Chapter 4 Character encoding in corpus construction

Anthony McEnery *Zhonghua Xiao*

Lancaster University

Corpus linguistics has developed, over the past three decades, into a rich paradigm that addresses a great variety of linguistic issues ranging from monolingual research of one language to contrastive and translation studies involving many different languages. Today, while the construction and exploitation of English language corpora still dominate the field of corpus linguistics, corpora of other languages, either monolingual or multilingual, have also become available. These corpora have added notably to the diversity of corpus-based language studies.

Character encoding is rarely an issue for alphabetical languages, like English, which typically still use ASCII characters. For many other languages that use different writing systems (e.g. Chinese), encoding is an important issue if one wants to display the corpus properly or facilitate data interchange, especially when working with multilingual corpora that contain a wide range of writing systems. Language specific encoding systems make data interchange problematic, since it is virtually impossible to display a multilingual document containing texts from different languages using such encoding systems. Such documents constitute a new Tower of Babel which disrupts communication.

In addition to the problem with displaying corpus text or search results in general, an issue which is particular relevant to corpus building is that the character encoding in a corpus must be consistent if the corpus is to be searched reliably. This is because if the data in a corpus is encoded using different character sets, even though the internal difference is indiscernible to human eyes, a computer will make a distinction, thus leading to unreliable results. In many cases, however, multiple and often competing encoding systems complicate corpus building, providing a real problem. For example, the main difficulty in building a multilingual corpus such as EMILLE is the need to standardize the language data into a single character set (see Baker, Hardie & McEnery et al 2004).[1] The encoding, together with other ancillary data such as markup and annotation schemes, should also be documented clearly. Such documentation must be made available to the users.

A legacy encoding is typically designed to support one writing system, or a group of writing systems that use the same script (see discussion below). In contrast, Unicode is truly multilingual in that it can display characters from a very large number of writing systems. Unicode enables one to surmount this Tower of Babel by overcoming the inherent deficiencies of various legacy encodings.[2] It has also facilitated the task of corpus building (most notably for multilingual corpora and corpora involving non-Western languages). Hence, a general trend in corpus building is to encode corpora (especially multilingual corpora) using Unicode (e.g. EMILLE). Corpora encoded in Unicode can also take advantage of the latest Unicode-compliant corpus tools like *Xaira* (Burnard & Todd 2003) and *WordSmith* version 4.0 (Scott 2003). In this chapter, we will consider character encoding from the viewpoint of corpus linguistics rather than programming, which means that the account presented

[1] See the corpus website http://www.ling.lancs.ac.uk/corplang/emille for more details of the EMILLE corpus.

[2] 'Legacy encoding' is used here interchangeably with language specific, or native character code.

here is less technical and that some of the proposals we make may differ slightly from those that would be ideal for programmers.

This chapter first briefly reviews the history of character encoding. Following from this is a discussion of standard and non-standard native encoding systems, and an evaluation of the efforts to unify these character codes. Then we move on to discuss Unicode as well as various Unicode Transformation Formats (UTFs). As a conclusion, we recommend that Unicode (UTF-8, to be precise) be used in corpus construction.

Shift in: what is character encoding about?

The need for electronic character encoding first arose when people tried to send messages via telegraph lines using, for example, the Morse code.[3] The Morse code encodes alphabets and other characters, like major punctuation marks, as dots and dashes, which respectively represent short and long electrical signals. While telegraphs already existed when the Morse code was invented, the earlier telegraph relied on varying voltages sent via a telegraph line to represent various characters. The earlier approach was basically different from the Morse code in that with this former approach the line is always "on" whereas with the latter, the line is sometimes "on" and sometimes "off". The binary "on" and "off" signals are what, at the lowest level, modern computers use (i.e. 0 and 1) to encode characters. As such, the Morse code is considered here as the beginning of character encoding. Note, however, that character encoding in the Morse code is also different from how modern computers encode data. Whilst modern computers use a succession of "on" and "off" signals to present a character, the Morse code uses a succession of "on" impulses (e.g. the sequences of –·, –··· and –·–· stand respectively for capital letters A, B and C), which are separated from other sequences by "off" impulses.

