inField

Data Management and Archiving

University of California at Santa Barbara, June 24-27, 2008

Unicode and Standardized Notation

Anthony Aristar

Once upon a time…

- There were people who decided to invent computers.
- And they thought: "What are all the characters anyone could possibly need? We'll code these."
- It was even perfect, in that you could use 7 bits to code these characters. The code set, which was called ASCII, was able to represent every character using a number between 32 and 127. Space was 32, the letter "A" was 65, and so on. Codes below 32 were used for control characters, and if you turned the eighth bit on, you could use codes 128-255 for whatever you liked.
- And everyone lived happily ever after.

Pesky Europeans

- Into this perfect world intruded all these odd people who don’t speak English.
- Most of them don’t matter of course, but there are a few, like Germans… French… Spanish… Italians… Russians…Swedes and so on, who have lots of money or power and complained.

The fix was in

- After grumbling about how picky these Europeans were, the computer people had a bright idea.
- What about the characters above 127? Why not just use some of these for the few characters most of these guys use? And in languages like Russian, you could use the ASCII codes that would usually give you “B” or “P”, but instead you’d see Б and П.

- Eventually this free-for-all got codified in the ANSI standard. In the ANSI standard, everybody agreed on what to do below 128, which was pretty much the same as ASCII, but there were lots of different ways to handle the characters from 128 and on up, depending on where you lived. These different systems were called *code pages*.

- The idea was that you could take a code like, say, 123, but it could appear as any one of these, on different code pages: ä æ é à ° ¨
- In short, you use the same code-point, but you'd ***render*** – display it -- it in a different way when you chose different code-pages.
- Everyone was now happy again…

- But how do you access these different code pages? Even with this ANSI standard, you couldn’t get more than a single character set on one computer, for the operating system had to choose what code page rendering to use.
- The solution: give the operating system access to all the code pages, and **let the font being used decide which rendering should be used** in each case.

- Thus, when you wanted to change to a different character set, you simply switched the font.
- The font would encode each character with the appropriate code-point from the appropriate code page, and display it with the appropriate character shape.
- Problem: are fonts, character-sets and code-pages the same?

So…

- Well, no.
- You can see the results of this confusion by considering the following. Here's a short sentence in Russian:

Если Украина вступит в НАТО, между ней …

- But if a colleague in Russia sent this to you, you would see this text:

  Esli Ukraina vstupit v NATO, me`du nei

  Exactly the same code-points from the right code-page are there…But since **he** has a Cyrillic font on this machine, and **you** don’t, the characters just aren’t being rendered as they should be.

Meanwhile, in the east…

- That’s one problem. But there’s another.
- 256 possible characters sounds like a lot.
- But what about Chinese? Japanese? There’s no way you can fit thousands of characters into 8 bits.
- “Solution”: DBCS, or "double byte character set" in which *some* letters are stored in one byte and others take two.

- Now you have another problem. A typical mail header in DBCS:
- Subject: ¹ÚÆ®·¹ÀÌ³Ê°¡ µå¸®´Â Çï½º¸Å°ÅÁø NO.4
- If you don’t happen to have a DCBS interpreter on your machine, anyway.

And then… Unicode!

- What about this as an idea?
  - Every character in every language has a code-point.
  - No code-point is ever reused.
  - Every code-point has a unique rendering or set of renderings, all of which are seen as equivalent.
  - That way, so long as you have a font coded for this system, no matter what someone else may be using, the character will always show up correctly…
  - In short, you can use any font for this system, just so long as it's built for it.

But how to encode this?

- Obviously, one 8-bit byte won't do it for such a system. We need tens of thousands of code-points
- So use more than one byte. But how many?
- Well, you could use two 8-bit bytes. That will give you 65,536 possible unique characters. That will deal with most – though not quite all – of the characters used in human language.

- So is Unicode a character encoding system that uses two bytes per character?
- No. And to prove it, take a text, write it in ASCII, drop it into a word-processor, and turn it into Unicode.
- Is the text still interpretable? Well, sometimes it is, and sometimes it isn’t. It seems to depend upon what kind of Unicode you turn it into.

