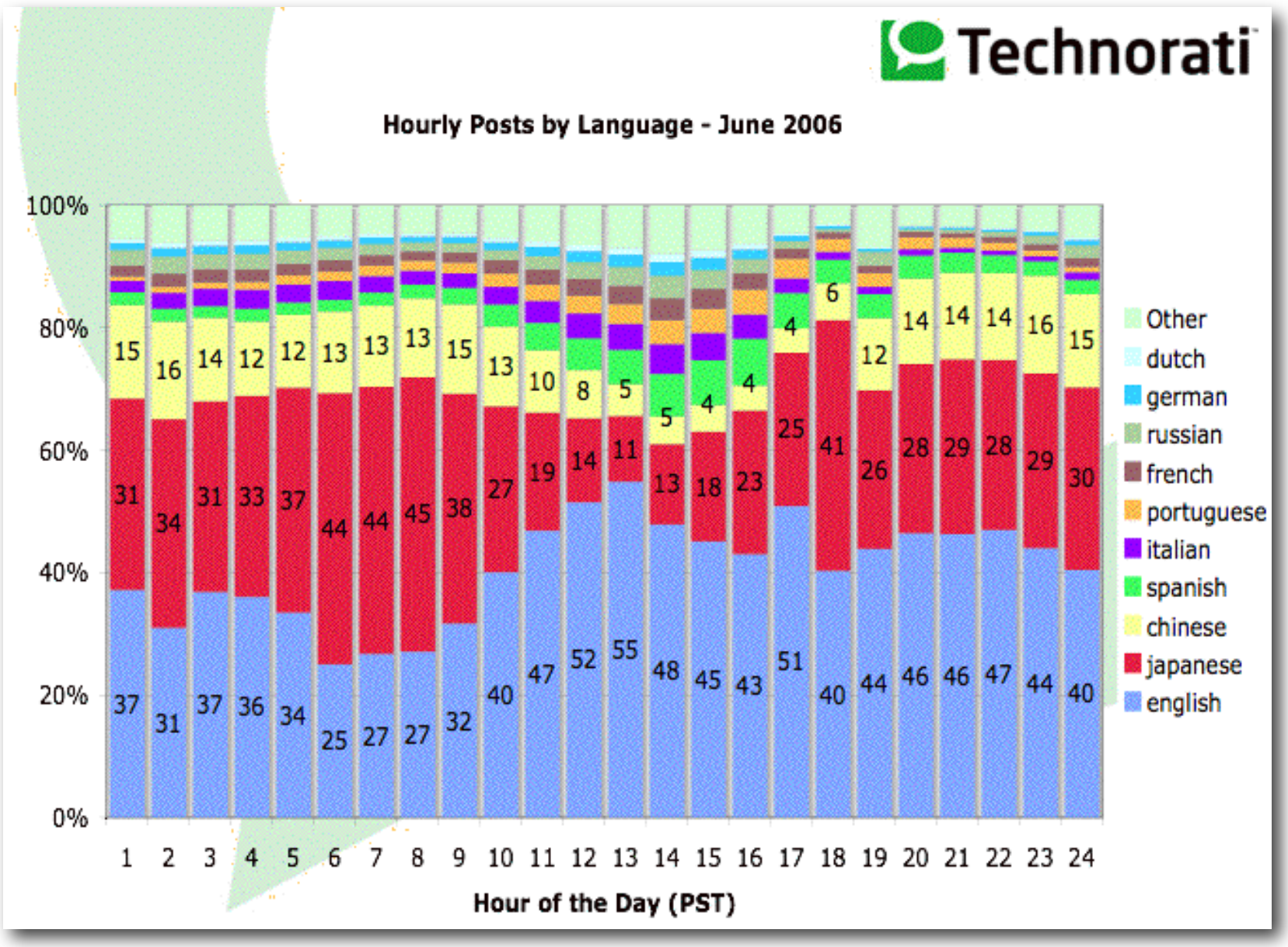
THE WORLD IN 30 MINUTES

Tim Bray
RubyKaigi 2007
Sun Microsystems

I've been processing text for a living since early 1987: OED, search engine, Japanese tokenizer, XML, Atom; so if someone wants to jump up and say "you're wrong" about something ruby-related, well, there's a good chance I am. On the other hand, when it comes to text processing and internationalization and Unicode and so on, I'm prepared to go on arguing incredibly fine points of technical detail until your arm falls off, and I'll even



