# Charsets oder „Warum funktionieren meine Umlaute nicht?“

## Einführung

Jeder hat es schon mindestens einmal erlebt: Ein Programm, das mit Text arbeitet, funktioniert wunderbar, solange man keine Umlaute eingibt. Sonst kommt nur noch Zeichenmüll heraus und ein bis zwei nicht korrekt dargestellte Zeichen pro Umlaut.

## ASCII

Um zu verstehen, warum es dazu kommt, muss man sich anschauen, wie „normaler“ Text und wie Umlaute binär abgespeichert werden.

Angefangen hat es 1963 mit ASCII, einem Standard, der 128 Zeichen je eine Zahl von 0 bis 127 zuweist, die mit 7 bit kodiert werden können.

Festgelegt sind die Zahlenwerte für lateinische Buchstaben, Ziffern, Satzzeichen und Kontrollzeichen wie „Carriage Return“ und „Line Feed“, also Zeilenumbrüche. Zeichen, die im Alltag eines Amerikaners nicht vorkommen, wie die deutschen Umlaute, kyrillische Zeichen und vieles mehr, wurden ausser Acht gelassen. Da ein Byte aus 8 Bits besteht, ist bei ASCII das erste, „most significant“ Bit immer 0.

## Andere Zeichenkodierungen

Als man in Europa anfing Computer zu benutzen, mussten die benötigten Zeichen irgendwie im Computer gespeichert werden und dazu benutzte man die verbleibenden 128 Zeichen pro Byte. So entstanden die Kodierungen Latin-1 für den westeuropäischen Raum, Latin-2 für Mitteleuropa und so weiter, auch bekannt als ISO-8859-1 und ISO-8859-2.

Diese Zeichensätze stimmen in den ersten 128 Zeichen mit ASCII überein, die zweiten 128 Zeichen, also die mit 1 als erstem Bit, unterscheiden sich untereinander.

Die Grenzen dieser Zeichensätze werden einem schnell anhand eines gar nicht so alten Beispieles klar: Mit der Einführung des Euros hatten viele Länder eine neue Währung und damit ein Währungssymbol, das sich nicht in den traditionellen Zeichensätzen ausdrücken ließ! (Dieses Problem wurde durch das Einführen des Zeichensatzes ISO-8859-15 behoben, der sich nur wenig von Latin-1 unterscheidet und das €-Zeichen enthält).

## Unicode

Die bisherigen Zeichenkodierungen konnten jeweils nur einen kleinen, lokal sinnvollen Bereich aller möglicher Zeichen darstellen - sobald man Texte mit gemischten Zeichensätzen verfassen wollte, ging das heillose Chaos los.

Um etwas Ordnung in das Chaos zu bekommen, hat das Unicode-Konsortium damit angefangen, jedem Zeichen, das in irgend einer Schrift in irgend einer Sprache vorkommt, eine eindeutige, ganze Zahl und einen Namen zuzuordnen.

Die Zahl heißt „Codepoint“ und wird üblicherweise als vier- oder sechsstellige, hexadezimale Zahl in der Form `U+0041` notiert; der dazugehörige Name wäre `LATIN SMALL LETTER A`.

Neben Buchstaben und anderen „Basiszeichen“ gibt es auch Akzentuierungen wie den z.B. `ACCENT, COMBINING ACUTE`, die auf den vorherigen Buchstaben einen Akzent setzen.

Wenn auf ein Basiszeichen eine Akzentuierung oder andere kombinierende Zeichen folgen, bilden mehrer Codepoints ein logischen Buchstaben, ein sogenanntes Grapheme.

### Unicode Transformation Formats

Die bisher vorgestellten Unicode-Konzepte stehen vollständig unabhängig davon, wie die Unicode-Zeichen kodiert werden.

Dafür wurden die „Unicode Transformation Formats“ definiert, Zeichenkodierungen, die alle möglichen Unicode-Zeichen darstellen können. Der bekannteste Vertreter ist UTF-8, das für die bisher vergebenen Codepoints 1 bis 4 Bytes benötigt.

Auch in UTF-8 stimmen die ersten 128 Zeichen mit denen von ASCII überein.

Von UTF-8 gibt es auch eine laxe Variante, UTF8 (ohne Bindestrich geschrieben), die mehrere mögliche Kodierungen für ein Zeichen zulässt. Das Perl-Modul `Encode` unterscheidet diese Varianten.