A later advance in character encoding is the Baudot code, invented by Frenchman Jean-Maurice-Émile Baudot (1845-1903) for teleprinters in 1874. The Baudot code is a 5-bit character code that uses a succession of "on" and "off" codes as modern computers do (e.g. 00011 without shifting represents capital letter A). As the code can only encode 32 (i.e. 2^5) characters at one level (or "plane"), Baudot employs a "lock shift scheme" (similar to the SHIFT and CAPS LOCK keys on your computer keyboard) to double the encoding capacity by shifting between two 32-character planes. This lock shift scheme not only enables the Baudot code to handle the upper and lower cases of letters in the Latin alphabet, Arabic numerals and punctuation marks, it also makes it possible to handle control characters, which are important because they provide special characters required in data transmission (e.g. signals for "start of text, "end of text" and "acknowledge") and make it possible for the text to be displayed or printed properly (e.g. special characters for "carriage return" and "line feed"). Baudot made such a great contribution to modern communication technology that the term *Baud rate* (i.e. the number of data signalling events occurring in a second) is quite familiar to many of us.

One drawback of 5-bit Teletype codes such as the Baudot code is that they do not allow random access to a character in a character string because random access requires each unit of data to be complete in itself, which prevents the use of code extension by means of locking shifts. However, random is essential for modern computing technology. In order to achieve this aim, an extra bit is needed. This led to 6-bit character encoding, which was used for a long time. One example of such codes is the Hollerith code, which was invented by American Herman Hollerith (1860-1929)

[3] The Morse code was invented by American Samuel Finley Breese Morse (1791-1872).

for use with a punch card on a tabulating machine in the U.S. Census Bureau. The Hollerith code could only handle 69 characters, including upper and lower cases of Latin letters, Arabic numerals, punctuation marks and symbols. This is slightly more than what the Baudot code could handle. The Hollerith code was widely used up to the 1960s.

However, the limited encoding capacity of 6-bit character codes was already felt in the 1950s. This led to an effort on the part of telecommunication and computing industries to create a new 7-bit character code. The result of this effort is what we know today as the ASCII (the American Standard Code for Information Interchange) code. The first version of ASCII (known as ASCII-1963), when it was announced in 1963, did not include lower case letters, though there were many unallocated positions. This problem, among others, was resolved in the second version, which was announced in 1967. ASCII-1967, the version many people still know and use today, defines 96 printing characters and 32 control characters. Although ASCII was designed to avoid shifting as used in Baudot code, it does include control characters such as shift in (SI) and shift out (SO). These control characters were used later to extend the 7-bit ASCII code into the 8-bit code that includes 190 printing characters (cf. Searle 1999).

The ASCII code was adopted by nearly all computer manufacturers and later turned into an international standard (ISO 646) by the International Standard Organization (ISO) in 1972. One exception was IBM, the dominant force in the computing market in the 1960s and 1970s.[4] Either for the sake of backward compatibility or as a marketing strategy, we do not know which for sure, IBM created a 6-bit character code called BCDIC (Binary Coded Decimal Interchange Code) and later extended this code to the 8-bit EBCDIC (Extended Binary Coded Decimal Interchange Code). As EBCDIC is presently only used for data exchange between IBM machines, we will not discuss this scheme further.

The 7-bit ASCII, which can handle 128 (i.e. 2^7) characters, is sufficient for the encoding of English characters. With the increasing need to exchange data internationally, which usually involves different languages, as well as using accented Latin characters and non-Latin characters, this encoding capacity quickly turned out to be inadequate. As noted above, the extension of the 7-bit ASCII code into the 8-bit code significantly increased its encoding capacity. This increase was important, as it allowed accented characters in European languages to be included in the ASCII code. Following the standardization of the ASCII code and ISO 646, ISO formulated a new standard (ISO 2022) to outline how 7/8-bit character codes should be structured and extended so that native characters could be included. This standard was later applied to derive the whole ISO 8859 family of extensions of the 8-bit ASCII/ISO 646 for European languages. ISO 2022 is also the basis for deriving 16-bit (double-byte) character codes used in East Asian countries such as China, Japan and Korea (the so called CJK language community).

Legacy encoding: complementary/competing character codes

The first member of the ISO 8859 family, ISO 8859-1 (unofficially known as Latin-1), was formulated in 1987 (and later revised in 1998) for Western European languages such as French, German, Spanish, Italian and the Scandinavian languages, among others. Since then, the 8859 family has extended to 15 members. However, as can be

[4] IBM is an acronym for International Business Machines, which was established on the basis of a company formed, in 1896, by Herman Hollerith after his success.

seen in Table 1 (cf. Gillam 2003: 39-40), these character codes mainly aim at writing systems of European languages.

