# 5.Textverarbeitung

**Suchen und Ersetzen in Strings**

**Reguläre Ausdrücke**



# Suchen und Ersetzen ohne Regex

- Suche mit `index(string, substr, offset);`
  - z.B. `$found = 1 if index ($_, "text") >= 0;`
- Ersetzen mit `substr(string, offset, length);`
  - z.B. `$who='Friebel'; substr($who,4,2) = 'd';`
  - negative Offsets zählen vom Ende des Strings
- `substr` kann auf linker und rechter Seite benutzt werden (als Funktion und als Zuweisung)
  - z.B. `substr($who,-2,1) = 'be';`
- `index` und `substr` sind schneller als regex (bei `m/constant_string/` wird intern `index` benutzt)

# Stringsuche in Tabellen

- **`unpack`** kann für Tabellen mit fester Spaltenbreite besser geeignet sein als reguläre Ausdrücke
- `@fields = unpack pattern, string`
  - String wird gemäß Pattern zerhackt
  - Beispiel: “A10xA5” (10Zeichen, ignoriere 1, 5Zeichen)

Geben Sie von einem `ls -l` Output Filename und Länge aus

- `$string = pack pattern, @fields` packt wieder ein
- pack/unpack kann auch gut zum Konvertieren zwischen ASCII und binären Formaten genutzt werden (s. Lektion 4)
- Weitere Anwendungen: **`perldoc -f pack`** und **`unpack`**



# pack/unpack Beispiele

- Entpacke eine 3 spaltige ASCII Tabelle

```
$string = 'I 123I 45I   67I';
@field = unpack 'xA4xA3xA5x', $string;
```

- Packe die Zahlen wieder (ohne Trennzeichen *I*)

```
$string = pack 'A4A3A5', @field;
```

# Text mit Trennzeichen

- `split` kann für Textzeilen mit konstanten Trennzeichen (z.B. CSV) besser sein als Regex
- `@fields = split /:/, $line;`
- `$line = join ':', @fields` ist inverse Funktion
- Geben Sie die Accountnamen aus /etc/passwd aus
- Da `split` reguläre Ausdrücke im ersten Argument benutzen kann, gibt es später mehr dazu
- Zeilen von Syslog Files können mit split gut zerlegt werden (Trennzeichen ist `' '`,nicht `/ /`), siehe Skript `05syslog.pl`, abgeleitet von `04gzread.pl`



# Reguläre Ausdrücke

- Zum Suchen und Ersetzen in Strings
- eins der mächtigsten Konzepte in Perl
- werden auch in anderen Programmen benutzt:
  - awk, sed, vi, egrep (Subset von regulären Ausdrücken in Perl)
- Erschöpfende Diskussion in
  - Reguläre Ausdrücke, J. Friedl, 2007 (O'Reilly)

# Regex Operatoren (1)

- Suche erfolgt mit dem Match operator `m/regex/`

  Begrenzerregeln wie für q, qw,...(auch `m(),m[],m!!` ...)

  m kann entfallen: `/regex/` wenn der Begrenzer `/` ist
- Ersetzung erfolgt mit dem Substitute Operator `s/regex/string/`

  Begrenzer: 3 gleiche Zeichen oder paarweise Klammern:

  `s/a/b/; s(a)(b); s[a]{b}; s/a/(b);`
- Optionen modifizieren search/replace Operationen:

  z.B. `m/regex/i;` und `s/regex/string/g;`



# Regex Operatoren (2)

- Normalerweise wird in `$_` gesucht oder ersetzt
- Durch die Pattern binding Operatoren `=~` und `!~` ist das änderbar:

```
$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
(my $program = $0) =~ (.*/)(); # script name
```

- Das Resultat ist im skalaren Kontext true/false
- Im Listenkontext wird Liste gefundener/ersetzter Strings zurückgegeben

