3.10 Reguläre Ausdrücke von Perl

3.10.1 Ersetzungen mit s///

Der **Ersetzungsoperator** `s///` arbeitet **ganz ähnlich wie** sein Gegenstück **bei** `sed`.

Der Ersetzungsoperator ist eine **normale Anweisung**:

▶▶ `view subst-statement.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
while(<STDIN>) {
  s/x/u/;
  print;
}
```

▶▶ `./subst-statement.pl`

(Eingabezeilen u.a. mit „Haxe“, „Taxe“.)

Der Ersetzungsoperator **arbeitet** per Default **auf** der **Default-Variablen** `$_`, genau wie `while(<STDIN>)` und `print`.

Für die Wahl anderer **Begrenzungszeichen** bietet `perl` mehr Möglichkeiten als `sed`:

```
s#^https://#http://#;

s{hugo}{karl};
s[hugo](karl);
```

Paarweise vorkommende Begrenzungzeichen werden anders behandelt.

3.10.2 Einschub: Kurzform für die while-print-Schleife

Für Programme wie das obige, die im wesentlichen aus einer **while-print-Schleife** bestehen, gibt es eine **Kurzform**:

▶▶ `view subst-statement2.pl` :

```
perl -p -e 's/x/u/'
```

ist äquivalent zu

▶▶ `view subst-statement3.pl` :

```
#!/usr/bin/perl
while(<>) {
  s/x/u/;
  print;
}
```

Diesen Trick hatten wir am Anfang des `perl`-Kapitels bereits verwendet, um **`sed`-artige**

Aufrufe von `perl` zu machen.

Anmerkung: Der **Diamant-Operator** `<>` arbeitet hier ähnlich wie der Operator `<STDIN>`. Näheres dazu kommt **später**.

Es gibt ebenfalls die **„-n“-Option** von `sed`. Sie ergibt die gleiche Schleife, aber ohne das implizite `print`:

▶▶ `view subst-statement4.pl` :

```
perl -n -e 'if(s/x/u/){print "ergibt: $_";}'
```

ist äquivalent zu

▶▶ `view subst-statement5.pl` :