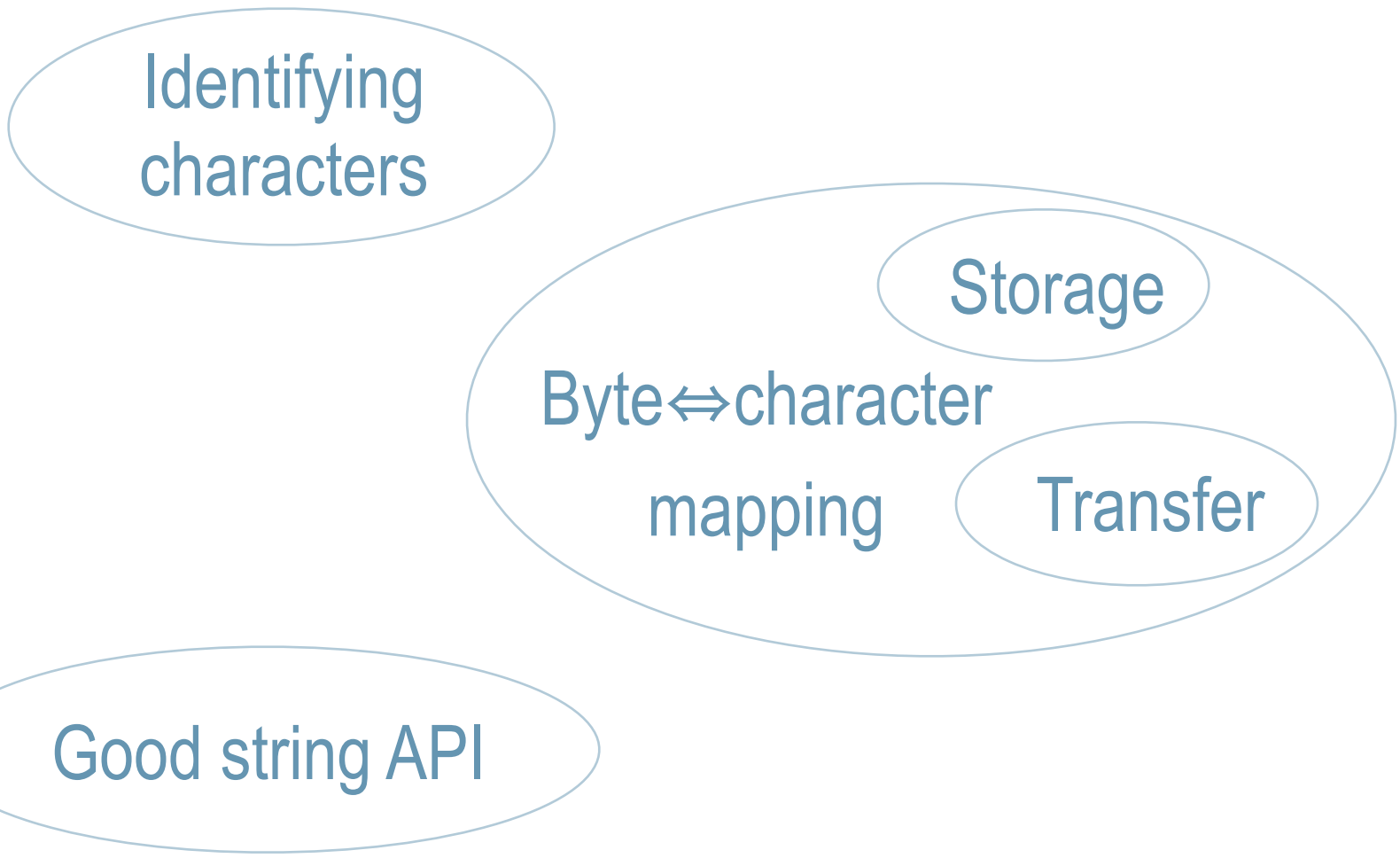
This graph was made by Dave Sifry of Technorati. It averages over all the days in June 2006. English is no longer the majority language of the Internet. Two languages are substantially under-reported here, French and Korean, because of the immense popularity in both countries of proprietary blogging servers that Technorati can't track. In 2007, it is no longer acceptable to do i18n-oblivious computing. It's bad

```
/[a-zA-Z]+/
```

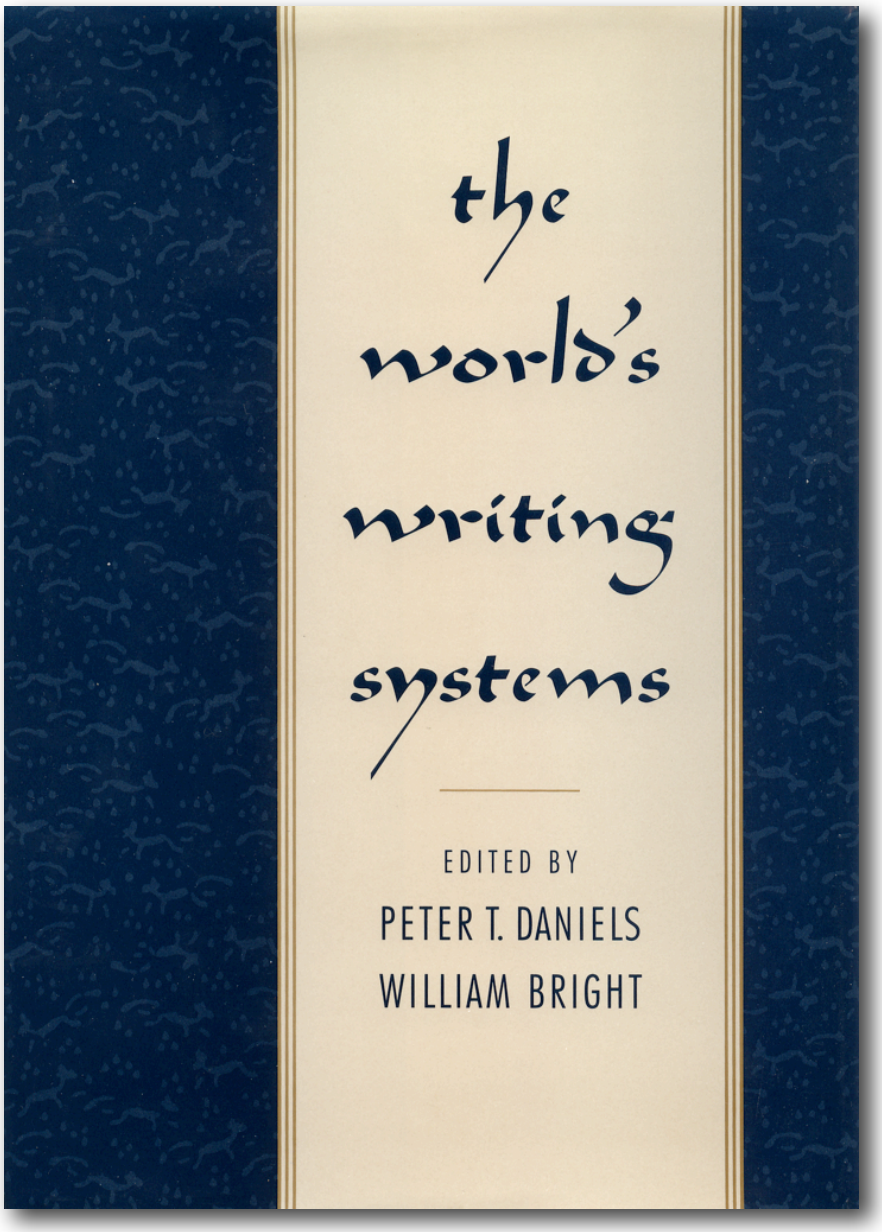
This is probably a bug.