# Regex Elemente (1)

- Alle **ASCII Zeichen** (Metazeichen \|()[{^$*+?. mit \)
  ```
  /this is a dot in parentheses: \(\.\)/;
  ```
- **Abkürzungen** für spezielle Zeichen, z.B.:
  \a \e \n \cC \t (beep, ESC, newline, CTRL-C, tab)
- **Metazeichen** (Zeichen \|()[{^$*+?. sowie andere ASCII Zeichen mit vorangestelltem \)
  ```
  /^\d+ items/; #Metachars ^, \d and +
  ```
- **Klassen von Zeichen** Metazeichen oder `[..]`
  ```
  [yYjJ] [a-z0-9] \d
  ```
- **Alternativen** mit dem Zeichen |, z.B.: /quit|exit/



# Regex Elemente (2)

- Beginn(^), Ende ($) und andere String **Positionen**
  ```
  /^$/; # the empty string
  ```
- **Gruppierung** (...)
- Die **Wiederholungsfaktoren** ? * + {n} {min,max}
  ```
  /\d{1,3}/; # a number with 1..3 digits
  ```
- Das "**beliebige**" Zeichen . (nicht \n, siehe Option s)
- **Assertions** (erfüllte Bedingungen) beginnen mit (?

# Kommentare in Regexes

- Reguläre Ausdrücke sind relativ schlecht lesbar
- Inline Kommentierung mit

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

- Erweiterte Kommentierung mit Option **x**
  - Leerraum wird ignoriert, Kommentare mit #

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



# Klassen von Zeichen

- Match auf ein **einzelnes** Zeichen aus der Klasse
- Liste von Zeichen: [jJyY] or [^jJyY] (not jJyY)
- Bereiche von Zeichen: [0-9a-fA-F]
- Metazeichen bezeichnen Klassen von Zeichen:

| | | | |
|---|---|---|---|
| \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 |

- Seit perl 5.6: Unicode und POSIX Zeichenklassen

# Wiederholungsfaktoren

- vor einem Zeichen beschreiben, wie oft ein Zeichen (oder eine Zeichengruppe) matchen muss
- * 0 oder mehrfach, identisch zu {0,}
- ? 0 or 1 mal, identisch zu {0,1}
- + mindestens 1 mal, identisch zu {1,}
- {min,max} mindestens min, maximal max mal
- {n} genau n mal, identisch zu {n,n}
- {n,} mindestens n mal



# String Positionen

- Auch "zero width assertions" oder anchors (Anker)
- Häufig benutzt: Anfang `^` und Ende `$` vom String
  matchen mit Option m auch nach/vor Zeilenende
  `\A` und `\Z` matchen nur Beginn und Ende vom String
- `\b` matcht an einer Wortgrenze und `\B` nicht dort
- Was unterscheidet (\w+)\W und (\w+)\b ?

# Optionale Ausdrücke

- Ausdrücke, die mehrere Möglichkeiten zulassen
  - Alternativen mit |
  - Ausdrücke mit Faktoren: ?, *, +, {min,max} or {min,}
- Alternativen werden von links nach rechts bearbeitet
- Elemente mit Faktoren sooft als möglich probiert
  - solche Ausdrücke sind “greedy” (maximal matching)
  - Änderung des Verhaltens durch nachgestelltes ?
    - Dann werden optionale Elemente zunächst übergangen
    - heisst auch non-greedy oder minimal matching



# Maximales Matching

- Suche in `"Doris,Petra,Hera,Tesla"` mit Regex `/,.*,/ # greedy, maximal matching`
  - Suche nach erstem Komma (5 x falsch, 1 x wahr)
  - Suche nach .* (Bis zum Ende wahr, mit Alternativen)
  - Suche nach Komma (falsch, Stringende erreicht)
  - Nimm letzte Alternative (`a` , matche Komma, falsch)
  - Wiederholung (Backtracking) bis Komma gefunden
- daher lautet das Resultat: ”`,Petra,Hera,`”

# Minimales Matching

- Suche in `"Doris,Petra,Hera,Tesla"` mit Regex
  `/,.*?,/ # non-greedy, minimal matching`
  - Suche nach erstem Komma (5 x falsch, 1 x wahr)
  - Überspringe .*, suche Komma (falsch, `P` erreicht)
  - probiere 1x Alternative .* (`P`, erreicht, wahr)
  - suche Komma (falsch, `e` erreicht)
  - Wiederholung (Backtracking) bis Komma gefunden
- daher lautet das Resultat: ”`,Petra,`”
  `/,[^,]*,/ # non greedy, fastest Search`
  - Elemente in diesem Regex am häufigsten wahr