UTF-16 dagegen benutzt für jedes Zeichen mindestens zwei Byte, für sehr hohe Unicode-Codepoints werden auch hier mehr Bytes benötigt.

UTF-32 kodiert jedes mögliche Zeichen mit vier Bytes.

Beispiele für Zeichenkodierungen (siehe Tabelle 1).

## Perl und Zeichenkodierungen

Strings können in Perl entweder als Bytestrings oder als Textstrings vorliegen. Wenn man z.B. eine Zeile aus `STDIN` liest, ist sie per Default ein Bytestring.

Mit der Funktion `decode` des Moduls `Encode` kann man sie in einen Textstring umwandeln, wenn man die Zeichenkodierung kennt. In Gegenrichtung, also von Textstring nach Bytestring konvertiert man mit `encode` aus dem gleichen Modul.

Alle Textoperationen sollte man auf Textstrings durchführen, weil so auch Nicht-ASCII-Zeichen korrekt behandelt werden: `lc` und `uc` funktionieren wie erwartet, und `\w` in regulären Ausdrücken passt auf jeden Buchstaben, auch auf Umlaute, ß und allen möglichen Zeichen in allen möglichen Sprachen, die dort als Bestandteil eines Wortes angesehen werden.

`cmp` vergleicht Nicht-ASCII-Zeichen allerdings nur dann so, wie man das erwartet (also `"ä" lt „b"`), wenn `use locale` aktiv ist. Da das Verhalten von `sort` durch `cmp` definiert ist, gilt dies auch für das Sortieren von Listen (siehe Listing 1).

```
#!/usr/bin/perl
use warnings;
use strict;
use Encode qw(encode decode);

my $enc = 'utf-8';
# in dieser Kodierung ist das
# Script gespeicher
my $byte_str = "Ä\n";

# Bytestrings:
print lc $byte_str;
# gibt 'Ä' aus, lc hat nichts verändert

# Textstrings:
my $text_str = decode($enc, $byte_str);
   $text_str = lc $text_str;
# gibt 'ä' aus, lc hat gewirkt
   print encode($enc, $text_str);
```

*Listing 1*

Es empfiehlt sich, alle Eingaben direkt in Textstrings umzuwandeln, dann mit den Textstrings zu arbeiten, und sie erst bei der Ausgabe (oder beim Speichern) wieder in Bytestrings umzuwandeln. Wenn man sich nicht an diese Regel hält, verliert man im Programm schnell den Überblick welcher String ein Textstring und welcher ein Bytestring ist.

Perl bietet mit den IO-Layern Mechanismen, mit denen man die Kodierungen per Dateihandle oder global automatisch umwandeln lassen kann (siehe Listing 2).

| Codepoint | Zeichen | ASCII | UTF-8 | Latin-1 | ISO-8859-15 | UTF-16 |
|---|---|---|---|---|---|---|
| U+0041 | A | 0x41 | 0x41 | 0x41 | 0x41 | 0x00 0x41 |
| U+00c4 | Ä | - | 0xc4 0x84 | 0xc4 | 0xc4 | 0x00 0xc4 |
| U+20AC | € | - | 0xe3 0x82 0xac | - | 0xa4 | 0x20 0xac |
| U+c218 | 수 | - | 0xec 0x88 0x98 | - | - | 0xc2 0x18 |

*Tabelle 1: Beispiele Zeichenkodierung*

```
# IO-Layer: $handle liefert beim Lesen jetzt Textstrings:
open my $handle, '<:encoding(UTF-8)', $datei;

# das gleiche:
open my $handle, '<', $datei;
binmode $handle, ':encoding(UTF-8)';

# Jedes open() soll automatich :encoding(iso-8859-1) benutzen:
use open ':encoding(iso-8859-1)';

# Alle Stringkonstanten werden als utf-8 interpretiert
# und in Textstrings umgewandelt:
use utf8;

# Schreibe Text mit der aktuellen locale nach STDOUT:
use PerlIO::locale;
binmode STDOUT, ':locale';
# alle Lese-/Schreibeoperation mit aktueller locale:
use open ':locale';
```

*Listing 2*

```
#!/usr/bin/perl
use warnings;
use strict;
use Encode;

my @charsets = qw(utf-8 latin1 iso-8859-15 utf-16);

my $test = "Ue: " . chr(220) . "; Euro: " . chr(8364) . "\n";

for (@charsets){
       print "$_: " . encode($_, $test);
}
```