```
#!/usr/bin/perl
while(<>) {
  if(s/x/u/) {
    print "ergibt: $_";
  }
}
```

▶▶ `./subst-statement4.pl`

Anmerkung: der Ersetzungsoperator `s///` **gibt die Anzahl der Ersetzungen zurück** (und den leeren String sonst).

3.10.3 Unterschiede zu `sed` und `grep`

Reguläre Ausdrücke von `perl`: **etwas andere Syntax**; mehr Möglichkeiten.

Die **alten Folien** zu `sed` **erweitert**:

Reguläre Ausdrücke von `sed` vs. `perl`

Gemeinsamkeiten:

Ausdruck
eine Folge von *Muster*n, optional mit *Zwischenraummarkierung*en dazwischen

Muster
ein Muster ist ein einzelnes *Zeichen* oder eine *Gruppierung*

Zeichen

c	einzelner Buchstabe: paßt auf sich selbst
.	Punkt: paßt auf jedes Zeichen außer dem Zeilenende
c	Backslash: macht aus Sonderzeichen normale Zeichen
[*Buchstaben*]	Menge von Buchstaben: paßt auf eines der Zeichen aus dieser Menge
	Buchstaben können auch Bereiche *von-bis* sein
	die Buchstaben „[“, „]“ und „-“ müssen ggf. ganz vorne oder hinten stehen
[^*Buchstaben*]	Komplementmenge von Buchstaben: paßt genau auf keines der Zeichen

Unterschiede bei `perl`:

Zeichen

c	Backslash: macht aus *allen* Sonderzeichen normale Zeichen

Kurs Unix-Tools, WS 01/02, Jan Bredereke 17

Reguläre Ausdrücke von `sed` vs. `perl` (Fortsetzung)

Gruppierung

`sed`	`perl`	
*Muster***	*Muster***	eine Folge von 0 oder mehr Mustern
Muster?	*Muster*?	0 oder einmal das Muster
Muster\+	*Muster*+	eine Folge von 1 oder mehr Mustern
Muster\{n\}	*Muster*{n}	eine Folge von n Mustern
Muster\{n,m\}	*Muster*{n,m}	eine Folge von n bis m Mustern
\(*Ausdruck*\)	(*Ausdruck*)	faßt Ausdruck zusammen; markiert Text für \\N
\\N	\\$N$ oder $$N$	das N-te mit Klammern eingeschlossene Muster
Muster\\|*Muster*	*Muster*|*Muster*	entweder das erste oder das zweite Muster

Zwischenraummarkierung

`sed`	`perl`	
^	^	Zeilenanfang
$	$	Zeilenende
\<	\b	Wortanfang
\>	\b	Wortende

Die **Backslash-Regeln müssen** bei `perl` **anders** sein, damit die **Regel** eingehalten werden kann, daß ein **Backslash *jedes* Zeichen** zu einem **normalen** Zeichen macht.

Man kann **sowohl `\1` als auch `$1`** benutzen, aber letzteres paßt besser zur sonstigen Variablensyntax. Man **darf** diese **Referenzen auch außerhalb** der regulären Ausdrücke als Variablen benutzen, aber dabei ist nur die zweite Form erlaubt.

Mächtigere reguläre Ausdrücke von `perl`

Zeichen

\w	Abkürzung für **[a-zA-Z0-9_]**	(alphanum. Zeichen)
\W	Abkürzung für **[^\w]**	(kein alphanum. Zeichen)
\s	Abkürzung für **[\t\n\r]**	(White-Space)
\S	Abkürzung für **[^\s]**	(kein White-Space)
\d	Abkürzung für **[0-9]**	(Ziffer)
\D	Abkürzung für **[^\d]**	(keine Ziffer)

Außerdem gibt es eine Reihe von fortgeschrittenen Erweiterungen mit **(?. . .)**
(z.B. Kommentar, Lookahead-Pattern, Bedingung, . . .)

Dokumentation dazu: auf der **perlre-Manpage**.

Kommentare in regulären Ausdrücken von `perl`:

`perl/latex-include1.pl`:

Kommentare in regulären Ausdrücken von `perl`

`sed`-Stil:

```
perl -n -e 'if(s/^[^%]*\\verbatimtabinput{([^}]+)}.*$/$1/){print;}'
```

Wenn der **x**-Modifikator angegeben ist, dann

- wird White-Space in regulären Ausdrücken ignoriert,
- ist # das normale Kommentarzeichen.

`perl/latex-include2.pl`:

Kommentare in regulären Ausdrücken von `perl` (Forts.)

Dasselbe mit `perl`-Kommentaren dokumentiert:

```
#!/usr/bin/perl -w
use diagnostics;
while(<STDIN>) {
  if( s/^       # Suche nur vollstaendige Zeilen (Zeilenanfang),
        [^%]*   # die vor dem entscheidenden Text keinen LaTeX-Kommentar haben
                # und die das richtige LaTeX-Kommando enthalten.
        \\verbatimtabinput{
        (       # Merke dir den Parameter des LaTeX-Kommandos.
        [^}]+   # Der Parameter darf alles ausser einer schliessenden Klammer
                # enthalten.
        )       # Ende des gemerkten Parameters.
        }       # Die schliessende Klammer des LaTeX-Parameters.
        .*$     # Beliebige weitere Zeichen bis zum Zeilenende.
                # Ersetze diese ganze Zeile durch den gemerkten
                # LaTeX-Parameter.
        /$1/x
  ) {
    print;      # Drucke nur, wenn eine Ersetzung stattgefunden hat.
  }
}
```


Die eigentlichen **Operatoren** bleiben zwar kryptisch, aber man kann sie erheblich besser dokumentieren und daher auch **später noch lesen**.

Anmerkung: Wir haben außerdem bessere Warnmeldungen hinzugefügt und den (noch nicht eingeführten) Diamant-Operator durch `<STDIN>` ersetzt.

3.10.4 Der Bindungsoperator `=~`

Die Textersetzung arbeitet **bisher immer auf** der **Default-Variablen `$_`**. Deswegen arbeitete sie **eben mit** dem Lesen von **STDIN und** der Ausgabe mit `print` zusammen.

Der **Bindungsoperator** kann das ändern:

▶▶ `view perl/bind-subst.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
$zeile = "Fahrt ins Gruene\n";
$zeile =~ s/Gruene/Blaue/;
print $zeile;
```

▶▶ `./bind-subst.pl`

3.10.5 Matchen mit `m//`

`sed` konnte dem Kommando ein **Suchmuster** voranstellen, so daß das Kommando **nur auf** den **passenden Zeilen ausgeführt** wurde. Das **Muster** stand **zwischen zwei Slashes**.

