Bits, bytes, and Unicode: Digital text for linguists

James A. Crippen

28 January 2016

Contents

1. Introduction

In this article I describe the essential concepts behind digital representation of text and particularly the structure and use of Unicode, emphasizing issues that are of particular importance for linguists. Linguists spend a large amount of their professional lives working with textual information, whether this involves language data, reports and publications, or emails. Linguists work with language data in orthographies and in phonetic transcriptions, and many also use mathematical notations for both theoretical and descriptive purposes. Essentially all textual work is digital today, with the only major exception being handwritten notes. Although linguists work with textual data on a daily basis, most are only dimly aware of how digital text is represented and manipulated. When technological problems rear their ugly heads, the uninformed linguist has little idea of how to find solutions.

In section 2 I outline some basic concepts for describing writing systems. In section 3 I discuss the essential concepts of the binary number system and the organization of binary numbers into larger units in computer architectures. Section 4 lays out the ideas behind the representation of textual information with binary numbers. I describe the structure of the Unicode character set in section 5, the primary encodings of the Unicode character set in section 6, and other issues like case and sorting that are standardized as part of Unicode in section **??**. Finally in section 7 I outline a few practical issues in using Unicode and digital text that are significant for the working linguist.

2. Concepts in writing systems

The phonetic–phonemic distinction in the analysis of spoken language is familiar to every linguist. Written language can be similarly analyzed, with segments and constituents in different levels of analysis. The fundamental unit of written linguistic communication is the GRAPH, analogous to the 'phone' which is the basic segment in speech. A GRAPHEME is an abstract category denoting a set of graphs that are related in some way in a particular writing system. The grapheme is analogous to the phoneme in speech. An example of a grapheme is ⟨x⟩ as shown in (1). This is a unit in English writing called 'ex' /ɛks/ that represents many different graphs.

(1) ⟨x⟩ = {x, *x*, **x**, ***x***, 𝒳, 𝔵, 𝕩, ×, *×*, ｘ, ...} *a grapheme and its graphs*

The grapheme ⟨x⟩ can be found as a variety of different graphs depending on context, but all are recognized as representing the same orthographic element in English. The distinction between graph and grapheme is often ignored in practice, but it is particularly significant for digital representation of text. Some characters in digital

text represent a single grapheme, others represent particular graphs of a grapheme. I return to this issue several times in this article.

DIGRAPHS are graphemes composed of two other graphemes like ⟨ph⟩ /f/ as in ⟨phone⟩ /foʊn/. There are also trigraphs like ⟨tch⟩ /tʃ/ in ⟨hitch⟩ /hɪtʃ/ and tetragraphs like ⟨ough⟩ /oʊ/ in ⟨though⟩ /ðoʊ/, and even one pentagraph ⟨tzsch⟩ /tʃ/ in the proper name ⟨Nietzsche⟩ /ˈniˌtʃʌ/ borrowed from German. English does not have any hexagraphs, but Modern Irish has e.g. ⟨eidhea⟩ /əi̯/ as in ⟨eidheann⟩ /əi̯nʸ/ 'ivy'. Any grapheme that appears in polygraphs is necessarily context dependent; the ⟨h⟩ in English is particularly so since it appears in many polygraphs and also monographically represents both /h/ as in ⟨happy⟩ /ˈhæpi/ and /∅/ as in ⟨an history⟩ /æn‿ˈɪstɹi/.

Other more complex constituents in writing are well known such as words, sentences, and paragraphs. These are not universal categories in every writing system, but they are very common and have historically tended to be added into writing systems that lack them. Digital text representations of these constituents are largely the same as those in analogue text. They are outside the scope of this article so I do not describe them any further.

A WRITING SYSTEM is a set of graphemes and a mapping between them and some set of speech units like phones, phonemes, syllables, words, etc. A special case of writing system is a TRANSCRIPTION SYSTEM that is not tied to a particular language; the canonical example of a transcription system is the IPA. Two different writing systems can use the same grapheme for different purposes: English often uses ⟨x⟩ to denote the velar stop and alveolar fricative sequence /ks/, but the IPA uses the same grapheme ⟨x⟩ to denote the voiceless velar fricative [x] that does not exist in English.