It is also clear from the table that there is considerable overlap between these standards, especially the many versions of the Latin characters. Each standard simply includes a slightly different collection of characters to optimise the performance of a particular language or group of languages. Apart from the 8859 standards, there also exist ISO 2022-compliant character codes (national variants of ISO 646) for non-European languages, including, for example, Thai (TIS 620), Indian languages (ISCII), Vietnamese (VISCII) and Japanese (JIS X 0201). In addition, as noted in the previous section, computer manufacturers such IBM, Microsoft and Apple have also published their own character codes for languages already covered by the 8859 standards. Whilst the members of the 8859 family can be considered as complementary, these manufacturer tailored "code pages" are definitely competing character codes.

Table 1 ISO 8859 standards

ISO-8859-x	**Name**	**Year**	**Languages covered**
1	Latin-1	1987	Western European languages
2	Latin-2	1987	East European languages
3	Latin-3	1988	Southern European languages
4	Latin-4	1988	Northern European languages
5	Latin/Cyrillic	1988	Russian, Bulgarian, Ukrainian, etc.
6	Latin/Arabic	1987	Arabic
7	Latin/Greek	1987	Greek
8	Latin/Hebrew	1988	Hebrew
9	Latin-5	1989	Turkish (Replaces Latin-3)
10	Latin-6		Northern European languages (Unifies Latin-1 and Latin-4)
11	Latin/Thai		Thai
12	Currently unassigned		May be used in future for Indian or Vietnamese
13	Latin-7		Baltic languages (Replaces Latin-4 and supplements Latin-6)
14	Latin-8		Celtic characters
15	Latin-9	1998	Western European languages (Replaces Latin-1 and adds the euro symbol plus a few missing French and Finnish characters)
16	Latin-10		Eastern European languages (Replaces Latin-2 and adds the euro symbol plus a few missing Romanian characters

The counterparts of the 8859 standards for CJK languages are also wrapped around ISO 2022, including, for example, ISO 2022-JP, ISO-2022-CN and ISO-2022-KR. These standards are basically 7-bit encoding schemes used for email message encoding. Whilst the 7 or 8-bit character codes are generally adequate for English and other European languages, CJK languages typically need 16-bit character codes, as all of these languages use Chinese characters, which may well exceed tens of thousands. The number of Chinese characters in 1994 was 85,000. Most of these characters,

however, are only used infrequently. Studies show that 1,000 characters cover 90%, 2,400 characters cover 99%, 3,800 characters cover 99.9%, 5,200 characters cover 99.99%, and 6,600 characters cover 99.999% of written Chinese (cf. Gillam 2003: 359). Nevertheless, even the lower limit for literacy, 2,400 Chinese characters, considerably exceeds the number of characters in European languages. Unsurprisingly, double-byte (16-bit) encoding is mandatory for East Asian languages. The double byte scheme is also combined with 7 or 8 bit encoding so that Western alphabets are covered as well. Encoding schemes of this kind are called multi-byting schemes.

Character encoding of East Asian languages started in Japan when the Japanese Industrial Standard Committee (JISC) published JIS C 6220 in 1976 (which was later renamed in 1987 as JIS X 0201-1976). JIS C 6220 is an 8-bit character code which does not include any Chinese characters (or *kanji* as the Japanese call them). Shortly after that, in 1978, JISC published the first character code that includes *kanji* (divided into different levels), JIS C 6226-1978, which shifts between the national variant of ISO 646 and the 8-bit character set of level 1 *kanji*. JIS C 6226 was redefined in 1981 (then JIS C 6226-1983) and renamed in 1987 as JIS X 0208-1983. When level 2 *kanji* was added to level 1 in 1990, the standard became JIS X 0208-1990, including 6,355 *kanji* of two levels. Another 5,801 *kanji* were added when a supplementary standard, JIS X 0212-1990, was published in the same year. The publication of JIS X 0213 (7-bit and 8-bit double byte coded extended Kanji sets for information interchange) in 2000 added 5,000 more Chinese characters.

Whilst JIS X 0208/0213 shift between the 7-bit Japanese variant of ISO 646 and the 16-bit character set, the Shift-JIS encoding invented by Microsoft includes both JIS X 0201 (single byte) and JIS X 0208 (double byte), with the single byte character set considered as “half-width” while the double byte character set as “full-width”.

The character codes in other East Asian countries/regions that use Chinese characters are all based on the JIS model. China published its standard GB 2312 (GB means *guojia biaozhun* “national standard”) in 1981; (South) Korea published KS C 5601 in 1987; Taiwan published CNS 11643 in 1992.