Contradiction?

- If each Unicode character is unique, what is going on? How come sometimes ASCII seems to come through fine, while at other times it doesn't? This seems to imply that a character can have more than one Unicode encoding.
- Is there a contradiction here?

Code points revisited

- No, it's not a contradiction, for Unicode does something very different from previous character systems.
- It distinguishes ***code points*** from ***encodings***.
- All Unicode characters have a unique code point.
- But these can be encoded in different ways.

- For example, the character “e” has the **code-point** U+0065. That’s it. “e” can never have any other code-point.
- This can be **encoded** in two bytes as 00 65
- This is called UTF-16, since it uses 16 bits to store an actual character.
- But it can also be encoded as UTF-8

inField

The Solution: UTF-8

- If UTF-16 uses 16 bits to store a character, does this mean that UTF-8 uses 8 bits?
- Well, partly. The way it works is:
  - Every code-point from 0-127 is stored in one byte (same as ASCII)
  - Every code-point above 127 is stored in either two, three, or four bytes.
- The result: every ASCII file looks exactly the same in UTF-8 as it does in a system using 256 code-points.
- Because this ensures that old files will be readable, almost all Unicode web sites use UTF-8.

Other Unicode Encodings

- UTF-7, which is very like UTF-8 except that the high (eighth) bit is always zero (rarely used, and then usually in archaic mail programs).
- UCS-4 stores each code point in 4 byte chunks, but uses a mammoth amount of memory, so almost no one uses it.
- UTF-32, which uses 4 bytes for every character. Very rare, but necessary if we want to be able to encode every character ever used.

Isn't all this variety wonderful?

- Well, not quite.
- It means that even if you're using Unicode, you have to say which Unicode encoding you're using.
- That's why every web page should have something like this in its header:
- **<meta http-equiv="Content-Type" content="text/html; charset=utf-8">**
- And this has to be ***the very first thing*** in the header because ***this is what tells the web-server how to read the rest of the file***.
- If you don't do this, the web-server is going to guess, and it often gets it wrong.

Rendering again

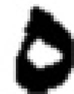
- We mentioned rendering before.
- In old style non-unicode fonts, the distinction between character and rendering meant nothing.
- Rendering was used to get both different **forms** of a character and different **characters**.
- The issue of glyph was completely ignored.
- In Unicode, character, glyph and rendering must all be distinguished.

The difference between Character and Glyph

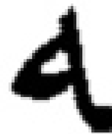
g g g g

The difference between glyph and character on the one hand and rendering on the other

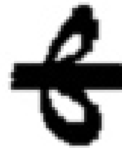
- One Character
- Many glyphs
- Rendered Appropriately



Independent



Joined to right



Joined on both sides

Joined on left

Ordering

Avram said “יש לי 23 ילדים.” and smiled.

Just one problem to be wary of: Combining characters

In Unicode accent marks can be represented with their own code point values…

é = U+0065 (e) U+0301 (accent)

…but common combinations of letters and accents are also given their own code points for convenience.

é = U+00E9

In summary: Character semantics

- The Unicode standard includes an extensive database that specifies a large number of *character properties,* including:
  - Name
  - Type (e.g., letter, digit, punctuation mark)
  - Decomposition
  - Case and case mappings (for cased letters)
  - Numeric value (for digits and numerals)
  - Combining class (for combining characters)
  - Directionality
  - Line-breaking behavior
  - Cursive joining behavior
  - For Chinese characters, mappings to various other standards and many other properties

The General Scripts Area

Range	Scripts			
00/01	Latin			
02/03	IPA	Diacritics	Greek	
04/05	Cyrillic	Armenian	Hebrew	
06/07	Arabic	Syriac	Thaana	
08/09	Devanagari	Bengali		
0A/0B	Gurmukhi	Gujarati	Oriya	Tamil
0C/0D	Telugu	Kannada	Malayalam	Sinhala
0E/0F	Thai	Lao	Tibetan	
10/11	Myanmar	Georgian	Hangul	
12/13	Ethiopic	Cherokee		
14/15	Canadian Aboriginal Syllabics			
16/17	Ogham	Runic	Philippine	Khmer
18/19	Mongolian			
1A/1B				
1C/1D				
1E/1F	Latin	Greek		

