-whitespace

- Since perl 5.6: Unicode and POSIX Char Classes

Quantifiers

- Describe, how often a Char (or a group of Chars) should match, gets appended
- * 0 or more times, same as {0,}
- ? 0 or 1 ´times, same as {0,1}
- + at least 1 time, same as {1,}
- {min,max} at least min, at most max times
- {n} precisely n times, same as {n,n}
- {n,} at least n times

String Positions

- Also known as zero width assertions or anchors
- Most widely used: Beginning `^` and End `$` of String

  match also after / before newline (with option m)

  `\A` and `\Z` match only at beginning and end of string
- `\b` matches at a word boundary and `\B` not there

Optional Regex Elements

- Expressions that allow a set of possibilities to match
  - Alternations with |
  - Use of Quantifiers ?, *, +, {min,max} or {min,}
- Alternations tried from left to right
- Elements with Quantifiers tried as much as possible
  - such Expressions are greedy (maximal matching)
  - Alteration of the greedy behavior by a trailing ?
  - Then optional elements are skipped if possible
  - leads to non-greedy or minimal matching

Maximal Matching Example

- Look in `"Doris,Petra,Hera,Tesla"` with Regex

  `/,.*,/ # greedy, maximal matching`

  - Look for first comma (5 times false, 1 time true)
  - Look for .* (true up to end of string, alternatives exist)
  - Look for comma (false letter **a** found)
  - Go back to alternative (to **l** , match comma, false)
  - Repeat this step (backtracking) until comma found

  ◆ therefore the result is: ”**,Petra,Hera,**”

Minimal Matching Example

- Look in **`"Doris,Petra,Hera,Tesla"`** with Regex

  `/,.*?,/ # non-greedy, minimal matching`

  - Look for first comma (5 times false, 1 time true)
  - Skip .*, look for comma (false, letter **`P`** found)
  - Go back to alternative (to **`e`** , match comma, false)
  - Repeat this step (4 more times) until comma found

  ◆ therefore the result is: ”**`,Petra,`**”

  `/,[^,]*,/ # non greedy, fastest Search`

  - Elements (for this string) most often true, hence fast

Backtracking Problems

- Regex engine is most effective if backtracking rarely occurs. Heavy usage of backtracking can be triggered by nested quantifiers like in `(\d+)*a`
  - "123a" succeeds in 5, "1234" fails after many steps
  - Badly written regex needs exponential time or excessive memory and may dump core
  - Some bad patterns cured by regex optimizer
- **Avoid nested regex with quantifiers**

Subpatterns

- Definition by enclosing pattern in parentheses
- Within a regex backreference with `\1`, `\2`...possible
- After successful match Variables `$1, $2` ... filled
- Variables not filled if assertion used `(?...)`
- Parens do cluster (subpattern) and capture (`$1`)
- If only clustering required use `(?:`***`subpattern`***`)`

```
/,(.*?),(.*?),(.*?)/;# $1='Doris',$2='Petra'
# $3='' as the last (.*?) matches '' !!!
```

The Variables $`, $& and $'

- are filled with parts of the string to be matched:
  - `$`` : first part of string that did not match
  - `$&`: part of string matching the regex
  - `$´`: remaining part of the string
- Variables get filled if they are at least used once
  - speedup if not used at all, `$&` has least overhead
- Since perl 5.6 the arrays `@+` and `@-` can be used
- No performance penalty, more powerful

The Variables `@+` and `@-`

- These arrays contain the first and last character position of the last pattern match
- `$-[0]` is the start position of the entire match
- `$+[0]` is the end position of the entire match
- `$-[`*n*`]` `$+[`*n*`]` are the corresponding values for `$`*n* that is the *n*th matching subpattern
- Therefore if `$x` matched then the equivalent of `$`` is

```
$substr($x, 0, $-[0])
```