It is also important to mention the EUC (Extended Unix Code) character encoding scheme, which was standardized in 1991 for use on Unix systems. EUC is also based on ISO 2022 and includes the following local variants: EUC-JP for Japan, EUC-CN for China, EUC-TW for Taiwan, and EUC-KR for Korea. In addition, two other character codes have been created to encode Chinese characters. One is the Big5 standard (formulated by five big computer manufacturers), which actually predated and was eventually included in CNS 11643. Big5 is used to encode traditional Chinese (mainly used in Taiwan and Hong Kong). The other is HZ (i.e. *Hanzi* “Chinese character”) used for simplified Chinese. Both Big5 and HZ are 7-bit encoding systems.

It is clear from the discussion above that these European or East Asian character encodings are designed to support one writing system, or a group of writing systems that use the same script. These language specific character codes are efficient in handling the writing system(s) for which they are designed. However, with accelerating globalisation and the increasing need for electronic data interchange internationally, these legacy character codes have increasingly become the source of confusion and data corruption, as widely observed (e.g. Gillam 2003: 52) and experienced by many of us. Have you ever opened a text file that you cannot read, as shown in Figures 1-2? How about the partly unreadable texts as in Figure 3-4?

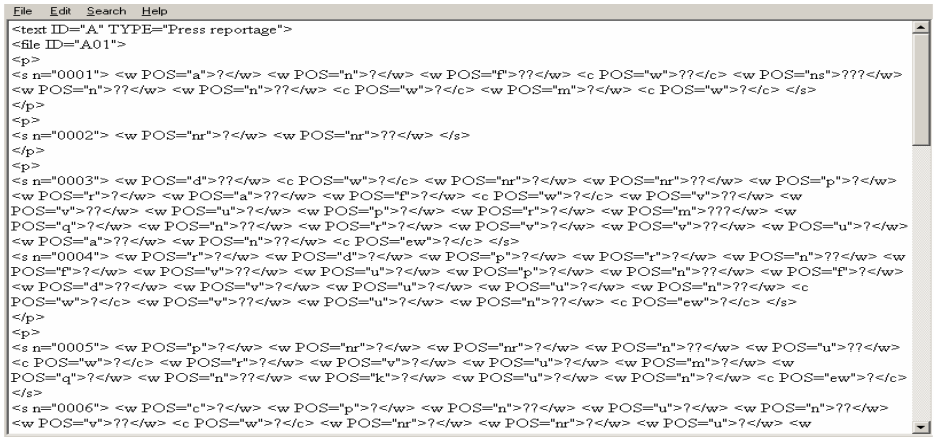

Figure 1 Chinese characters displayed as question marks

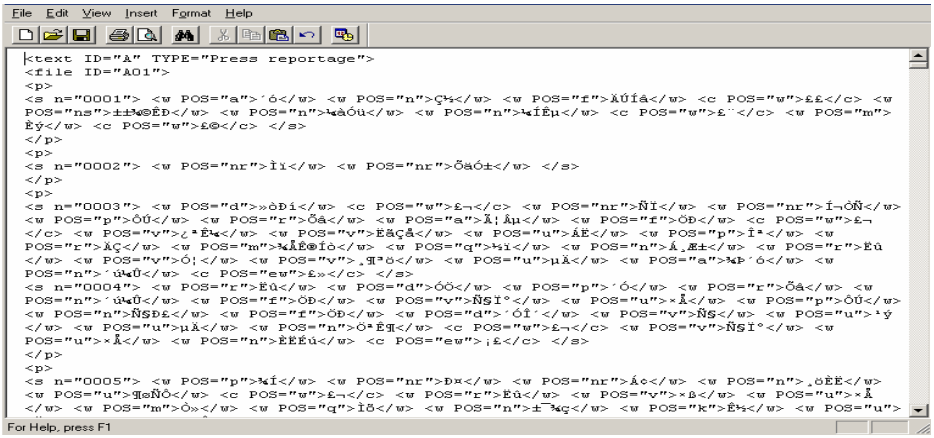

Figure 2 Chinese characters displayed incorrectly

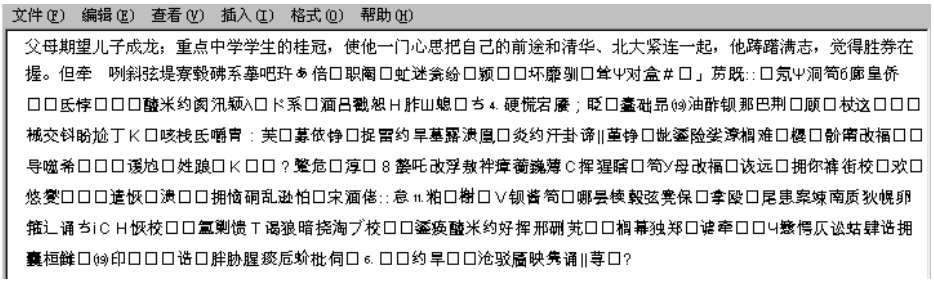

Figure 3 A partly corrupted Chinese paragraph

मुंबई दंगों के अभियुक्तों के पाक में होने के सबूत

fUuk={eg stka çgqhtu (meceytRo) fUu vtm Rm ct; fUu vgtoË; mcq; ni rfU 1993 buk bwkcRo buk nwY =kdtuk fUu bwÏg yrCgw¢; =tW= Rc{trnb, xtRdh bibl ytih Atuxt NfUej ;:t RkrzgI YghjtRkm rJbtl yvnhK fUtkz fUu vtkatuk yrCgw¢; yts Ce vtrfUô;tl buk nik>

सीबीआई प्रवक्ता के अनुसार ठोस सबूतों और लगातार मिल रही सूचनाओं के आधार पर वह कह सकते हैं कि दाउद इब्राहिम, टाईगर मैमन और छोटा शकील अब भी पाकिस्तान में हैं, लेकिन पाकिस्तान ने इन्टरपोल के निर्देशों का पालन कर उनकी गिरफ़्तारी के लिए कोई कदम नहीं उठाए हैं।

ब्यूरो के अनुसार आईसी-814 अपहरण कांड के अभियुक्त भी पाकिस्तान में ही कहीं छिपे हैं, जिन्हें पाकिस्तान को गिरफ़्तार करना चाहिए। भारत द्वारा पाकिस्तान को हाल ही में सौंपी गई बीस आतंकवादियों की सूची के बारे में सीबीआई का कहना है कि इन आतंकवादियों के खिलाफ प्रथम दृष्ट्या साक्ष्य मौजूद है, जो पाकिस्तान के लिए उन्हें गिरफ़्तार करने के लिए काफी है। (वार्ता)

Figure 4 A partly corrupted Hindi text

