8.5 Sonstiges/Reguläre Ausdrücke

8.5.1 Reguläre Ausdrücke

Mit regulären Ausdrücken kann man nach Zeichenketten suchen, die zu bestimmten Regeln passen.

Eine ähnliche Form der Suche ist die Wildcard-Expansion von Pfadnamen (*filename globbing*): Unter MS-Windows löscht man mit

Terminal

```
C:\> del *.exe
```

alle Dateien, deren Dateiendung `.exe` ist[1].

Die Programme `sed`, `grep` und `awk` benutzen ebenfalls Such-Ausdrücke, allerdings nicht für Pfadnamen, sondern für Texte. Deshalb haben sie ein etwas anderes Regelwerk, das unter dem Namen *POSIX regular expressions* bekannt ist[2].

8.5.2 *POSIX regular expressions*

a) Der Punkt passt auf jedes Zeichen: `B.ch` auf `Buch` und `Bach`.

b) Mit eckigen Klammern kann man eine Auswahl von erlaubten Zeichen darstellen: `[Bb]ach` passt auf `Bach` und `bach`.

c) Das Minuszeichen in eckigen Klammern gibt einen Bereich an: `[0-9][0-9]` passt auf alle zweistelligen Zahlen.

d) Mit dem Zirkumflex direkt nach der öffnenden eckigen Klammer kann man die Auswahl umkehren: `[^\n]` erlaubt jedes Zeichen außer `\n`.

e) Der Stern erlaubt das vorherige Zeichen null bis beliebig viele Male: `10*` erlaubt 1, 10, 100, 1000 usw.

f) Der Zirkumflex steht für Zeilenanfang: `^-` findet alle Zeilen, die mit - anfangen.

g) Das Dollarzeichen steht für Zeilenende: `R$` findet alle Zeilen, die mit `R` enden.

h) Bei den erweiterten *POSIX regular expressions* erlauben die runden Klammern eine Gruppierung: `([0-9][A-Z])*` findet alle Zeichenketten, bei denen abwechselnd eine Ziffer und ein Buchstabe folgt.

i) Alle diese Sonderzeichen sind durch ein Backslash zu maskieren, wenn sie als normale Zeichen benutzt werden sollen, wenn man z.B. nach Zeichenketten suchen will, die selbst Sterne oder eckige Klammern enthalten.

8.5.3 Unterschied zum *filename globbing*

Zuerst ein paar Beispiele:

Muster	Zeichenkette	Ergebnis
`*nerv*`		ist als Muster nicht erlaubt
`*nerv`		ebenso nicht erlaubt
`nerv`	`genervt`	match
`nerv*`	`genervt`	ebenso, Stern erlaubt beliebig viele `v`

Das heißt: Es reicht, wenn das Suchmuster irgendwo in der Zeichenkette auftaucht.

Soll es spezifisch am String- bzw. (bei `REG_NEWLINE`) Zeilenanfang gesucht werden, so muss es `^nerv` heißen. Am String- bzw. Zeilenende muss es `nerv$` heißen.

[1] Unter den Linux-Shells wie `bash` ist diese Wildcard-Expansion in verbesserter Form bereits eingebaut und steht damit allen Programmen zur Verfügung

[2] Die Programmiersprache `perl` bietet unter dem Namen PCRE ein erweitertes Regelwerk an. Man muss sich nur für eines entscheiden

Hier noch einmal als Gegenüberstellung:

Reg. Ausdruck	Wildcards
`nerv`	`*nerv*`
`nerv$`	`*nerv`
`^nerv`	`nerv*`
`^nerv$`	`nerv`
`n.rv`	`*n?rv*`

Vergleichen kann man die beiden Systeme, wenn man in einem Verzeichnis einmal `ls -1|grep n.rv` (Reg. Ausdruck durch das Programm `grep`) und einmal `ls -1 *n?rv*` (Wildcard-Expansion der Shell) aufruft.

8.5.4 Reguläre Ausdrücke in C

Ein Beispiel, wie man in C nach regulären Ausdrücken suchen kann, zeigt `src/passt.c`:

```
1  #include <stdio.h>
2  #include <regex.h>
3  #define LEN 80
4  #define LENSTR "80"
5  int main(void)
6  {
7      int rc;
8      regex_t muster_tr;
9      char muster[LEN+1]="";          /* z.B. B.CH */
10     char zeichenkette[LEN+1]=""; /* z.B. BACH */
11
12     printf("Eingabe Muster (z.B. >>B.CH<<):");
13     scanf("%" LENSTR "[^\n]", muster);
14     while(getchar()!='\n'){}
15
16     rc=regcomp(&muster_tr, muster, REG_EXTENDED|REG_ICASE);
17     if(rc)
18     {
19         char fehlermeldung[LEN+1]="";
20         regerror(rc, &muster_tr, fehlermeldung, LEN);
21         printf("Fehler: %s\n", fehlermeldung);
22         return 1;
23     }
24
25     printf("Eingabe Zeichenkette (z.B. >>BACH<<):");
26     scanf("%" LENSTR "[^\n]", zeichenkette);
27     while(getchar()!='\n'){}
28
29     rc=regexec(&muster_tr, zeichenkette, 0, NULL, 0);
30     printf("Ergebnis (0=gefunden): %d\n", rc);
31
32     regfree(&muster_tr);
33     return 0;
34 }
```

Zeile 2 Die Headerdatei wird eingebunden.

