Unicode Issues in Perl

Meir Guttman

e-mail: meir@guttman.co.il

A Few Unicode Facts

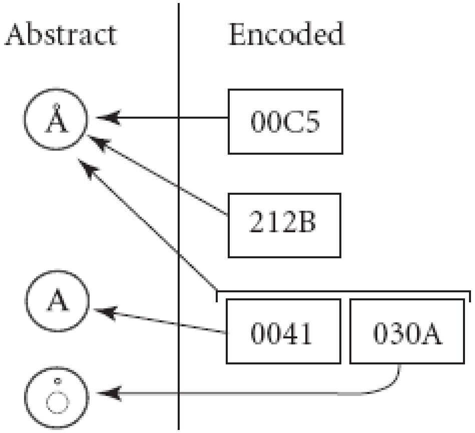
- ✰ Not “characters”, but “Code-Points” in the range of U+00’0000 to U+10’FFFF
- ✰ Short designation: U+hhhh
- ✰ Includes a “code point” for each and every conceivable character in all conceivable “scripts”:
  - ● “Scripts”, as opposed to “Languages”. For example, Chinese, Japanese and Korean share the same script.
  - ● 93 scripts as of v. 6.0, including for example Egyptian Hieroglyphs
  - ● Numbers, General Punctuation, General Symbols, Mathematical Symbols, Musical Symbols, Technical Symbols, Dingbats, Arrows, Braille Patterns and more
- ✰ Hebrew occupies code points U+0590 to U+05FF

A Few Unicode Facts (cont.)

- ☆ Unicode includes rules for the support of Bi-Directional (Bi-Di) text
- ☆ However, when "Unicode support" is claimed, it does not imply Bi-Di support, and it seldom does!
  - According to some claims, the official Unicode Bi-Di algorithm sucks…
- ☆ It supports the notion of a “paragraph” and a forced new-line (i.e., one that doesn’t terminate a paragraph)
- ☆ It supports all kind of text directions
  - LTR, RTL, and one embedded within the other (Bi-Di),
  - Top-to-bottom, bottom-up with “lines” going either from left-to-right or right-to-left
  - Boustrophedon: early Greek and Egyptian hieroglyphs used it. It Literally means “ox-turning”

A Few Unicode Facts (cont.)

- ✫ Supports both fully formed and superimposed diacritics ("ניקוד" - פחות או יותר) on a bare base code-point



Encodings

- ✰ “Encoding” only applies to I/O and files:
  - ● Text files
  - ● Downloaded Internet pages
  - ● Software source code (hence strings in it)
  - ● Text streams
  - ● etc.
- ✰ It is *not* (necessarily) how it is coded in memory
- ✰ Databases, editors, compilers, etc. can *read and/or write* (e.g. UTF-8) Unicode encoded text, but it doesn’t necessarily mean that they *internally* represent text as “encoded” Unicode!
- ✰ Current encodings are only UTF-8, UTF-16 and UTF-32
- ✰ Older, deprecated encodings are UCS-2, UCS-4 and UTF-7)
- ✰ Practically, I never encountered anything other then UTF-8…

Encodings (cont.)

- UTF-8:
  - variable length encoding, 1-4 bytes.
  - code-points in the range 0-127 are identical to “pure” ASCII encoding (please note, 7-bit ASCII, not 8-bit Latin-1!)
  - Encoding:

Code Points U+xx xxxx	1st Byte	2nd Byte	3rd Byte	4th Byte
0aaa aaaa	0aaaaaaa			
0000 0bbb bbaa aaaa	110bbbbb	10aaaaaa		
cccc bbbb bbaa aaaa	1110cccc	10bbbbbb	10aaaaaa	
000d ddcc cccc bbbb bbaa aaaa	11110ddd	10cccccc	10bbbbbb	10aaaaaa

  - Hebrew UTF-8 encoding is therefore in the range of 0xD690 to 0xD7BF

Encodings (cont.)

- ✰ UTF-16 encoding
  - Variable length, one or two 16-bits units
  - Code points in the range U+0000..U+FFFF are represented as a single 16-bit code unit.
  - This range contains the vast majority of common-use characters for all modern scripts of the world.
  - Lookup “Unicode surrogate code points” for further details.
- ✰ UTF-32: the simplest one of all, where each code point is directly represented by a single 32-bit unit (word).
- ✰ One must know beforehand for the last two encodings on what “Endianess” was it originated, otherwise it would be impossible to interpret it.
- ✰ A Byte-Order-Mark (BOM) of U+FFFE, as the first code-point provides such a clue.

Canonical Equivalence

- ✫ Unicode heroically tried to be as backward compatible as possible with previous “locals” and “code-pages”.
- ✫ What made life difficult was:
  - Diacritics: the lowercase letter "ñ" of the Spanish alphabet can be set as either:
    - *A single code point U+00F1, or*
    - *code point U+006E (Latin lowercase "n") followed by U+0303 (the combining tilde "◌̃")*
- ✫ Code point sequences that are defined as **canonically equivalent** are assumed to have the **same appearance and meaning** when printed or displayed.
- ✫ Those sequences should be displayed in the same manner, should be treated in the same way by applications such as sorting or searching, and may be substituted for each other.