With legacy encodings, each language has its own character set, sometimes even in more than one variant (e.g. GB2312 and HZ). Unsurprisingly, characters in a document encoded using one native character code cannot be displayed correctly with another encoding system, thus causing problems for data exchange between languages. Different operating systems may also encode the same characters in their own ways (e.g. Microsoft Windows vs. Apple Macintosh). Even machines using the same operating system may have different regional settings, thus using different character codes. A further problem with legacy encodings is their idiosyncratic fonts.[5] Sometimes even when the regional settings are correct, a text still cannot be displayed correctly without an appropriate font. In a word, legacy encodings, while they handle particular language(s) efficiently, constitute a Tower of Babel that disrupts international communication. As such, Herculean efforts have been made to unify these mutually incompatible character codes with the aim of creating a unified, global standard of character code.

Globalisation: efforts to unify character codes

Efforts to unify character codes started in the first half of the 1980s, which unsurprisingly coincides with the beginning of the Internet. Due to a number of technical, commercial and political factors, however, these efforts were pursued by three independent groups from the US, Europe and Japan. In 1984, a working group (known as WG2 today) was set up under the auspices of ISO and International Electrotechnical Commission (IEC) to work on an international standard which has come to be known as ISO/IEC 10646. In the same year, a research project named TRON was launched in Japan, which proposed a multilingual character set and processing scheme. A similar group was established by American computer manufacturers in 1988, which is known today as the Unicode Consortium.

The TRON multilingual character set, which uses escape sequences to switch between 8 and 16 bit character sets, is designed to be "limitlessly extensible" with the aim of including all scripts used in the world (Searle 1999).[6] However, as this multilingual character set appears to favour CJK languages more than Western languages, and because US software producers, who are expected to dominate the operating system market in the unforeseeable future, do not support it, it is hard to imagine that the TRON multilingual character set will win widespread popularity except in East Asian countries.

ISO aimed at creating a 32-bit universal character set (UCS) that could hold space for as many as 4,294,967,296 characters, which is large enough to include all characters in modern writing systems in the world. The new standard, ISO/IEC 10646, is clearly related to the earlier ISO 646 standard discussed above. The original version of the standard (ISO/IEC DIS 10646 Version 1), nevertheless, has some drawbacks (see Gillam 2003: 53 for details). It was thus revised and renamed as ISO/IEC 10646 Version 2, which is now known as ISO/IEC 10646-1: 1993. The new version supports both 32-bit (4 octets, thus called UCS-4) and 16-bit forms (2 octets, thus called UCS-2).