*Listing 3*

```
use Devel::Peek;
use Encode;
my $str = "ä";
Dump $str;
$str = decode("utf-8", $str);
Dump $str;
Dump encode('latin1', $str);
__END__
SV = PV(0x814fb00) at 0x814f678
      REFCNT = 1
      FLAGS = (PADBUSY,PADMY,POK,pPOK)
      PV = 0x81654f8 "\303\244"\0
      CUR = 2
      LEN = 4
SV = PV(0x814fb00) at 0x814f678
      REFCNT = 1
      FLAGS = (PADBUSY,PADMY,POK,pPOK,UTF8)
      PV = 0x817fcf8 "\303\244"\0 [UTF8 "\x{e4}"]
      CUR = 2
      LEN = 4
SV = PV(0x814fb00) at 0x81b7f94
      REFCNT = 1
      FLAGS = (TEMP,POK,pPOK)
      PV = 0x8203868 "\344"\0
      CUR = 1
      LEN = 4
```

*Listing 4*

Mit Vorsicht sollte man den Input Layer `:utf8` genießen, der annimmt, dass die Eingabedatei gültiges UTF-8 ist. Sollte sie das nicht sein, ist das eine potentielle Quelle für Sicherheitslücken (siehe `http://www.perlmonks.org/?node_id=644786` für Details).

Als Output Layer dagegen ist `:utf8` anstelle von `:encoding(UTF-8)` problemlos verwendebar.

Das Modul und Pragma `utf8` erlaubt es auch, Nicht-ASCII-Zeichen in Variablennamen zu verwenden. Da das jedoch kaum getestet ist und zum Teil mit Modulenamen und Namespaces nicht gut funktioniert, sollte man Variablennamen weiterhin nur aus lateinischen Buchstaben, dem Unterstrich _ und Ziffern zusammensetzen.

## Die Arbeitsumgebung testen

Ausgestattet mit diesem Wissen kann man testen, ob das Terminal und locales auf die gleiche Kodierung eingestellt sind, und auf welche (siehe Listing 3)

Wenn man dieses Programm in einem Terminal ausführt, wird nur eine Textzeile korrekt angezeigt werden, die erste Spalte darin ist dann die Zeichenkodierung des Terminals.

Wie vorher gesagt ist das Eurozeichen € nicht in Latin-1 vorhanden, das `Ü` sollte in einem Latin-1-Terminal trotzdem richtig angezeigt werden.

## Troubleshooting

Die Art, wie perl Textstrings intern speichert, führt zu etwas ungewöhnlichem Verhalten wenn diese ausgegeben werden. Wenn alle Codepoints in dem String kleiner als 256 sind, wird der String intern als Latin-1 gespeichert, und als solcher ausgegeben.

Sind Codepoints größer gleich 256 in einem String vorhanden, speichert perl den String als UTF-8, und wenn man sie mit `print` ausgibt (und wenn `warnings` aktiv sind), kommt die Warnung `Wide character in print`.

Daher empfiehlt es sich, eigene Scripte mit Zeichen zu testen, die nicht in Latin-1 vorhanden sind, z.B. das Euro-Zeichen €. Dann sieht man immer, wenn man ein `Encode::encode` an enstprechender Stelle vergessen hat, die oben genannte Warnung.

Wenn Daten aus externen Module kommen, z.B. aus `DBI`, kann man mit `utf8::is_utf8(STRING)` überprüfen, ob ein String als Textstring abgespeichert ist. Diese Abfrage liefert aber nur ein sinnvolles Ergebnis, wenn Zeichen außerhalb des ASCII-Zeichensatzes in dem String enthalten sind.

Auch `Devel::Peek` zeigt an, ob ein String intern als Textstring oder als Bytestring vorliegt (siehe Listing 4).

Der String `UTF8` in der Zeile `FLAGS =` zeigt, dass der String als Texttring vorliegt. In der Zeile `PV =` sieht man bei Textstrings die Bytes und in Klammer eckigen Klammern die Codepoints.

Weitere Probleme können durch fehlerhafte Module entstehen. So ist die Funktionalität des Pragmas `encode` sehr verlockend:

```
# automatische Konvertierungen:
use encoding ':locale';
```

Allerdings funktionieren unter dem Einfluss von `use encoding` AUTOLOAD-Funktionen nicht mehr, und das Modul funktioniert nicht im Zusammenspiel mit Threads.