I think this is stolen from Larry Wall

The Problems We Have To Solve



To do a good job, here are the three important problems. First of all, we have to agree on how we're going to identify and describe characters. Then while characters may be abstract, computers aren't; they can only store and interchange is bytes, we have to agree how we're going to express these characters as byte sequences. There are two sides of this problem; storing characters on RAM or disk, and interchanging them on the network.



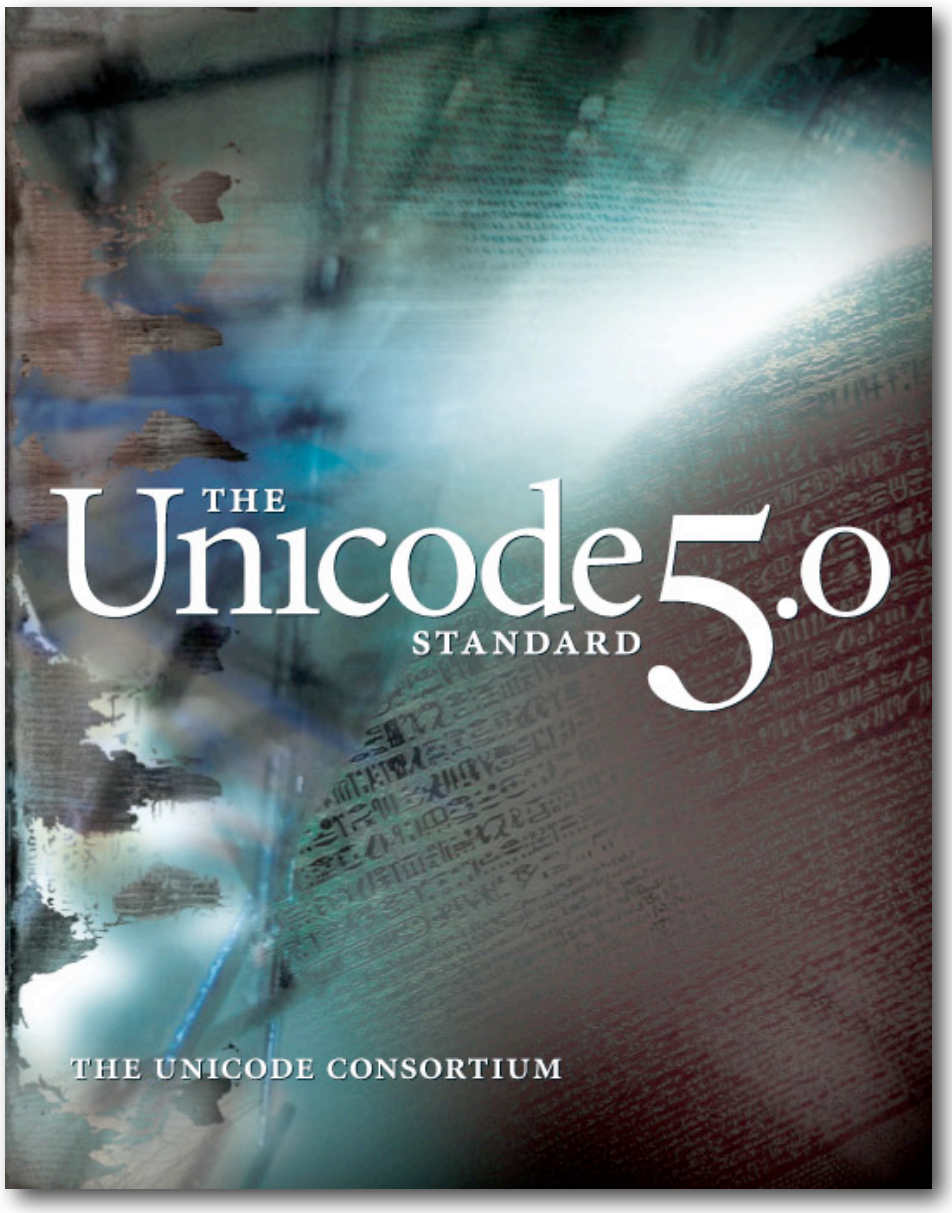
Published in 1996; it has 74 major sections, most of which discuss whole families of writing systems.

If you want to really understand the problem, start here. There aren't that many areas of human scholarship where there is one definitive book; but I think this one may qualify. If you love language and writing and text, you might want to get this just to read for fun.

www.w3.org/TR/charmod

This is Unicode-centric, but its lessons are generally applicable to anyone who is trying to do network computing and using a lot of text.