The term SCRIPT idenotes a family of writing systems based around a set of shared graphemes. Many languages use the Latin script such as English, Vietnamese ⟨tiếng Việt⟩, Yoruba ⟨èdè Yọ̀rùbá⟩, and Guaraní. The Cyrillic script is used for many Slavic languages like Russian and Bulgarian, but also for many non-Slavic languages like Mari (Uralic), Tajik (Iranian), Kazakh (Turkic), Buryat (Mongolic), Evenki (Tungusic), and Chukchi (Paleosiberian). The Devanagari script is used for many South Asian languages like Hindi, Nepali, Konkani, and Bodo. There are also scripts used for a single language like the Cherokee syllabary for Cherokee and Thaana for Maldivian.

The same graph can appear in different scripts. The graph ⟨T⟩ denotes a phoneme like /t/ in all three of the Latin, Cyrillic, and Greek scripts. But the graph ⟨H⟩ denotes something like /h/ in Latin, /ɛː/ or /i/ in Greek, and /n/ in Cyrillic. The Cherokee ⟨A⟩ is visually related to Latin, Greek, and Cyrillic ⟨A⟩ but Cherokee ⟨A⟩ denotes /ɡo/ and /a/ is represented by ⟨D⟩. Digital text systems usually represent these kinds of cross-script homographs as unrelated characters despite similar visual appearances and even related etymologies. Thus in Unicode there are separate characters for each

of the Latin, Greek, and Cyrillic graphs that have the form ⟨T⟩ even though they all denote roughly the same phonemes.

The graphemes presented so far have all been LETTERS that stand for units in spoken language like phonemes or syllables. NUMBERS are graphemes that represent mathematical quantities. Some graphemes have special purposes which may not be part of spoken language. The category of PUNCTUATION includes graphemes like ⟨ ⟩ 'space' that separates words, ⟨“⟩ 'left double quote' that indicates the beginning of a quotation, and ⟨.⟩ 'period; full stop' that indicates an abbreviation or the end of a sentence. There are also DIACRITICS such as the ⟨◌́⟩ 'acute accent', ⟨◌̃⟩ 'tilde', ⟨◌̨⟩ 'ogonek', ⟨◌̐⟩ 'chandrabindu', ⟨◌̛⟩ 'horn', and ⟨◌̉⟩ 'hook'. Diacritics are added to base graphemes to produce complex monographs like ⟨á⟩, ⟨ã⟩, ⟨ą⟩, ⟨å⟩, ⟨a̛⟩, and ⟨å⟩. The SYMBOLS in a writing system may stand for words or phrases or may have textual functions like ⟨$⟩ 'dollar sign', ⟨&⟩ 'ampersand', ⟨#⟩ 'octothorpe', ⟨¶⟩ 'paragraph mark', ⟨*⟩ 'asterisk', ⟨№⟩ 'numero', ⟨™⟩ 'trademark', and ⟨♡⟩ 'heart'. Unicode specifies these distinct categories along with several others which I discuss further in section 5.

There are a few typographic distinctions useful in discussing writing systems. Typography is, according to the OED, "the action or process of printing; esp. the setting and arrangement of types and printing from them; typographical execution; hence, the arrangement and appearance of printed matter". Since the invention of the printing press until the mid-20th century, printing involved pressing small metal blocks with ink on them onto paper to form permanent images of written language. Each metal block with a graph on its surface is a TYPE. A collection of types all based on a single design is a TYPEFACE or just FACE. The subset of a typeface in a single height like 12 or 14 points is a FONT or FOUNT (1 point = $\frac{1}{72}$ inch, ≈ 0.353 mm). The distinction between typeface and font has become blurred because digital typefaces are usually resizable and thus designed as a single font. Latin, Cyrillic, and Greek scripts have specialized style distinctions between ROMAN/UPRIGHT, ITALIC/CURSIVE, and BOLD, but these do not necessarily exist in other scripts.