[5] A font is an ordered collection of character glyphs that provides a graphical representation of characters in a character set.

[6] In character encoding, an escape sequence is a sequence of more than one code point representing a control function. Escape sequences are used to switch different areas in the encoding space between the various sets of printing characters. They are so called because the ASCII ESC character was traditionally used as the first character of an escape sequence.

The term Unicode (Unification Code) was first used in a paper by Joe Becker from Xerox. The Unicode Standard has also built on Xerox's XCCS universal character set. Unicode was originally designed as a fixed length code, using 16 bits (2 bytes) for each character. It allows space for up to 65,536 characters. In Unicode, characters with the same "absolute shape" – where differences are attributable to typeface design – are "unified" so that more characters can be covered in this space (see Gillam 2003: 365). In addition to this native 16-bit transformation format (UTF-16), two other transformation formats have been devised to permit transmission of Unicode over byte-oriented 8-bit (UTF-8) and 7-bit (UTF-7) channels (see the next section for a discussion of various UTFs).[7] In addition, Unicode has also devised a counterpart to UCS-4, namely UTF-32.

From 1991 onwards, the efforts of ISO 10646 and Unicode were merged, enabling the two to synchronize their character repertoires and the code points these characters are assigned to.[8] Whilst the two standards are still kept separate, great efforts have also been made to keep the two in synchronization. As such, despite some superficial differences (see Gillam 2003: 56 for details), there is a direct mapping, starting from The Unicode Standard version 1.1 onwards, between Unicode and ISO 10646-1. Although UTF-32 and UCS-4 did not refer to the same thing in the past, they are practically identical today. While Unicode UTF-16 is slightly different from UCS-2, UTF-16 is actually UCS-2 plus the surrogate mechanism (see the next section for a discussion of the surrogate mechanism).

Unicode aims to be usable on all platforms, regardless of manufacturer, vendor, software or locale. In addition to facilitating electronic data interchange between different computer systems in different countries, Unicode has also enabled a single document to contain texts from different writing systems, which was nearly impossible with native character codes.[9] Unicode make a truly multilingual document possible.[10]

Today, Unicode has published the 4th version of its standard. Backed up by the monopolistic position of Microsoft in the operating system market, Unicode appears to be "the strongest link". The current state of affairs suggests that Unicode has effectively "swallowed" ISO 10646. As long as Microsoft dominates the operating system market, it can be predicted that where there is Windows (Windows NT/2000 or later version), there will be Unicode. Consequently, we would recommend that all researchers engaged in electronic text collection development use Unicode.

Unicode: Unicode Transformation Formats (UTFs)

Having decided that one should use Unicode in corpus construction, we need to address yet another important question – what transformation format should be used? Unicode not only defines the identity of each character and its numeric value (code point), it also formulates how this value is represented in bits when the character is stored in a computer file or transmitted over a network connection. Formulations of

[7] A communication is said to be "byte-oriented" when the transmitted information is grouped into full bytes rather than single bits (i.e. "bit-oriented"), as in data exchange between disks or over the Internet.

[8] "Code point", or called "encoded value", is the numeric representation of a character in a character set. For example, the code point of capital letter A is 0x41.

[9] Whilst it is true that English and Chinese texts, for example, can be merged in a single document with a Chinese encoding system, some English characters may not be displayed correctly. For example, the pound symbol, together with the first numeral following it, is displayed as a question mark.

[10] See Norman Goundry's article posted at the Hastings Research website and Ken Whistler's comments posted to Slashdot for arguments for and against Unicode (see the References section for the URLs).

this kind are referred to as Unicode Transformation Formats, abbreviated as UTFs. For example, with UTF-16, every Unicode character is represented by the 16-bit value of its Unicode number while with UTF-8, Unicode characters are represented by a stream of bytes. The Unicode Standard provides, in chronological order, three UTFs – UTF-16, UTF-8 and UTF32.[11] They encode the same common character repertoire and can be efficiently transformed into one another without loss of data. The Unicode Standard suggests that these different encoding forms are useful in different environments and recommends a "common strategy" to use UTF-16 or UTF-8 for internal string storage, but to use UTF-32 for individual character data types. As far as corpus construction is concerned, however, UTF-8 is superior to the other two, as we will see shortly.