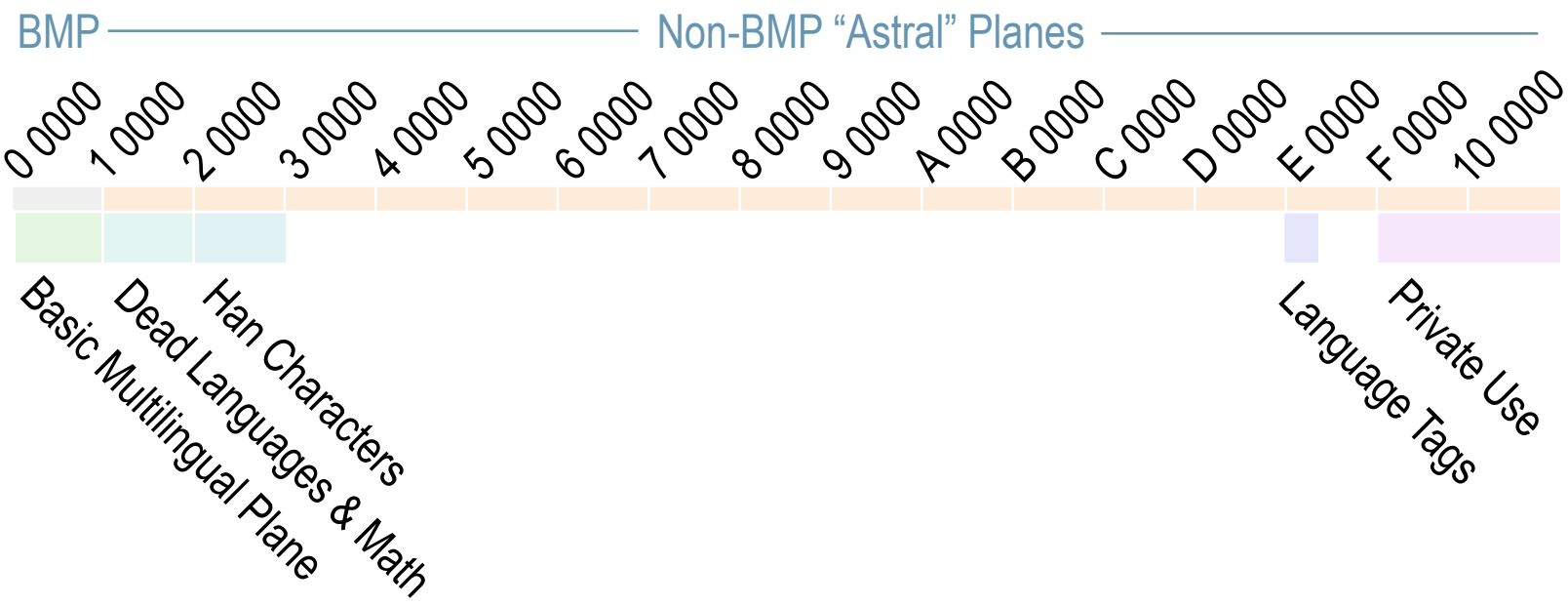
Identifying Characters



This is the largest and most ambitious attempt to make the universe of the world's characters useful to computer programmers and users. It's not perfect, but it's still an impressive piece of work. I strongly recommend that anyone who's going to be doing serious text processing buy this and keep it around. It's immensely big and thick, but 80% of it is just tables of all the characters. There are a few short chapters in the front quarter of the

1,114,112 Unicode Code Points

17 “Planes” each with 64k code points: U+0000 – U+10FFFF

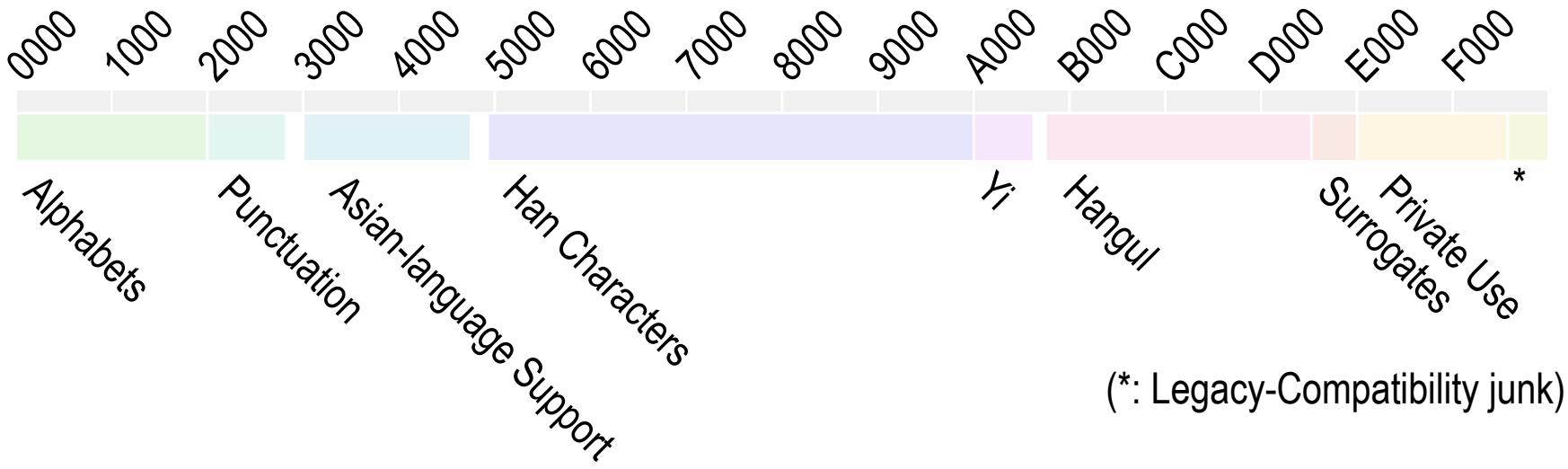


99,024 characters defined in Unicode 5.0

Unicode characters are identified by numbers called “code points”, and they’re always given in hex with U+ in front. The available space is about 8.9% full. Humans have created less than 250,000 characters in all of history, so if we’re typical, we should be OK for the next three alien races we meet. The original idea of Unicode was that characters would be 16-bit; the people who had this idea were Americans and Europeans who didn’t

The Basic Multilingual Plane (BMP)

U+0000 – U+FFFF



(*: Legacy-Compatibility junk)

27,484 Han characters, 11,172 Hanguls, 6,400 private-use chars

Unicode Character Database

```
00C8;LATIN CAPITAL LETTER E WITH
GRAVE;Lu;0;L;0045 0300;;;;N;LATIN
CAPITAL LETTER E GRAVE;;;00E8;
```

È

"Character #200 is LATIN CAPITAL LETTER E WITH GRAVE, a lower-case letter, combining class 0, renders L-to-R, can be composed by U+0045/U+0300, had a different name in Unicode 1, isn't a number, lowercase is U+00E8."