# Backtracking Probleme

- Regex Engine arbeitet stur nach diesen Regeln
- Regex Engine arbeitet effektiv, wenn Backtracking selten vorkommt. Geschachtelte optionale Ausdrücke wie in `(\d+)*a` sind problematisch
  - "123a" matcht in 5, "1234" schlägt nach vielen Schritten fehl
  - Schlecht geschriebene Regexe haben exponentielles Zeitverhalten, brauchen übermäßig Speicher und können Coredumps verursachen
  - Einige pathologische Regex werden durch Optimierer repariert
- **Vermeide geschachtelte Regex mit *, +, {..} etc.**

# Subpatterns

- Definiert durch runde Klammern um Teilausdrücke
- Innerhalb des Regex Rückbezug mit `\1, \2`...möglich
- Nach erfolgreichem Match Variablen `$1, $2 ...` gefüllt
- Ab perl 5.10 auch benannte match Variablen möglich
- `s/(?<letter>.)\k<letter>/$+{letter}/g`
- Keine Variablenzuweisung mit Assertions `(?...)`
- Klammern gruppieren (subpattern) und weisen zu `($1)`
- Falls nur Gruppierung gebraucht: `(?:subpattern)`




# Aufgaben

- Führen Sie die Statements aus. Warum ist $3 leer?

```
$_ = "Doris,Petra,Hera,Tesla";
/,(.*?),(.*?),(.*?)/; # $1='Petra',$2='Hera'
```

- Extrahieren Sie die Sekunden aus Zeilen der Form

```
$line='Date: Mon, 4 Feb 2008 13:57:05 +0100';
```

# Die Variablen $`, $& und $'

- Werden mit Teilen des Strings gefüllt :
  - `$`` : Erster Teil des Strings, der nicht matcht
  - `$&`: Teil des Strings, der Regex matcht
  - `$'`: Restlicher Teil des Strings
- Variablen nur gefüllt, falls mindestens 1x benutzt
  - Skript ist performanter, falls gar nicht benutzt
- Seit perl 5.6 können `@+` und `@-` benutzt werden
- Kein Geschwindigkeitsverlust, flexibler einsetzbar



# Die Variablen @+ und @-

- Diese Felder enthalten erste und letzte Position im String, wo die Patterns gematcht haben
- `$-[0]` Startposition des gesamten Matches
- `$+[0]` Endposition des gesamten Matches
- `$-[`*n*`]` `$+[`*n*`]` sind die entsprechenden Werte für `$`*n* d.h. das *n*te Subpattern, was gematcht hat
- Wenn in `$x` ein Match war, dann ist `$`` äquivalent zu

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

# Assertionen

- Ausdrücke (der Länge 0!) die matchen müssen
- Einfachste Assertions: Positionen wie ^, $ und \b
- Positive Lookahead Assertion `(?=pattern)`
  - Damit in greedy matches nicht zuviel gematcht wird
- Hier wird nicht behandelt:
  andere lookahead assertions (?=...) (?!...)
  lookbehind assertion (?<=...) (?<!...)
  und andere Ausdrücke (@!#^&!?$!!! ;-)



# Optionen

- Sowohl in Match als auch Replace Operatoren
- Bedeutung der Optionen (Option x siehe oben):
  - s - Behandle String als 1 Zeile ( . matcht \n)
  - m - Multizeilenstring (^ matcht nach, $ vor \n)
  - i - Ignoriere Klein/Großschreibung
  - g - Globales Suchen/Ersetzen **aller** Matches
  - o - Optimiere (kompiliere String nur einmal)
  - e - Behandle Ersetzungsstring als Ausdruck

