COSC345 Week 24

Internationalisation and Localisation

29 September 2015

Richard A. O'Keefe

From a Swedish hôtel room

Hjälp oss att värner om vår miljö!
För att minska utsläpp av tvättmedel,
byter vi Er handduk bara när Ni vill:

1. Handduk på golvet
— betyder att Ni vill ha byte

2. ...

The translation

Help us to care for our environment!
To reduce the use of laundry detergents,
we shall change your towel as follows:

1. Towel on the floor
— you want to have a new towel.

2. *Towel hung up*
— *you want to use it again.*

People should be able to use computers in their own language.

— It's just *right* not to make people struggle with unfamiliar linguistic and cultural codes.

— Sensible people won't pay for programs that are hard to use.

— Internationalisation (I18N) means making a program so that it does not enforce a particular language or set of cultural conventions

— Localisation (L10N) means adapting an internationalised program to a particular language *etc*.

— UNIX, VMS, Windows, all support internationalisation and localisation; the Macintosh operating system has done this better for longer.

Characters

You know that there are 26 letters in 2 cases.

But Swedish has å, ä, ö, Å, Ä, and Ö (29 letters), Croatian has d j, D j, D J, and others (3 cases), German has ß, which has no single upper case version (might be SS, might be SZ, both of which are two letters), Latin-1 has 58 letters in 2 cases (including 2 lower case letters with no upper case version), Arabic letters have 4 contextual shapes (beginning, middle, or end of word, or isolated), which are not *case* variants (Greek has one such letter, and Hebrew has several; even English used to), and Chinese has tens of thousands of characters.

Characters continued

You know that blanks separate words, but they don't in Chinese, and Unicode (ISO 10646) contains several zero-width characters, some of which are separators (zero-width space, for example) and some of which are not (zero-width joiner, for example).

You know that there are 10 decimal digits 0–9. But Unicode 4.0 has no fewer than 37 versions of "DIGIT THREE": plain, subscript, superscript, Arabic-Indic, Eastern Arabic-Indic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Tibetan, Myanmar, Ethiopic, Khmer, Mongolian, Limbu, Osmanya, + decorated versions.

How do you know which digits to use in output?

A little history of character sets

Baudot code (5 bits, 3 or 4 shifts), still used in radio

Fieldata and BCD (6 bits, no lower case letters)

ASCII (7 bits, the-computer-is-a-typewriter model)

ISO 8859 family (8 bits, lots of ASCII extensions)

Unicode (21 bits, 120,672 chars)

ISO 10646 (31 bits, Unicode was Basic Multilingual Plane of this, planes 0, 1, 2, and 14 currently have characters).

C and C++ have `wchar_t` for wide characters, Java has `char` (16 bits only, not enough any more!), and Ada 95 has `Wide_Character`.

Added in Unicode 6.0

0840..085F	Mandaic
1BC0..1BFF	Batak
AB00..AB2F	Ethiopic Extended-A
11000..1107F	Brahmi
16800..16A3F	Bamum Supplement
1B000..1B0FF	Kana Supplement
1F0A0..1F0FF	Playing Cards
1F300..1F5FF	Miscellaneous Symbols And Pictographs
1F600..1F64F	Emoticons
1F680..1F6FF	Transport And Map Symbols
1F700..1F77F	Alchemical Symbols
2B740..2B81F	CJK Unified Ideographs Extension D

You know there is a one-to-one correspondence between "characters" and codes.

But glyph, grapheme, coded character, code, and byte are five different concepts with no one-to-one correspondence. In English, "é" is two graphemes (a letter e + a stress mark); in French it's one. A single character may be one or two codes in Unicode; a single code may be stored as 1–4 bytes in UTF-8 (so character ≠ code ≠ byte). The letter ÿ may be stored as U+00FF or U+0079,U+0308.

But Indic and Semitic scripts have a consonantal skeleton with vowels "around" the consonants; one "glyph" may be 2 "letters".