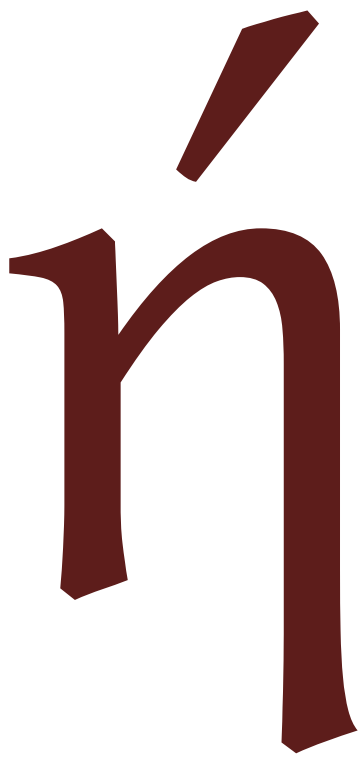
www.unicode.org/Public/Unidata

What does Unicode know about a character, aside from its number? This is in a bunch of semicolon-delimited ASCII files, and it's kind of complicated, but quite usable. ActiveSupport::MultiByte uses it. I usually have it loaded in an Emacs browser buffer. Note the combining-form issue; there are two separate ways to encode "È". Note that some of these properties are potentially useful in regular expressions.