Unicode Coverage

- European scripts
  - Latin, Greek, Cyrillic, Armenian, Georgian, IPA
- Bidirectional (Middle Eastern) scripts
  - Hebrew, Arabic, Syriac, Thaana
- Indic (Indian and Southeast Asian) scripts
  - Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Sinhala, Thai, Lao, Khmer, Myanmar, Tibetan, Philippine
- East Asian scripts
  - Chinese (Han) characters, Japanese (Hiragana and Katakana), Korean (Hangul), Yi
- Other modern scripts
  - Mongolian, Ethiopic, Cherokee, Canadian Aboriginal
- Historical scripts
  - Runic, Ogham, Old Italic, Gothic, Deseret
- Punctuation and symbols
  - Numerals, math symbols, scientific symbols, arrows, blocks, geometric shapes, Braille, musical notation, etc.

- This seems like a very simple issue. How do we mark up linguistic material?
- Does it matter after all what we call grammatical categories in something like the following so long as we are clear?

Afisi	a -na -ph -a	nsomba
hyenas	SP-PST-kill-ASP	fish

'The hyenas killed the fish.'

- The problem is that though something like that might be clear to a human being, it's not at all clear to a computer.
- So how do we use computers to compare data across mark-up schemes, when everyone uses the annotation they like?

The Solution: Standardized Markup

- The solution is to standardize markup in some way. So what do we do?
- The answer: use an “interlanguage” for translation among markups

A number of Interlanguages…

- Leipzig Glossing rules (http://www.eva.mpg.de/lingua/resources/glossing-rules.php)
- Developed by Bernard Comrie, Martin Haspelmath, and Balthasar Bickel at Leipzig.
- Consist of:
  - Ten rules for the "syntax" and "semantics" of interlinear glosses and
  - an appendix with a proposed "lexicon" of abbreviated category labels.
- If used consistently, these can be used to produce interoperability between data sets.

But…

- Problem: the rules require that everyone do exactly the same thing, and use exactly the same terms.

- What is needed is a system that allows us to compare linguistic data sets which use different markup definitions… But also:
  - Methods are needed by which the semantics as well as the syntax of each tagging scheme can be compared, e.g..
    - Both A and B use “absolutive”. Do they mean the same thing?
    - A uses “possessive” where B uses “genitive”. How do they compare?

What we need…

- A system that allows us to compare linguistic data sets which use different markup definitions… BUT:
  - Methods are needed by which the semantics as well as the syntax of each tagging scheme can be compared, e.g..
    - Both A and B use “absolutive”. Do they mean the same thing?
    - A uses “possessive” where B uses “genitive”. How do they compare?

- Two means exist for doing this:
  - DatCats
  - Ontologies

DAT-CATs

- DatCats: **ISO-TC37/SC4 Data Category Registry** (http://www.localisation.ie/resources/presentations/2003/Conference/Presentations/Laurent%20Romary.ppt)
- Examples of values: ablativeCase or dativeCase.
- Since you can link your own terms to these DatCats, you can use them while still using your own terms.
- DatCats are relatively unstructured. They are essentially a set of terms to which things can be linked.
- (But more structure seems to be on the way…)