# Regex Beispiele mit Optionen

```
$str = "Doris,Petra,Hera,Tesla";
# Option i
print "/pEtRA/i does match\n" if $str =~ /pEtRA/i;
# Option g und Zahl der Matches
$n = $str =~ s/,/\n/g;
print "$n Replacements:\n$str\n";
### from now on $str is a multiline string ###
# 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;
```



# Fehlersuche in regulären Ausdrücken

- `use re 'debugcolor';` kann hilfreich sein, um reguläre Ausdrücke zu verstehen (`perldoc re`)
- Es gibt Regex Debugger
  - Siehe z.B. http://weitz.de/regex-coach (neuere Versionen nur für Windows)
  - Proof of Concept: http://perl.plover.com/Rx (2001)
- Regexp Debugger in einigen IDEs (z.B.. Active State's Komodo) implementiert
- Hilfsmittel für Regex haben geringe Bedeutung

# Regex und split

- `split` is einzige Funktion, die Regex benutzt

  `@fields = split pattern, string`

  zerlegt den String gemäß *`pattern`*

  *`pattern`* schreibt man `'pat'`oder `/pat/`, nicht `"pat"`
- Leeres Pattern zerlegt *`string`* in einzelne Zeichen:

  `@digits = split //, "0123456789";`
- Zusätzliche Elemente können mit Subpattern erzeugt werden. Anwendungsbeispiel:

  Parsen von Configfiles (Zeilen der Form `key = val`)



# Parsen mit split

- Konfigurationsfile komplett in `$lines` enthalten

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

- Regexp zerlegt Zeile in zwei Teile (beim = )
- rechte Seite wird zusätzliches Element
- Liste (key1,val1,key2,...) wird als Hash interpretiert

# There is more than one way to do it (TIMTOWDI)

- Extrahiere Worte aus `$str = "abc def ghi jkl ";`
  - mit unpack

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

  - mit Regex

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

  - mit split

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

  - mit substr (Originalstring wird dabei zerstört)

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



# Regex Beispiele

- Entferne Leerzeichen an beiden Stringenden

```
s/^\s+|\s+$//;
s/^\s+//; s/\s+$//; # für Optimierer
```

- Name des PerlSkriptes (funktioniert auch für Win)

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

- Vertausche zwei Worte (Worttrenner: Leerzeichen)

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

# Der Operator tr (bzw. y) (Transliteration)

- Operator tr oder auch y benutzt keine Regex
  - Hat aber ähnliche Syntax wie Regex

  `tr /Zeichenfolge/Ersetzungsliste/Optionen`
  - Wird zum Konvertieren/Löschen von Zeichen benutzt
  - Option d: delete, c: complement, s: squash
  - Resultat ist Zahl der Ersetzungen, daher oft zu sehen

  ```
  $count = ( $var =~ tr/e/e/); #count letter e
  ```
- Wandeln Sie Texte mit CR/LF in Texte mit LF um



# Weitere Beispiele mit tr

- Lösche alle Zeichen, die nicht alphanumerisch sind

  ```
  tr/[0-9A-Za-z]//cd;
  ```
- Ersetze diese Zeichen stattdessen durch **ein** Leerz.

  ```
  tr/[0-9A-Za-z]/ /cs;
  ```
- Ver- und Entschlüsseln Sie Text mit Rot13 (a->n, ...)
- Erstellen Sie eine Liste der häufigsten Wörter in einem Text (Tipp: wie oben, Ersetzen durch LF, dann UNIX Befehle uniq -c und sort (-n) benutzen)

# Empfohlene Lektüre zu Regexp

- Die offizielle Referenz für Regex in Perl
  - `perldoc perlre`
  - und einige Abschnitte in `perldoc perlop`
- Tutorials
  - `perldoc perlpacktut`
  - `perlretut`
  - `perlrequick`



# Weitere Themen zu Regex

- Unicode und POSIX Unterstützung in Perl
  - beeinflusst die Art, wie man Regex schreiben kann
  - bessere Behandlung von Zeichensätzen (UTF-8!)
  - neue Klassen von Zeichen
- Dynamische Erzeugung von Regexs im Programm
- Nicht behandelte Regex Erweiterungen in perl 5.10

5-35

5-36