U+0024 DOLLAR SIGN

U+017D LATIN CAPITAL LETTER Z WITH CARON

U+00AE REGISTERED SIGN

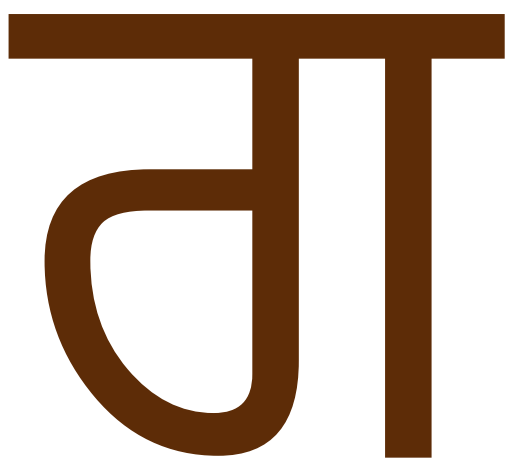


U+03AE GREEK SMALL LETTER ETA WITH TONOS

U+0416 CYRILLIC CAPITAL LETTER ZHE

U+05D0 HEBREW LETTER ALEF

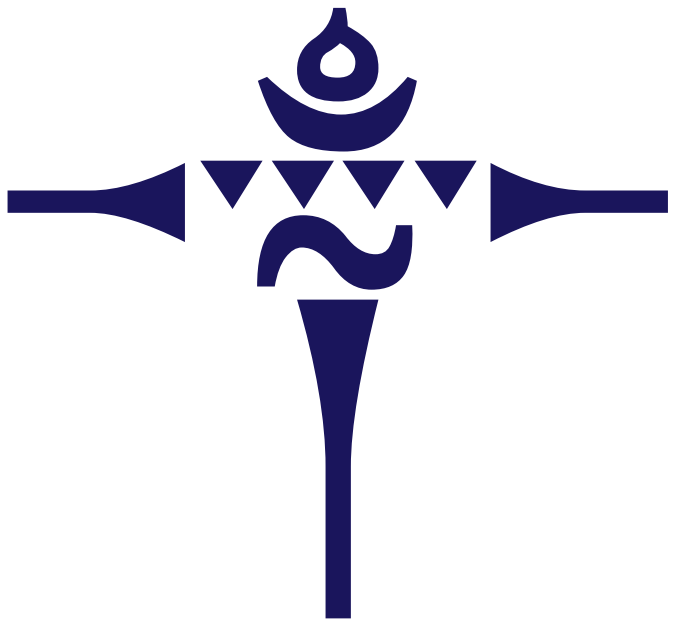
U+0638 ARABIC LETTER ZAH



U+0A17 GURMUKHI LETTER GA

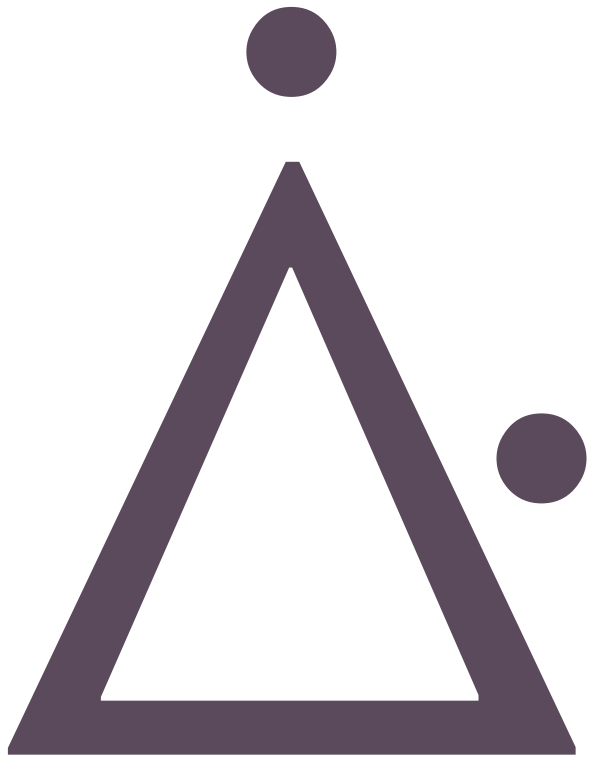
U+0A88 GUJARATI LETTER II

U+0E06 THAI CHARACTER KHO RAKHANG

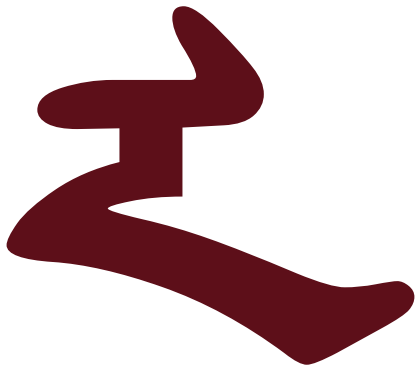


U+0F12 TIBETAN MARK RGYA GRAM SHAD

U+13BA CHEROKEE LETTER ME



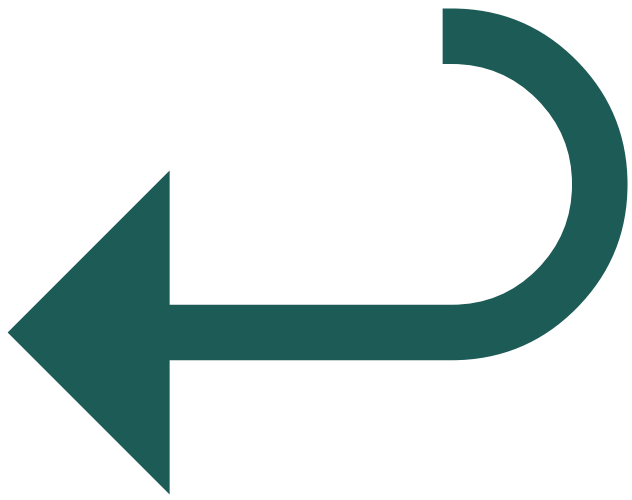
U+1411 CANADIAN SYLLABICS WEST-CREE WII



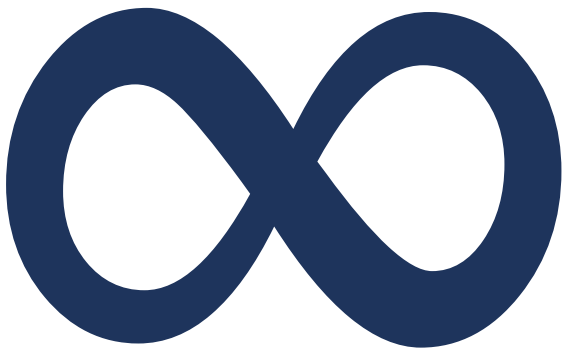
U+1820 MONGOLIAN LETTER ANG

U+2030 PER MILLE SIGN

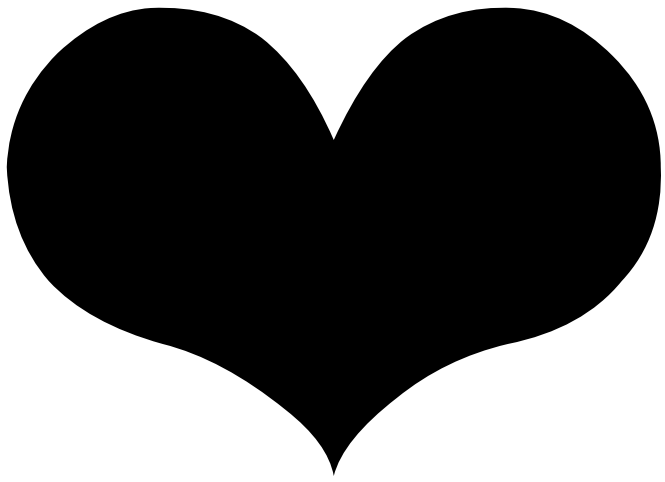
U+215D VULGAR FRACTION FIVE EIGHTHS



U+21A9 LEFTWARDS ARROW WITH HOOK



U+221E INFINITY



U+2764 HEAVY BLACK HEART

U+3055 HIRAGANA LETTER SA

U+30C0 KATAKANA LETTER DA

U+4E2D (Han character)

U+8A9E (Han character)

U+AC7A (Hangul syllabic)

U+1D12B (Non-BMP) Musical Symbol Double Flat

U+2004E (Non-BMP) (Han character)

Nice Things About Unicode

Huge repertoire
Room for growth
Private use areas
Sane process
Unicode character database
Ubiquitous standards/tools support

Private use areas: NTT Docomo emoji, European legislation special sorts. The IETF and the W3C require that all new Internet and Web protocols support the Unicode character set and the UTF-8 encoding, so if you can't do Unicode, you can't be part of the Net.

Difficulties With Unicode

Combining forms
Awkward historical compromises
Han unification

Combining forms: Early Uniform Normalization. Historical problem is that every one of the original character sets that was harvested to produce Unicode contained stupid design errors, and most had to be preserved. Finally, there is the process of Han unification, in which a team of Asian scholars chose single Unicode code points for originally-Chinese characters that are used separately in Chinese, Japanese, and Korean, but are

Han Unification

Pro: en.wikipedia.org/wiki/Han_Unification

Contra: tronweb.super-nova.co.jp/characcodehist.html

Neutral: www.jbrowse.com/text/unij.html

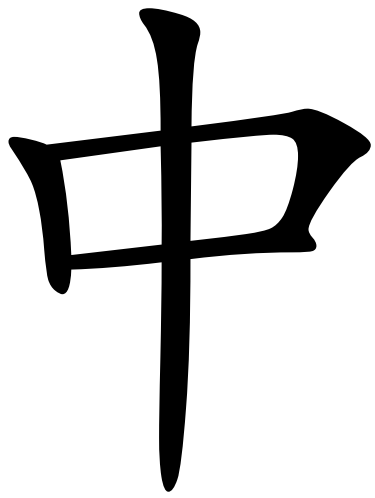
Alternatives

For Japanese scholarly/historical work: Mojikyo, www.mojikyo.org; also see Tron, GTCode.

Also see Wittern, *Embedding Glyph Identifiers in XML Documents*.

Mojikyo has over 140,000 characters last time I checked.

Byte⇔Character Mapping



U+4E2D (Han character)
How do I encode 0x4E2D in bytes for computer processing?

Issues: byte order, non-BMP, C str* functions.

Storing Unicode in Bytes

Official encodings: UTF-8, UTF-16, UTF-32
Practical encodings: ASCII, EBCDIC, Shift-JIS, Big5, GB18030, EUC-JP, EUC-KR, ISCII, KOI8, Microsoft code pages, ISO-8859-*, and others.

Most of the text in the world is in ISO-8859*, Microsoft code pages, JIS, Big5, or one of the GB* standards. But they are all Unicode characters, so standardizing on Unicode doesn't mean you can't use all this stuff, it just means you have to filter it on the way in.