## Kodierungen im WWW

Beim Schreiben von CGI-Scripten muss man sich überlegen in welcher Kodierung die Daten ausgegeben werden sollen und das enstprechend im HTTP-Header vermerken.

Für die meisten Anwendungen empfiehlt sich UTF-8, da man damit einerseits beliebge Unicode-Zeichen kodieren kann, andererseits auch deutschen Text platzsparend darstellen kann.

HTTP bietet zwar mit dem `Accept-Charset`-Header eine Möglichkeit herauszufinden, ob ein Browser mit einer Zeichenkodierung etwas anfangen kann, aber wenn man sich an die gängigen Kodierungen hält, ist es in der Praxis nicht nötig, diesen Header zu prüfen.

Für HTML-Dateien sieht ein Header typischerweise so aus: `Content-Type: text/html; charset=UTF-8`. Wenn man einen solchen Header sendet, muss man im HTML-Code nur die Zeichen escapen, die in HTML eine Sonderbedeutung haben (`<`, `>`, `&` und innerhalb von Attributen auch `"`).

Beim Einlesen von POST- oder GET-Parametern mit dem Modul `CGI` muss man darauf achten, welche Version man benutzt: In älteren Versionen liefert die `param`-Methode immer Bytestrings zurück, in neueren Versionen (ab 3.29) werden Textstrings zurückgegeben, wenn vorher mit `charset` die Zeichenkodierung UTF-8 eingestellt wurde - andere Kodierungen werden von `CGI` nicht unterstützt.

Damit Formularinhalte vom Browser mit bekanntem Zeichensatz abgeschickt werden, gibt man im Formular die `accept-charset`-Entity mit an:

```
<form method="post" accept-charset="utf-8"
      action="/script.pl">
```

Bei Verwendung eines Template-Systems sollte man darauf achten, dass es mit charsets umgehen kann. Beispiel sind `Template::Alloy`, `HTML::Template::Compiled>` (seit Version 0.90) oder Template Toolkit in Verbindung mit `Template::Provider::Encoding`.

## Weiterführende Themen

Mit den Grundlagen zu den Themen Zeichenkodierungen und Perl kommt man schon sehr weit, zum Beispiel kann man Webanwendunen „Unicode-Safe“ machen, also dafür sorgen, dass alle möglichen Zeichen vom Benutzer eingegeben und dargestellt werden können.

Damit ist aber noch längst nicht alles auf diesem Gebiet gesagt. Der Unicode-Standard erlaubt es beispielsweise, bestimmte Zeichen auf verschiedene Arten zu kodieren. Um Strings korrekt miteinander zu vergleichen, muss man sie vorher „normalisieren“. Mehr dazu gibt es in der Normalisierungs-FAQ: `http://unicode.org/faq/normalization.html`.

Um landesspezifisches Verhalten für Programme zu implementieren, lohnt es, die locales genauer anzusehen, ein guter Einstiegspunkt ist das Dokument `perllocale`.

# Moritz Lenz

TPF-TICKER

## Perl 5.10 veröffentlicht

Pünktlich zum 20. Geburtstag von Perl (18. Dezember 2007) hat Pumpking Rafael Garcia-Suarez die neue Perl-Version auf CPAN veröffentlicht. Mit Perl 5.10 gibt es nach über 5 Jahren wieder ein „großes“ Update von Perl. Dabei wurden viele neue Features implementiert, die zum Teil aus der Perl6-Entwicklung stammen. Auch an der Engine für die Regulären Ausdrücke wurde komplett neu gestaltet, so dass jetzt auch ziemlich einfach Grammatiken definiert werden können.

## Chris Dolan beendet Perl::Critic-Grant

Im Laufe der Zeit wurden 20 neue Policies für Perl::Critic umgesetzt. Dabei hat sich Chris Dolan an „Perl Best Practices“ orientiert. Während der Entwicklung wurden einige Bugs in PPI gefixt und neue Features eingeführt. Der Grant lief seit April 2007 und hatte einen Wert von 2000 US$.

Während der Zeit hat Chris Dolan Perl::Critic an unterschiedlichen Stellen promoted. Unter anderem war er Sprecher auf der WebGUI Users Conference.