Das kann `perl` auch. **Hier** ist es ein **boolscher Ausdruck**.

▶▶ `view perl/match-cmd.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
print "Magst Du Perl? ";
$zeile = <STDIN>;
if ($zeile =~ /ja/) {
  print "Fein!\n";
} else {
  print "Schade.\n";
}
```

▶▶ `./match-cmd.pl`

Anmerkung: Man kann die Variable `$zeile` auch eliminieren.

Die Form mit den **zwei Slashes** ist **nur** die **Kurzform** des Match-Operators. Langform:

▶▶ `view perl/match-cmd2.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
print "Magst Du Perl? ";
while (<STDIN>)
{
  if ( m{ja/nein} ) {
    print "Dann entscheide Dich!\nMagst Du perl jetzt? ";
  } elsif ( m/ja/ ) {
    print "Fein!\n";
    exit;
  } else {
    print "Schade.\n";
    exit;
  }
}
```

▶▶ `./match-cmd2.pl`

Der **zweite reguläre Ausdruck** zeigt die **normale Langform**. Der **erste** zeigt, daß man bei der Langform **auch andere Begrenzungszeichen** nehmen darf, wenn Slashes ungünstig sind.

3.10.6 Modifikatoren

Modifikatoren für das **Matchen** und für das **Ersetzen**:

i	Case-**i**nsensitiv
s	. umfaßt auch Newline („**s**ingle line")
m	^ und $ matchen auch an Newline („**m**ultiple lines")
x	Kommentare erlaubt
g	(nur `s///`) ersetze **g**lobal alle Treffer

▶▶ `view perl/match-mod.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
$text = 'Erste Zeile
zweite zeile
3. Zeile
-------------------
';
print $text;
$text =~ s/zeil.*/Reihe/ ; # probiere Modifikatoren: i, ig, g
# $text =~ s/^Erste.*Zeile$/Erste Reihe/ ; # probiere Modifik.: m, ms, s
print $text;
```

▶▶ `./match-mod.pl`
Varianten durchspielen.

3.10.7 Interpolieren in Mustern

Muster werden **wie Doppelquote-Strings interpoliert**. Man kann also **Variablen** und **Escape-Zeichen** verwenden.

▶▶ `view perl/match-mod2.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
$text = 'Erste Zeile
zweite zeile
3. Zeile
-------------------
';
$von = 'zeil.*';
$nach = 'Reihe';
print $text;
$text =~ s/$von/$nach/ig ;
print $text;
$text =~ s/(Erste )(.*)/$1\U$2/s ;
print $text;
```

▶▶ `./match-mod2.pl`

3.10.8 Die Match-Variablen

Die Variablen `$1`, `$2`, `$3`, ... enthalten die **mit runden Klammern markierten Teile** des **letzten erfolgreichen Matches**. Dies gilt **sowohl** für den **Match**-Operator als auch für den **Ersetzungs**operator.

Man darf sie auch außerhalb des Operators verwenden:

▶▶ `view perl/match-var.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
$text = 'Ein Wort ist ein WORT ist ein Wort';
if( $text =~ /\b([A-Z]+)\b/ ) {
  print "Das grosse Wort ist '$1'.\n";
} else {
  print "Kein grosses Wort zu sehen.\n";
}
```

▶▶ `./match-var.pl`

Sie sind **bis** zum nächsten **erfolgreichen Match gültig**.

`sed` kannte nur `\1`, `perl` kennt sowohl `\1` als auch `$1`. `\1` bezieht sich auf den **gegenwärtig matchenden** Text, während `$1` sich auf den **letzten insgesamt erfolgreich matchenden** Text bezieht.

Beispiel:

▶▶ `view perl/match-var2.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
$text = "Affe Affe Affe Baer Chamaeleon\n";
print $text;
$text =~ s/(\w+) \1/Zikade Zikade/; # OK
print $text;
$text =~ s/(\w+) $1/Quagga Quagga/; # Wollen wir das wirklich???
print $text;
```

▶▶ `./match-var2.pl`

Bei der **zweiten Ersetzung matcht** das `$1` **auf** das **gemerkte Muster** aus der *vorigen* Ersetzung!