Compatibility Equivalence

- Sequences assumed to have possibly distinct *appearances*, but the same *meaning* in some contexts.
- For example, the code point U+FB00 (the typographic ligature "ﬀ") is defined to be compatible — but not canonically equivalent — to the sequence U+0066 U+0066 (two Latin "f" letters).
- Compatible sequences may be treated the same way in some applications (such as sorting and indexing), but not in others
- They may be substituted for each other in some situations, but not in others.
- Sequences that are canonically equivalent are also compatible, but the opposite is not necessarily true.

Normalization

- Unicode string searches and comparisons in text processing software must take into account the presence of equivalent code points.
- In the absence of this feature, users searching for a particular code point sequence would be unable to find other visually indistinguishable glyphs that have a different, but canonically equivalent, code point representation.
- **Unicode normalization** replaces equivalent sequences of characters so that any two texts that are equivalent will be reduced to the same sequence of code points.
- Unicode defines two normal forms:
  - **A fully composed** one, where multiple code points are replaced by single points whenever possible;
  - A **fully decomposed** one, where single points are split into multiple ones. Each of these four normal forms can be used in text processing.

Normalization (cont.)

- ✰ Unicode provides standard normalization algorithms (plural…!)
- ✰ These produce a *unique* (normal) code point sequence for all sequences that are *equivalent*
- ✰ Unicode defined four normalization “forms” (next slide)
- ✰ All four were implemented by Perl package (so I am told…):

(From the Unicode cookbook)

```
use Unicode::Normalize;
while (<>) { $_ = NFD($_); # decompose + reorder canonically
  ...
} continue { print NFC($_); # recompose (where possible) + reorder canonically
 }
```

Unicode normalization forms

NFD *Normalization Form (Canonical) Decomposition*	Characters are decomposed by canonical equivalence, and multiple combining characters are arranged *in a specific order*.
NFC *Normalization Form (Canonical) Composition*	Characters are decomposed and then recomposed by canonical equivalence.
NFKD *Normalization Form Compatibility Decomposition*	Characters are decomposed by compatibility, and multiple combining characters are arranged *in a specific order*.
NFKC *Normalization Form Compatibility Composition*	Characters are decomposed by compatibility, then recomposed by canonical equivalence.

Perl Support

✰ “Support” means dealing in Perl with Unicode in:

- Strings
- Text I/O
- Regular Expressions
- Normalization

Perl Strings

- ✫ As of Perl 5.8.1, the Perl native internal representation of strings is Unicode.
- ✫ You will find in many, even in canonical, Perl documents that this representation is UTF-8.
- ✫ There are some indications that this is “almost true” (as in “almost dead”???)
- ✫ If it is indeed true, then IMHO it is not wise. Think of all the overhead required to decipher the actual length in bytes…
- ✫ The good part is that you don’t need to know its internal representation!
- ✫ Even more, you should *never use or rely* on its internal structure. Here today, gone tomorrow…

Perl Strings (cont.)

- ✰ To make Perl’s recognize Unicode strings, you *must* insert the pragma:

```
Use utf8;
```

- ✰ Once you did that, you can use Unicode strings as you would any other string: e.g., one can do translation as follows:

```
my %ID_types = (
  'מספר תעודת זהות'                => 'IL_ID',
  'מספר ברשם החברות בישראל'       => 'IL_CorpID',
  'מספר ברשם'                     => 'IL_CorpID',
  'מספר ברשם השותפויות בישראל'    => 'IL_PartnerID',
  'מספר דרכון'                    => 'Passport',
  'מספר רשם בארץ ההתאגדות בחו"ל'  => 'ForeignCorpID',
  'מספר ביטוח לאומי'              => 'SSN',
  'מספר מזהה אחר'                 => 'OtherID'
);

$Eng_ID_type = $ID_types{'מספר ברשם'};
```

An Important Note

- ✫ Even if :
  - Your source script is saved in Unicode/UTF-8 (or other) encoding,
  - It looks right in your Unicode/UTF-8/whatever supporting editor,
  - The encoding specification in the "open" statement of the output file is UTF-8 (…),
  - You placed a 'binmode' statement with its (optional) encoding as Unicode/UTF-8
- ✫ It ***Will NOT*** produce legible Unicode (e.g. Hebrew) text in the output file,
- ✫ UNLESS a «`use utf8;`» pragma is specified!!!
- ✫ Again, I am referring to strings *embedded in the script code*!
- ✫ Unicode text read as such from a file or downloaded from a web site and then written out and properly I/O encoded will be fine. Only strings in the body of the code require the pragma!