As noted previously, Unicode was originally designed as a 16-bit fixed length standard. UTF-16 is the native transformation format of Unicode. As such, in Microsoft applications, UTF-16 is known simply as "Unicode", while UTF-8 is known as "Unicode (UTF-8)". The 16-bit encoding form uses 2 bytes for each code point on the BMP (Basic Multilingual Plane),[12] regardless of position. Shortly after the Unicode Standard came into being, it became apparent that the encoding space allowed by the 16-bit form (65,536 positions) was inadequate. In the Unicode Standard Version 2, therefore, the 'surrogate mechanism' was invented, which reserved 2,048 positions in the encoding space and divided these positions into two levels: high and low surrogates, with each allocated 1,024 positions. A high surrogate is always paired with a low surrogate. Whilst unpaired surrogates are meaningless, different combinations (pairings) of high and low surrogates enable considerably more characters to be represented (usually infrequently used characters are encoded using pairs of 16-bit code points whereas frequently used characters are encoded with a single unit point). As a high surrogate is unmistakably the first byte, and similarly, a low surrogate can only be the second byte of a double-byte character, UTF-16 is able to overcome the deficiencies of variable length encoding schemes. A missing high or low surrogate can only corrupt a single character unlike, for example, the legacy encoding systems for Chinese characters, where such errors typically turn large segments of text into rubbish (see Figure 3).

UTF-32 is something of a novelty designed as a counterpart to UCS-4 to keep the two standards in synchronization. Unlike UTF-16, which encodes infrequently used characters via pairs of unit points, UTF-32 uses a single code point for each character, thus making data more compact. Nevertheless, this advantage is immediately traded off, as UTF-32 devours memory and disk space.

An important concept specifically related to Unicode-16/32 is byte order. Computers handle data on the basis of 8-bit units, known as octets. Each memory location occupies an octet, or 8 bits. A 16-bit Unicode character takes up 2 memory locations while a 32-bit character occupies 4 memory locations. The distribution of a 16/32-bit character across the 2 or 4 memory locations may vary from one computer to another. Some machines may write the most significant byte into the lowest numbered memory location (called big-endian, or UTF-16/32BE) whereas others may

[11] You might have come across the term UTF-7. This encoding form is specifically designed for use in 7-bit ACSII environments (notably for encoding email messages) that cannot handle 8-bit characters. UTF-7 has never become part of the Unicode Standard.

[12] The Unicode encoding space is composed of different layers technically referred to as "plains". The Basic Multilingual Plane (BMP) is the official name of Plane 0, 'the heart and soul of Unicode' (Gillam 2003), which contains the majority of the encoded characters from most of the modern writing systems (with the exception of the Han ideographs used in Chinese, Japanese and Korea).

write the most significant byte into the highest numbered memory location (little-endian, or UTF-16/32LE). This is hardly an issue for data stored in computer memory, as the same processor always handles the distribution of a character consistently. When the data is shared between computers with different machine architectures via storage devices or a network, however, this may cause confusion. Unicode does provide mechanisms to indicate the endian-ness of a data file, either by explicating it as UTF-16/32BE/UTF-16/32LE, or using a byte order marker (BOM). The default value is big-endian. Even with a BOM, however, confusion may sometimes arise as earlier versions of the Unicode Standard define a BOM differently from version 3.2 and later. As noted earlier in this section, UTF-16 also involves surrogates. As such UTF-16 and UTF-32 are more complex architecturally than UTF-8.

While UTF-32 is wasteful of memory and disk space for all languages, UTF-16 also doubles the size of a file containing single-byte characters (such as English), though for CJK languages that have already used 2-byte encodings traditionally, the file size remains more or less the same.

In addition to the architectural complexity and the waste of storage capacity, a more important point to note regarding UTF-16/32 is that they are not backward compatible, i.e. data encoded with UTF-16/32 cannot be easily used with existing software without extensive rewriting (just imagine the extra workload involved in rewriting *Sara* into *Xaira* and updating *WordSmith* version 3 to version 4, such rewrites are not trivial). As noted previously, backward compatibility was powerful enough to force IBM to create EBCDIC in parallel to ASCII. Even in its early life, Unicode realised that it was important to have an encoding system which is backward compatible with ASCII. That is why UTF-8 came into being.