Anmerkung: Wenn man `\1` **auf der rechten Seite einer Ersetzung** verwendet, darf man das zwar im Prinzip, aber man bekommt eine **Warnung**.

Es gibt noch drei weitere, **„automatische“ Match-Variablen**.

Variable	**Bedeutung**
`$&`	der gesamte matchende Text
`` $` ``	der Text vor dem Match (Dollar-Backtick)
`$'`	der Text nach dem Match (Dollar-Apostroph)

▶▶ `view perl/match-var3.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
$text = "Diese Zeile ist langweilig, oder doch nicht?\n";
print $text;
if ( $text =~ /\w+,/ ) {
  print "$`\U$&\E$'";
} else {
  print "Kein Wort mit Komma danach gefunden.\n";
}
```

▶▶ `./match-var3.pl`

Anmerkung: **Merkhilfe** für `` $` `` und `$'`: Backtick und Apostroph sind die **öffnenden und schließenden Anführungszeichen** in LaTeX, die folglich ein Wort einrahmen.

3.10.9 Einschub: Langformen für kryptische Variablennamen

Langformen für kryptische Variablennamen

Kurzform	Langform	Bedeutung
`$_`	`$ARG`	die Default-Variable
`$&`	`$MATCH`	der gesamte matchende Text
`` $` ``	`$PREMATCH`	der Text vor dem Match (Dollar-Backtick)
`$'`	`$POSTMATCH`	der Text nach dem Match (Dollar-Apostroph)
`$$`	`$PID` *oder* `$PROCESS_ID`	Die Prozeßnummer des laufenden Perl-Programms
`$0`	`$PROGRAM_NAME`	Der Dateiname des Perl-Skripts (Dollar-Null)
`$.`	`$NR` *oder* `$INPUT_LINE_NUMBER`	Die aktuelle Zeilennummer der zuletzt gelesenen Eingabedatei
...		

Benutzung möglich nach:

`use English;`

Dokumentation:

`man perlvar`

Kurs Unix-Tools, WS 01/02, Jan Bredereke 22

Das **letzte Beispiel nochmal**, aber **mit Langformen** (und mit kommentiertem regulärem Ausdruck).

▶▶ `view perl/match-var4.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
use English;
$text = "Diese Zeile ist langweilig, oder doch nicht?\n";
print $text;
if ( $text =~ /\w+ # Suche ein Wort,
               ,   # das von einem Komma gefolgt wird.
              /x ) {
  print "$PREMATCH\U$MATCH\E$POSTMATCH";
} else {
  print "Kein Wort mit Komma danach gefunden.\n";
}
```

Ein Nachteil: *Alle* Regulären Ausdrücke werden **langsamer,** wenn irgendwo `use English;` verwendet wird.

3.10.10 Der `split`-Operator

`split`: **Teilt einen String** in eine Liste von Teilstücken **auf**, an den Stellen, wo ein bestimmter Separator im String steht.

Beispiel: Die bekannte `sed`-Aufgabe, die **Liste der „echten" Benutzer** aus /etc/passwd zu extrahieren:

▶▶ `view perl/passwd_realnames.pl` :

```
#! /usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
# In order to avoid warnings about unused variables, we declare them
# explicitly. (The latter will be explained later.)
my ($login, $passwd, $uid, $gid, $realname, $rest);
($login, $passwd, $uid, $gid, $realname, $rest) = split /:/ ;
if($uid >= 500 and $uid <= 599) {
  print "$realname\n";
}
```

▶▶ `./passwd_realnames.pl < ../sed/passwd`

Hier wird `split` **implizit auf die Variable `$_`** angewandt. Man kann `split` **auch** auf **andere Variablen** anwenden, und man kann auch andere Muster verwenden. **Alle regulären Ausdrücke** sind **erlaubt**.