The Perl UTF8$^{(*)}$ flag

- ✰ Internally, Unicode strings are encoded as either ISO-8859-1 or UTF8.
- ✰ A flag, called "SvUTF8", a.k.a. "the UTF8 flag", is set to 1 for strings that are UTF-8 internally, and to 0 for strings that are ISO-8859-1
- ✰ Once the UTF8 flag is set, Perl does not check the validity of the UTF8 sequences further. *This might be a security breach*
- ✰ The :utf8 PerlIO layer sets the UTF8 flag, without checking the byte sequences, on incoming data.
- ✰ This is not a bug or a flaw, but the very function of this PerlIO layer.
- ✰ It is used internally by other layers (most importantly the :encoding layer), after they have (safely) converted the input to UTF8.
- ✰ So, for your own protection, instead of the :utf8 PerlIO layer, use :encoding(UTF8) or :encoding(UTF-8)

* This was taken from a semi-official source. It should have been ‘Unicode’ rather than ‘UTF-8

Unicode Collation

- ✰ The Unicode Collation Algorithm (UCA) defines several levels of collation strength:
  - Level 1: ignoring case and diacritics, examining basic characters only
  - Level 2: adds diacritic comparisons to the ordering algorithm
  - Level 3: adds case ordering
  - Level 4: adds a tiebreaking comparison (sorry, can’t explain… ☹)

  Level 4 is the default
- ✰ In simple terms, you can use collation strength to tell a UCA-aware sort to ignore case or diacritics.

```
use Unicode::Collate;
my $col = Unicode::Collate->new(level => 1);
my @list = $col->sort(@old_list);
```

Perl Unicode I/O

- Declaring I/O default encodings:

```
use open OUT => ":encoding(UTF-8)";
use open IO  => ":encoding(iso-8859-7)";
```

  (Importing non-Unicode text to a Unicode processing environment)

- Or, on an “open” by “open” basis:

```
open(my $fh, "<:encoding(windows-1255)", $filename) or die"$!\n";
```

  - This also avoids the “`Wide character in print…`” warnings
  - There are other good reasons to use this 3-arguments version of “open”

- To avoid “Wide character in print…” warnings in STDOUT and STDERR, you are advised to place

```
binmode STDOUT, ":encoding(UTF-8)“;
```

  (which will nevertheless send garbage to a ‘cmd’ window when emitting Hebrew text)

DBI and Unicode

```
my %conn_attrs = (RaiseError=> $RaiseError,
                 PrintError => $PrintError,
                 AutoCommit => $AutoCommit,
                 mysql_enable_utf8 => 1);

my $dbh = DBI->connect
           ($dsn, $user_name, $password, \%conn_attrs);
```

Hebrew HTML page scrapping example

- We need to download a Hebrew HTML file, `windows-1255` encoded, and to build an `HTML::TreeBuilder` object from it. We start by:

```
my $root = HTML::TreeBuilder->new();
```

- Although there is a TreeBuilder method `$root->new_from_file($filename)` to do it directly…
- … it assumes a default UTF-8 encoding.
- It will therefore not work with a `windows-1255` encoded file!
- Rather, one must first "open" the file, thus giving us the opportunity to specify its encoding and use another method to parse it:

```
open(my $fh, "<:encoding(windows-1255)", $filename);
$root->parse_file($fh);
```

(This method can accept either a file-name or a file-handle)

Unicode Regular Expressions

- The Unicode Consortium specified three levels of RegEx support, “Basic”, “Extended” and “Tailored”, see Technical Standard #18.
- Perl versions supports most of the first and very little of the other two
- Perl 14 (supposedly) added more support
- You can usually use Unicode strings as RegEx patterns
- Unicode defines :
  - Character names (e.g., “HEBREW LETTER ALEF”)
  - Character properties (e.g., “Lowercase_Letter”)
  - Script names (e.g., “Tamil”)
- You can specify all these by an escape `\p{}` and `\P{}`, e.g.:
  - `\p{Hebrew}` (any Hebrew character)
  - `\P{HEBREW POINT HOLAM}` (any character except one with a חולם)

More…

- "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)“ by Joel Spolsky
  http://joelonsoftware.com/articles/Unicode.html
- The Unicode Standard
  http://www.unicode.org/
  (easy reading, all 670 pages of it…)
- Unicode Standard Annex #9 Unicode Bidirectional Algorithm
  http://www.unicode.org/reports/tr9/
- Perl Unicode Tutorial
  http://perldoc.perl.org/perlunitut.html
- Unicode support in Perl
  http://perldoc.perl.org/perlunicode.html
- Perl Unicode FAQ
  http://perldoc.perl.org/perlunifaq.html

and even more…

- Analyzing Unicode Text with Regular Expressions by Andy Heninger (IBM Corporation)
  http://icu-project.org/docs/papers/iuc26_regexp.pdf
- UTF8 related exploit (PerlMonks post)
  http://www.perlmonks.org/?node_id=644786
- **Unicode and Passwords** by Ovid
  http://blogs.perl.org/users/ovid/2012/02/unicode-and-passwords.html
- **Why Unicode Normalization Matters** by chromatic
  http://www.modernperlbooks.com/mt/2013/01/why-unicode-normalization-matters.html

- **And, for a day-to-day work, until you are versed, use** Tom Christiansen’s **Perl Unicode Cookbook**
  http://www.perl.com/pub/2012/04/perlunicook-standard-preamble.html