inField

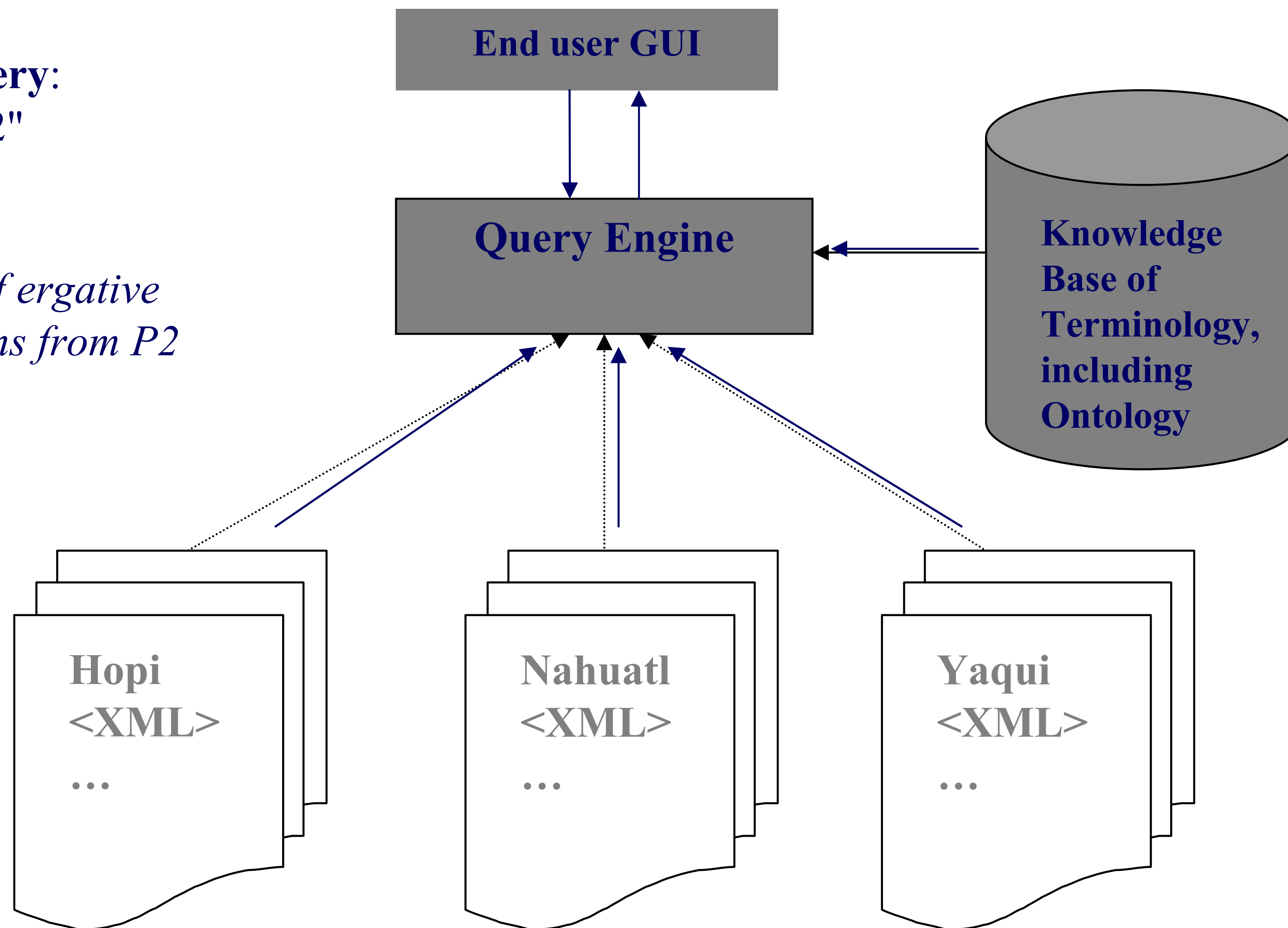
Or you can use an Ontology

- An “ontology” is most similar to the DatCat system, but has more structure, in the form of hierarchy.
- It defines, in computer terms, what terms are the same, how they are equivalent, and how terms can be compared.
- It defines what terms subsume others.
- Just as with DatCats you can allow everyone to use their own terminology — no-one has to change — but still allow machines to compare linguistic material.
- Example: the GOLD (the General Ontology of Linguistic Description): http://www.linguistics-ontology.org/

In short, you can have a system like this…

Sample query:
"ergative P2"

Returns:
Examples of ergative constructions from P2 languages



GOLD: A linguistic Ontology

- Contents
  - Knowledge base of Linguistic Terminology
  - Current version concentrates specifically on Morphosyntactic Terminology
- Structure
  - Inheritance, *is-a*
  - Multiple Inheritance
  - Mereological (Part-Whole) Relations
  - And other relations