How long is a string?

“Ljubljana”	7 codepoints	7 or 9 letters?
“Æneas”	5 codepoints	5 or 6 letters?
“$\vec{a}$”	2 codepoints	1 letter
“$\ddot{e}$”	1 codepoint	1 letter
“$\ddot{x}$”	2 cdepoints	1 letter
“ā”	1 codepoint	1 letter
“w̄”	2 codepoints	1 letter
“$\acute{\omega}_{\iota}$”	1 codepoint	2 letters?

Character solution

Use library code for

* classifying codes

* stepping through strings by characters, words, lines

* normalisation (so that “é” and “e”+“´” are equal)

* comparison (the draft ISO standard was 150 pages)

whenever possible. There is useful stuff in C and C++ (`ctype.h`, `wctype.h`) and really good coverage in Java.

Dates and Times

What date does 1/2/3 represent? USA: 2 Jan 2003; here: 1 Feb 2003; some places: 3 Feb 2001. Year number could be Gregorian, Julian, Gregorian mod 100, Gregorian - 1900, regnal year of Japanese emperor, year since the founding of Rome (A.U.C.), and so on.

Use library code to read and write dates (strftime() in C writes: %x is date, %X is time, %c is date and time, all according to locale's convention; strptime() reads).

Tell library what locale to use. (Look up `LC_TIME`.) There are many calendars in use other than the Gregorian one; it's not just names of months and days that differ.

Dates and Times II

Are times written as 14:30 (Europe), 1430 (US Military), 2:30 p.m. (English), or 2.30 p.m. (strictly 2.30 should be 2:18pm; this is the "I don't care if it's stupid, it saves ink" notation). Use locale-sensitive library code to give people what they are used to.

For machine/machine communication, use ISO 8601:
yyyy[.]*mm*[.]*dd*[T]*hh*[:]*ii*[:]*ss*[.*sss*][±*timezone*].
New Zealand *timezone* is +1200 (winter) or +1300 (summer).
`ISO8601.html` summarises; `ISO8601-1988.pdf` and `ISO8601-2000.pdf` are obsolete editions.

In SQL, use DATE, TIME, and TIMESTAMP [WITH TIMEZONE] types whenever you can.

A Warning about Dates

Arithmetic on calendar dates is amazingly tricky, especially when more than one calendar is involved.

To compute with dates, keep them as Julian Day Numbers or Modified Julian Day Numbers. Do arithmetic on these numbers. Convert back to y/m/d when you want to produce output.
`dates.c` has code.

The book "Calendrical Calculations" by Nachum Dershowitz and Edward Reingold gives calculations for many calendars; it's in the Central library.

Getting it wrong: an example

Timestamps in a certain programming language are represented as an absolute time in UTC (year, month, day, hour, minute, second) combined with a time zone offset.

Error: the range of offsets is -12 hours to +12 hours, but the limit in the real world is +14 hours (the Line Islands). The language's limit does not even include the Chatham Islands.

Problem: if you do arithmetic on a time stamp, the zone offset does not change, meaning that arithmetic that crosses Daylight Savings Time changes arrives in the wrong zone.

The problem can be traced back to ISO 8601, which lets you write a local time but does not let you name the rules used.

Numbers and sums of money

The decimal point might be '.' or ',' or '·' (the last is best). The thousands separator might be ',' or '.' (hence the DECIMAL POINT IS COMMA option in COBOL) or ' ' (the last is unambiguous). Is the millions separator the same as the thousands separator? Are digits grouped in 3s, 4s, or 5s, and are all groups the same width? Are negative numbers written as -*nn*, (*nn*), or <RED>*nn*</RED>?

Is money written with a symbol $ (pound, dollar, yen, general currency sign, euro, florin) or in letters (GPB, NZD, EUR, SEK, Kr) and does it precede or follow the number? Is the negative sign the same as for numbers or different? How are fractions shown? Is a cent sign used?