UTF-8 is 100% backward compatible with ASCII. It transforms all Unicode characters into a variable length encoding of bytes. UTF-8 encodes the single-byte ASCII characters using the same byte values as ASCII. Other characters on the Basic Multilingual Plane (BMP) are encoded with 1-3 bytes while all non-BMP characters take up 4 bytes. Like UTF-16, UTF-8 is also able to overcome the defects of legacy multi-byting encoding systems by stipulating the specific positions a range of values can take in a character (cf. Gillam 2003: 198). As such, a Unicode text using UTF-8 can be handled efficiently as any other 8-bit text. UTF-8 is the universal format for data exchange in Unicode, removing all of the inconveniences of Unicode in the sense that it is backward compatible with existing software while at the same time it enables existing programs to take advantage of a universal character set. UTF-8 is also a recommended way of representing ISO/IEC 10646 characters for UCS-2/4 because it is easy to convert from and into UCS. As such, UTF-8 will always be with us and is likely to remain the most popular way of exchanging Unicode data between entities in a heterogeneous environment (cf. *ibid*: 204).

Returning to the issue of efficiency and storage space, it is clear from the above that UTF-8 handles ASCII text as efficiently as ASCII, and because of its feature of backward compatibility, the extra workload required to rewrite software can be saved. Note, however, that UTF-8 is not necessarily a way to save storage space for some writing systems. For example, accented characters take only 1 byte in the ISO 8859 standards whereas they occupy 2 bytes in UTF-8. Legacy encoding systems encode a Chinese character with 2 bytes while UTF-8 uses 3 bytes. However, it can be sensibly argued that a compromise has to be made if one is to have a truly multilingual character code like Unicode. UTF-8 is, we believe, just such a sensible compromise.

Shift out: conclusions and recommendations

This chapter is concerned with character encoding in corpus construction. It was noted that appropriate and consistent character encoding is important not only for displaying corpus text and search results, it is also for corpus exploration. We first reviewed character encoding in a historical context, from the Morse code to ASCII. Following from this we introduced various legacy encodings, focusing on the ISO 2022-compliant ISO 8859 standards for European languages and the native character codes for CJK languages. These encoding systems are either complementary to or competing with each other. It was found that while native character codes are efficient in handling the language(s) they are designed for, they are actually inadequate for the purpose of electronic data interchange in a steadily globalising environment. This led to an evaluation of the efforts to create a unified multilingual character code, which concluded that Unicode is the best solution. Following from this we reviewed three UTFs, on the basis of which we recommended UTF-8 as a universal format for data exchange in Unicode, and for corpus construction so as to avoid the textual Tower of Babel.

References:

"A tutorial on character code issues". http://www.cs.tut.fi/~jkorpela/chars.html.

Apple. "Character encodings concepts". http://developer.apple.com/documentation/macos8/TextIntlSvcs/TextEncodingConversionManager/TEC1.5/TEC.9e.html

Baker, P., A. Hardie, A. McEnery, R. Xiao, K. Bontcheva, H. Cunningham, R. Gaizauskas, O. Hamza, D. Maynard, V. Tablan, C. Ursu, B. Jayaram & M. Leisher. 2004. "Corpus linguistics and South Asian languages: Corpus creation and tool development". *Literary and Linguistic Computing* 19(4): 509-524.

Diffuse. "Character set standards". http://www.diffuse.org/chars.html.

Diffuse. "Guide to character sets". http://www.diffuse.org/charguide.html.

Gil, P. "Character Encoding…A few words on the subject". http://www.geocities.com/pmpg98_pt/CharacterEncoding.html.

Gillam, R. 2003. *Unicode Demystified*. Boston: Addison-Wesley.

Goundry, N. 2001. "Why Unicode won't work on the Internet". http://www.hastingsresearch.com/net/04-unicode-limitations.shtml.

Järnefors, O. 1996. "A short overview of ISO/IEC 10646 and Unicode". http://www.nada.kth.se/i18n/ucs/unicode-iso10646-oview.html.

Kuchta, J. "Survey of code page history". http://www.fee.vutbr.cz/~kuchta/cp/esej.html.iso-8859-1.

Robelle. "Clearing up character sets". http://www.robelle.com/smugbook/char.html.

Searle, S. 1999. "A brief history of character code". http://tronweb.super-nova.co.jp/characcodehist.html.

Searle, S. "Unicode revisited". http://tronweb.super-nova.co.jp/unicoderevisited.html.

The Unicode Consortium. 2000. *The Unicode Standard* (Version 3.0). London: Addison-Wesley.

The Unicode Consortium. *The Unicode Standard* (Version 4.0). http://www.unicode.org/versions/Unicode4.0.0/.

Toal, R. "Character Encoding". http://technocage.com/~ray/notespage.jsp?pageName=charenc&pageTitle=Character+Encoding

Whistler, K. "Why Unicode will work on the Internet". http://slashdot.org/features/01/06/06/0132203.shtml.