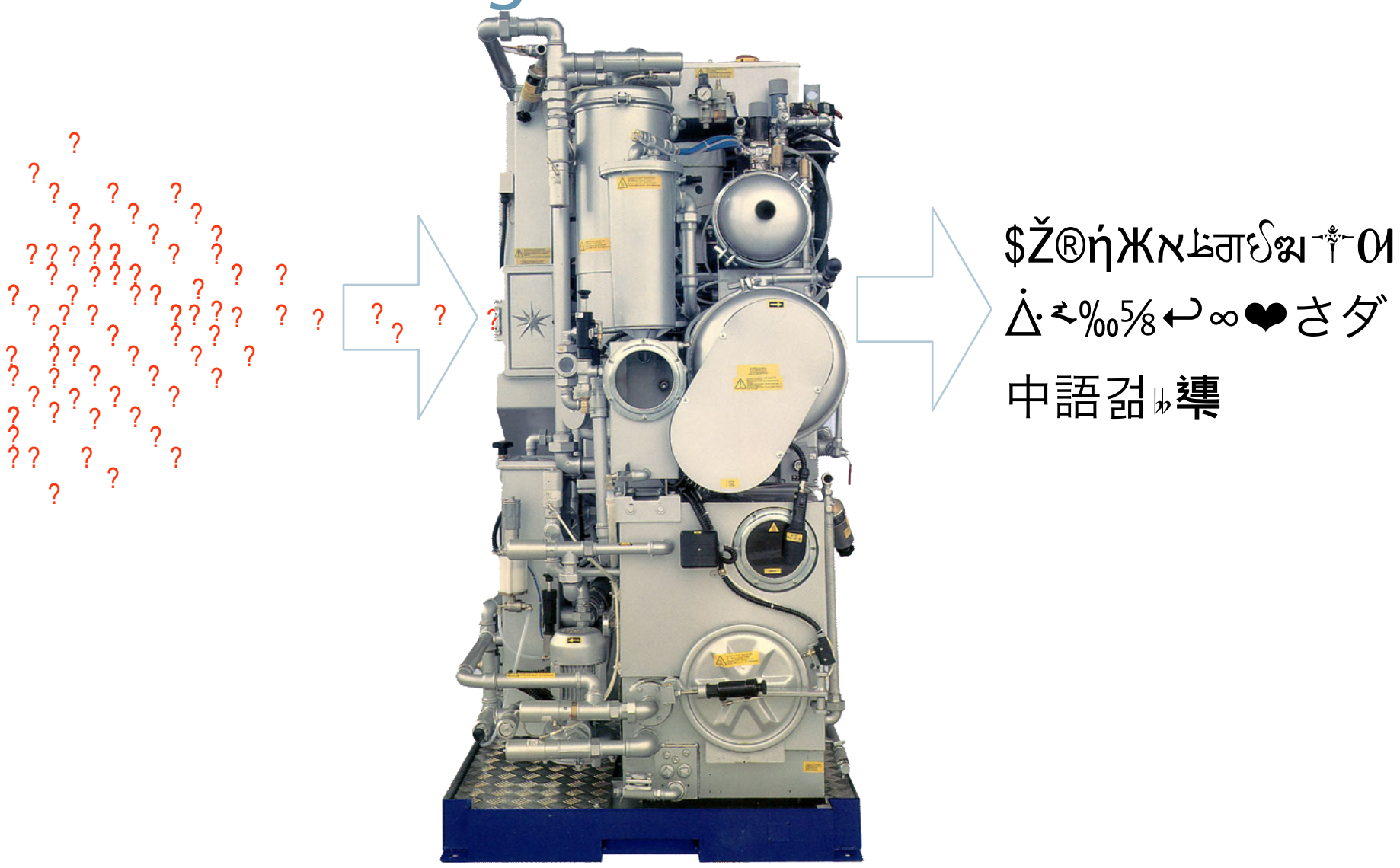
UTF-* Trade-offs

UTF-8: Most compact for Western languages, C-friendly, non-BMP processing is transparent.
UTF-16: Most compact for Eastern languages, Java/C#-friendly, C-unfriendly, non-BMP processing is horrible.
UTF-32: *wchar_t*, semi-C-friendly, 4 bytes/char.
Note: Video is 100MB/minute...

Web search: “characters vs. bytes”

One of the reasons I’m not going to dive deep on all that crap is that ordinary civilian programmers who just want to get a field out of a database and render it on the screen, and want to do this in multiple languages, do not care about encodings! And shouldn’t have to either. This is exactly the kind of thing that you’d a programming langauge to take care of for you

Text Arriving Over the Network



4 approaches: (1) Guess (like web browsers, cf python library). (2) Believe MIME metadata. (3) Trust. (4)...

"An XML document knows what encoding it's in."

- Larry Wall

Notes

What Java Does

Strings are Unicode. A Java “char” is actually a UTF-16 code point, so non-BMP handling is shaky. Strings and byte buffers are separate; there are no unsigned bytes. The implementation is generally solid and fast. The APIs are a bit clumsy and there’s no special regexp syntax.

It’s amazing how much special regexp syntax buys you.

What Perl Does

Perl 5 has Unicode support, in theory. In a typical real-world application, with a Web interface and files and a database, it is very difficult to round-trip Unicode without damage. However, regexp support is excellent. Perl 6 is supposed to fix all the problems...

Notes

What Python 3000 Will Do

String Types Reform (Guido's Slide)

- bytes and str instead of str and unicode
  - bytes is a mutable array of int (in range(256))
  - encode/decode API? bytes(s, "Latin-1")?
  - bytes have some str-ish methods (e.g. b1.find(b2))
  - but not others (e.g. not b.upper())
- All data is either binary or text
  - all text data is represented as Unicode
  - conversions happen at I/O time
- Different APIs for binary and text streams
  - how to establish file encoding? (Platform decides)

I think that Ruby should pay close attention to the policies around the Python 3000 reform project.

Core Methods With I18n Issues

```
== =~ [] []= eql? gsub gsub! index length
lstrip lstrip! match rindex rstrip
rstrip! scan size slice slice! strip
strip! sub sub! tr tr!
```

==/eql interesting because of combining-forms. [] is and length and so on are interesting because you have to know whether you're counting characters or bytes. Strip-family methods are interesting because there are more white-space characters than space, tab, nl, cr.