Numbers and Money II

The C standard includes a function `localeconv`() which returns a pointer to a record with fields including mon_decimal_point, frac_digits, mon_thousands_sep, int_frac_digits, int_curr_symbol, & currency_symbol (money features), decimal_point, thousands_sep, positive_sign, negative_sign, p_cs_precedes, n_cs_precedes, p_sep_by_space, n_sep_by_space, p_sign_posn, and n_sign_posn (number features).

The C99 standard does not include any functions for formatting numbers using this information; you'll have to write your own. strfmon() is popular, but not C99 or POSIX.

Which digits are used? English, real Arabic, Indic (several sets), or what? Are they full width or half width? Should symbols for fractions be used, or decimal fractions? `localeconv`() doesn't tell you.

Words and Messages

There are vocabulary differences between dialects: stove/cooker, crib/bach/weekender, hut/cubby, togs/swimming trunks/swim shorts, tin/can, frying pan/frypan.

Main issue: different languages. Natural Language Generation from symbolic structures *can* be done (see the ILEX project or Aarne Ranta's GF for examples) but is still difficult to set up; simplest way is "Message Catalogue".

A message catalogue is basically a table mapping a message identifier to a string. These strings could be file names or even mini programs, not just text.

Message Catalogues I

See catclose (3), catgets (3), catopen (3), gencat (1), gettxt (3) in your UNIX manual.

Beware: different languages may use different scripts, different phrase orders ("insert *X* into *Y*" may become "into *Y* insert *X*"), and different lengths. A box big enough for the English message may be too small for other languages. Strings in a catalogue could have size codes if you choose.

Message Catalogues II

The Macintosh has had *resource forks* in the file system since day 1. A file has data, plus a resource map (like a message catalogue). An application keeps a stack of resource forks, typically document (on top), application, and system (at bottom). Resources are identified by type, id code, and possibly an index. Resources can be strings, icons, keyboard layout, window layout, comparison methods, date/time formats, *etc*.

Applications were localised using the ResEdit program, but one copy of an application can only handle one locale. One machine = one user = one locale = one resource fork.

Message Catalogues III

In the NTFS Windows file system, a file can have a list of attributes with associated values attached to it. Programs can have resource data (from `.rc` files) compiled in; resources are also identified by language. Basically the MacOS model.

There is a very good book on internationalisation for Windows.

Message catalogues IV

UNIX "resources" are held *outside* the program in the file system as message catalogues. Several programs can share a catalogue. One program can be used by many people at once, each using messages from a different catalogue (`LC_LANG` is part of the environment, not the program). Use the `LC_*` environment variables to customise your locale.

C-defined locale data (character set, character classification, comparison, number, money, and date-time formatting) uses a different mechanism that applies to *all* programs (most of this stuff was in the operating system's own resource fork in old MacOS).

http://java.sun.com/docs/books/tutorial/i18n/ is a tutorial for Java internationalisation; Java "Resource Bundles" are message catalogues.

General theme

* Know what can vary

* Work through (standard) libraries

* Avoid literal strings

* Text can change size and order

* Native speaker for final polishing, please

* Know your libraries! There is special support for this in C and C++, and a lot of support in Java.

X/Open

All in ∼ok/ANSI-standards/

X:Open-IGV2.pdf Internationalization Guide (all)

X:Open-C616.pdf Portable Layout Services

→ Chapter 2 about un-Western scripts

X:Open-E401.pdf (Unicode) Coëexistence & Migration

X:Open-E408.pdf Internationalization of X/Open Specifications

→ Chapters 2 and 4 overview and interworking

Resources

Standards and exam review.

File names mentioned in this lecture can be found in the top level of the 345 web site.

A major resource is the Unicode "Common Locale Data Repository", http://www.unicode.org/cldr/.

"Unicode Demystified" book.