Beispiel:

▶▶ `view perl/split-ws.pl` :

```
#! /usr/bin/perl -w
use diagnostics;
$text = "Dies     ist ein \t       Test.\n";
@felder = split /\s+/, $text;
print "Die Felder sind: ";
foreach $feld (@felder) {
  print "'$feld' ";
}
print "\n";
```

▶▶ `./split-ws.pl`

Was passiert, wenn wir das Muster `\s*` verwenden?

▶▶ `view perl/split-ws-ast.pl` :

```
#! /usr/bin/perl -w
use diagnostics;
$text = "Dies     ist ein \t       Test.\n";
@felder = split /\s*/, $text;
print "Die Felder sind: ";
foreach $feld (@felder) {
  print "'$feld' ";
}
print "\n";
```

▶▶ `./split-ws-ast.pl`

Wenn das **Muster auch den leeren String matcht**, wird nach *jedem* Buchstaben aufgeteilt.

Was passiert, wenn Felder leer sind?
▶▶ `view perl/split-leer.pl` :

```
#! /usr/bin/perl -w
use diagnostics;
$text = "::hallo:::";
@felder = split /:/, $text;
print "Die Felder sind: ";
foreach $feld (@felder) {
  print "'$feld' ";
}
print "\n";
```

▶▶ `./split-leer.pl`

Es gibt **vorne entsprechend Leerstrings** als Listenelemente.

Leere Felder hinten werden **nicht in die Liste** aufgenommen. **Vorteil:** Bei einer Zuweisung an eine Liste von skalaren Variablen ergibt das ggf. den Wert `undef` **für nicht belegbare Variablen**.

Wenn man nicht weniger Felder bekommen **will**, kann man als **dritten Parameter** die **Anzahl** der gewünschten Felder angeben. **Damit** bekommt man **auch nicht mehr** als die gewünschte Anzahl.

Sonderfall: Anstelle eines Musters ein **einzelnes Blank** (*ohne* die Slashes). Fast wie `\s+`, aber führende leere Felder werden unterdrückt.

Beispiel:
▶▶ `view perl/split-blank.pl` :

```
#! /usr/bin/perl -w
use diagnostics;
$text = "        Dies     ist ein \t       Test.\n";
@felder = split ' ', $text;
print "Die Felder sind: ";
foreach $feld (@felder) {
  print "'$feld' ";
}
print "\n";
```

▶▶ `./split-blank.pl`
Dies ist auch das Default als Muster.

3.10.11 Die `join`-Funktion

`join` benutzt **keine regulären Ausdrücke**. Aber sie ist das **Gegenstück** zu `split`. `join` **fügt eine Liste** von Strings **zu** einem **einzigen String** zusammen und fügt an den **Verbindungsstellen** einen festen String ein.

Beispiel:

▶▶ `view perl/join-date.pl` :

```
#! /usr/bin/perl -w
use diagnostics;
$date = '13/11/2001';
print "The British date is $date\n";
@felder = split /\//, $date;
$datum = join '.', @felder;
print "Das deutsche Datum ist $datum\n";
```

▶▶ `./join-date.pl`

3.11 Hausaufgabe

3.11.1 Aufgabe: Cross-Reference-Programm

Schreibe ein `perl`-Programm, das ein ein **`perl`-Programm einliest** und eine **Liste aller skalaren Variablen** (`$xy`) sowie eine **Liste aller Feldvariablen** (`@xy`) ausgibt. Es sollen **nur die alphanumerischen Namen** gesucht werden, nicht die kryptischen Kurzformen. `perl`-Kommentare sollen ignoriert werden, aber die besondere Behandlung von `$` und `@` innerhalb von einfachen Anführungsstrichen vernachlässigen wir.

Hinweise: Verwende die **zusätzlichen Konstrukte in regulären Ausdrücken**, die `perl` für alphanumerische Zeichen bereitstellt. Verwende zum **Zerlegen der Eingabe** die `split`-Operation. Benutze nach Möglichkeit auch den **Bindungsoperator**.