Assertions

- Expressions (of length 0!) that need to match
- Simplest Assertions: Positions like ^, $ and \b
- Positive Lookahead Assertion `(?=pattern)`

  to prevent matching too much in greedy matches
- ´Not covered:

  other lookahead assertions (?=...) (?!...)

  lookbehind assertion (?<=...) (?<!...)

  and other constructs for obfuscated perl contest

Options (1)

- Both the m and s operators allow for options
- Most options the same for m and s
- Meaning of the options (option x see above):
  - s - treat string as single line
  - m - treat string as multiline string
  - i - case insensitive search
  - g - globally find or replace all matches
  - o - optimize (compile string only once)
  - e - treat replace string as expression

Options (2)

- Options s and m for strings with newlines:
  - a .* match halts at \n, as \n not in .
    - Change with option **s**: . now matches newline as well
  - ^ and $ yield true only at beginning and end of string
    - with option **m** ^ and $ do also match after/before \n
- Use option **i** to ignore the case of letters in search
- The **g** option tries to match/replace all occurrences
  - then result = number of successful matches (scalar)
  - result = found/replaced strings (list context)

Regex examples using options

```
$n = $str =~ s/,/\n/g;
print "$n Replacements:\n$str\n";
# Option i
print "/pEtRA/i does match\n" if $str =~ /pEtRA/i;
# Option m
print "/^Petra/ does not match\n" if $str !~ /^Petra/;
print "/^Petra/m does match\n" if $str =~ /^Petra/m;
# Option s
print "/Petra.Hera/ does not match\n" if $str !~ /Petra.Hera/;
print "/Petra.Hera/s does match\n" if $str =~ /Petra.Hera/s;
```

Debugging regular expressions

- `use re 'debug';` can be used to understand regular expressions (see `perldoc re`)
- Standalone regex debuggers do exist
  - See e.g. http://weitz.de/regex-coach
- Regexp debugging implemented in some IDE (e.g. Active State's Komodo

Regex in the split function

- **`split`** is the only function that accepts a regex

  ```
  @fields = split pattern, string
  ```

  chops string according to delimiter ***`pattern`***

  `pattern` is written as **`'pat'`** or **`/pat/`**, not **`"pat"`**

- An empty pattern splits ***`string`*** into characters

  ```
  @digits = split //, "0123456789";
  ```

- Additional elements get created for each subpattern

  processing of config files (containing lines **`key = val`**)

  (see example below)

Splitting text

- processing config files (containing lines **`key = val`**)

```
$lines = "key1 = value1\nkey2 = value2\n";
%conf = split /\s*=\s*(\S+)\n/, $lines);
for ( sort keys %conf ) {
    print "Key:$_, Value:$conf{$_}.\n";
}
```

There is more than one way to do it

- Extract the words from `$str = "abc def ghi jkl ";`
  - with unpack

```
$fmt = "A3x"x4; @words = unpack $fmt, $str;
```

  - with regex

```
@words = $str =~ /\b\S+\b/g;
```

  - with split

```
@words = split / /, $str;
```

  - with substr (by destroying the original string)

```
@words = ();
push @words, substr($str,0,4,"") while $str;
```

Regex idioms

- Removing white space from both ends of string

```
s/^\s*(.*?)\s*$/$1/; # not recommended, slow
s/^\s+//; s/\s+$//;  # the recommended way
```

- Get the name of the executed program

```
($program = $0) =~ s(^.*[\\/])();
```

- Swap two words delimited by white space

```
s/(\S+)\s+(\S+)/$2 $1/;
```

Suggestions for further reading

- Have a look in the regex manpage of perl
  - `perldoc perlre`
- Read the tutorials
  - `perldoc perlpacktut`
  - `perlretut`
  - `perlrequick`

Suggestions for further reading (2)

- Get acquaintant with Unicode and POSIX support starting with Perl 5.6, improved in Perl 5.8
  - it affects the way how a regex has to be written
  - better internationalization
  - more character classes
- Try to understand the rules for the regex engine
  - (Camel book 3rd edition p. 197 ff)