Zeile 8 In einer Variable wird Platz für ein Suchmuster bereitgestellt, und zwar in einer Form, die für wiederholtes Suchen (mit verschiedenen Texten) bezüglich Effizienz besser geeignet ist als ein String.

Zeile 16 `regcomp()` übersetzt das Muster vom String in diese Form. Hier im Beispiel werden erweiterte *POSIX regular expressions* verwendet (`REG_EXTENDED`) und es wird absichtlich nicht nach Groß- und Kleinschreibung unterschieden (`REG_ICASE`), so dass auch `bucH` auf `B.CH` passt.

Zeile 20 Falls `regcomp()` fehlschlägt, liefert `regerror()` eine detaillierte Fehlermeldung.

Zeile 29 Mit `regexec()` wird das Muster in der Zeichenkette gesucht. Bei Erfolg gibt die Funktion eine 0 zurück. Die drei letzten Parameter erlauben zusätzliche Möglichkeiten (finden der Treffer im Text), die hier nicht gebraucht werden.

Zeile 32 `regfree()` gibt den Platz, der für das übersetzte Suchmuster belegt ist, wieder frei.

Terminal

```
schueler@debian964:~$ ./passt
Eingabe Muster (z.B. >>B.CH<<):b.ch
Eingabe Zeichenkette (z.B. >>BACH<<):BUCH
Ergebnis (0=gefunden): 0
schueler@debian964:~$ ./passt
Eingabe Muster (z.B. >>B.CH<<):b.ch
Eingabe Zeichenkette (z.B. >>BACH<<):Zitrone
Ergebnis (0=gefunden): 1
```

Im nächsten Beispiel sucht man nach den Stellen im Text, an der das Suchmuster auftritt (`src/such.c`):

```
1  #include <stdio.h>
2  #include <regex.h>
3  #define LEN 80
4  #define LENSTR "80"
5  #define ANZAHL 1
6  int main(void)
7  {
8     int rc, offset;
9     regex_t muster_tr;
10    regmatch_t treffer[ANZAHL];
11    char muster[LEN+1]="";
12    char zeichenkette[LEN+1]="";
13    char fehlermeldung[LEN+1]="";
14
15    printf("Eingabe␣Muster:");
16    scanf("%" LENSTR "[^\n]", muster);
17    while(getchar()!='\n'){}
18
19    rc=regcomp(&muster_tr, muster, REG_EXTENDED|REG_ICASE);
20    if(rc)
21    {
22       regerror(rc, &muster_tr, zeichenkette, LEN);
23       printf("Fehler:␣%s\n", zeichenkette);
24       return 1;
25    }
26
27    printf("Eingabe␣Zeichenkette:");
```

```
28      scanf("%" LENSTR "[^\n]", zeichenkette);
29      while(getchar()!='\n'){}
30
31      offset=0;
32      rc=regexec(&muster_tr, &zeichenkette[offset], ANZAHL, treffer, 0);
33      while(rc==0)
34      {
35          if(treffer[0].rm_so==-1)
36              printf("ungenutzt\n");
37          else
38              printf("Treffer: %.*s\n",
39                      treffer[0].rm_eo - treffer[0].rm_so,
40                      &zeichenkette[offset]+treffer[0].rm_so);
41
42          offset = offset + treffer[0].rm_eo;
43          rc=regexec(&muster_tr, &zeichenkette[offset], ANZAHL, treffer, 0);
44      }
45      printf("Keine (weiteren) Treffer.\n");
46      regfree(&muster_tr);
47      return 0;
48  }
```

Zeile 10 Hier wird eine Variable eingerichtet, die die Treffer eines `regexec()`-Aufrufs speichert. Es interessiert *hier* zu einem Treffer immer nur der ganze gefundene String[3], daher reicht für unser Array in diesem Fall ein Element.

Zeilen 31–32 und 42–43 Anders als bei `strtok()` muss man hier als Nutzer den zu durchsuchenden String selbst im Auge behalten: Nach jedem Finden einer Fundstelle muss die Suche hinter der Fundstelle weitergehen. Dazu wird die Variable `offset` nach jedem Fund um den End-Offset der Fundstelle (relativ zum Anfang) erhöht. Man hätte auch einen Zeiger nehmen können.

Zeile 32 Die Variable `treffer` nimmt hier den Treffer auf.

Zeilen 35–36 Fehlschlag im Zusammenhang zu Untermustern – hier uninteressant

Zeile 38 Der Text befindet sich zwischen `rm_so` und `rm_eo` (Start- und End-Offset), relativ zu `&zeichenkette[offset]`. Die komplizierte `printf()`-Anweisung versucht das nur möglichst kurz zu fassen: `printf("%*s",20,s)` gibt maximal 20 Zeichen von s aus.

Zeile 33 Wenn `regexec()` nichts mehr findet, gibt es einen Wert ungleich null zurück.

Terminal

```
schueler@debian964:~$ ./such
Eingabe Muster:b.ch
Eingabe Zeichenkette:Die Fachbuchhandlung liegt am Bach.
Treffer: buch
Treffer: Bach
Keine (weiteren) Treffer.
```

[3] im Suchmuster könnten mit runden Klammern abgeteilte Untermuster existieren; zu jedem Untermuster bekommt man dann einen Untertreffer.