Problem: Missing Character Iterator

Maybe `String#each_char`

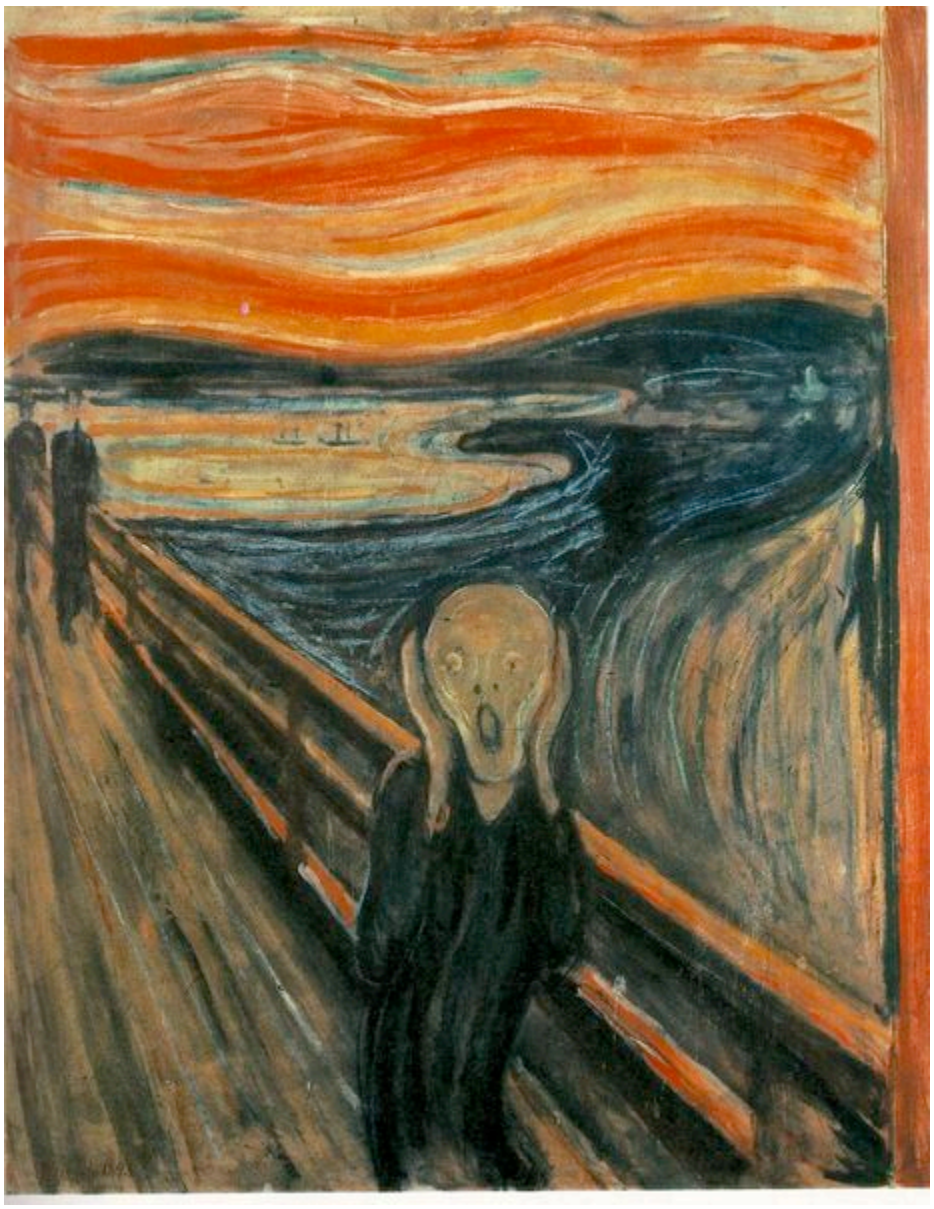
Maybe change `String#each`

Maybe `String#chars`

Notes

On Case-folding

Lower-case ‘I’: ‘i’ or ‘ı’?
Upper-case ‘i’: ‘I’ or ‘İ’?
Upper-case ‘ß’?
Upper-case ‘é’?
Just Say No!



A careful implementation of casefolding, e.g. Java’s, will be insanely slow and still often produce incorrect results.

Regexp and Unicode

```
stag = "<[^/]([^>]*[^/>])?>"
etag = "</[^>]*>"
empty = "<[^>]*/>"

alnum = '\p{L}|\p{N}|' +
   '[\x{4e00}-\x{9fa5}]|' +
   '\x{3007}|[\x{3021}-\x{3029}]'
wordChars =
   '\p{L}|\p{N}|' + "[-._:']|" +
   '\x{2019}|[\x{4e00}-\x{9fa5}]|\x{3007}|' +
   '[\x{3021}-\x{3029}]'

word = "((#{alnum})((#{wordChars})*(#{alnum}))?)"
text = "(#{stag})|(#{etag})|(#{empty})|#{word}"
regex = /#{text}/
```

Will Ruby 2.0 do these?

e.g. “won’t-go”

Notes

Referring to Characters

```
if in_euro_area?
  append 0x20ac # Euro
elsif in_japan?
  append 0xa5   # Yen
else
  append '$'
end
```

Common idiom while writing XML.

Notice that there's no concern for the encoding of the string; the semantics are those of characters.

What Should Ruby Do?

In 2007, programmers around the world expect that, in modern languages, strings are Unicode and string APIs provide Unicode semantics correctly & efficiently, by default. Otherwise, they perceive this as an offense against their language and their culture. Humanities-computing academics often need to work outside Unicode. Few others do.

Notes

Who's Working on the Problem?

Matz/Ko1: M17n for Ruby 2
Julik: ActiveSupport::MultiByte (in edge Rails)
Nikolai: Character encodings project (rubyforge.org/projects/char-encodings/)
JRuby guys: Ruby on a Unicode platform

Notes

Thank You!

Tim.Bray@sun.com
www.tbray.org/ongoing/
this talk: www.tbray.org/talks/rubykaigi2007.pdf