Unicode and other standards for digital text are careful to distinguish between typographic or 'formatting' features like size and style versus graphemic features like numbers versus letters. Digital text is graphemic whereas typography is graphic, corresponding roughly to the phonology–phonetics distinction in linguistics.

⟦FIXME: Digital typography and glyphs. A glyph is a graph that is implemented in a font. The glyph representing a character is not standardized and its appearance is subject to the whims of the font designer, the display and rendering software, and the medium in which the text is displayed (paper, screen, sign, laser, etched metal, etc.). Crucially, grapheme ≠ character ≠ glyph.⟧

bin	*hex*	*dec*	*bin*	*hex*	*dec*	*bin*	*hex*	*dec*	*bin*	*hex*	*dec*
0	0	0	1000	8	8	10000	10	16	11000	18	24
1	1	1	1001	9	9	10001	11	17	11001	19	25
10	2	2	1010	A	10	10010	12	18	11010	1A	26
11	3	3	1011	B	11	10011	13	19	11011	1B	27
100	4	4	1100	C	12	10100	14	20	11100	1C	28
101	5	5	1101	D	13	10101	15	21	11101	1D	29
110	6	6	1110	E	14	10110	16	22	11110	1E	30
111	7	7	1111	F	15	10111	17	23	11111	1F	31

Table 1: Binary, hexadecimal, and decimal numbers from 0 to 2^4–1 = 31.

3. Bits and bytes

Computers were first invented for numeric processing and in essence everything since has been an extension of numeric processing. Today essentially all computers use BINARY numbers with base-two arithmetic. In the binary number system the basic unit is the BIT with a value of either 0 or 1; compare this with the decimal unit of a DIGIT with a value from 0 to 9. The symbols 0 and 1 are arbitrary but convenient; binary values could be just as well be represented with symbols like ⟨–⟩ and ⟨+⟩, ⟨◯⟩ and ⟨●⟩, or even ⟨ቀ⟩ and ⟨ኮ⟩ among many other possibilities.

Mathematical logic using binary numbers is often known as either BOOLEAN LOGIC or BOOLEAN ALGEBRA after its inventor George Boole, and thus binary values are sometimes known as BOOLEAN VALUES when they represent 'false' and 'true'. Digital text rarely requires operation on individual bits and the truth value of characters is usually irrelevant, so boolean algebra is normally not invoked.

Each bit in a binary number represents multiplication by a power of two; compare how decimal digits represent multiplication by powers of ten. The lowest (rightmost) bit b_0 in a binary number represents the value $b_0 \times 2^0$, the next higher (leftward) bit b_1 represents the value $b_1 \times 2^1$, etc. The binary number 101 thus has the value (1 × 2^2) + (0 × 2^1) + (1 × 2^0) = 4 + 0 + 1 = 5. Table 1 gives the first 32 binary numbers starting from zero, along with decimal and hexadecimal representations.

Binary numbers are very difficult for humans to read when their values are large. Even a highly experienced computer hardware engineer would need some time to interpret a long string like 11011110101011011011111011101111. Hence binary numbers are almost always represented using some other number system with a more compact representation. The two most often encountered are octal and hex-

adecimal. The OCTAL number system is base 8 so that each unit is a multiple of a power of 8 and has a value from 0 to 7. The HEXADECIMAL or 'hex' system is base 16 so each unit is a multiple of a power of 16 and has values from 0 to 15. Octal used to be more popular up until the mid-1980s when hexadecimal became dominant. I only detail hexadecimal here.

A hexadecimal unit, also known as a HEXIT by analogy with 'bit' and 'digit' ⟦FIXME: ref HDic⟧, represents more possible values than a decimal digit, so more than 10 graphemes are necessary to represent the possible hexadecimal values. Because the hexadecimal number system was primarily developed by users of the Latin script, one counts from 0 to 9 and then from A to F.[1] I show in (2) how the hexadecimal number $CAFE_{16}$ is converted to decimal by positional multiplication of powers of 16 and subsequent addition.

(2) $CAFE_{16} = (C_{16} \times 16^3) + (A_{16} \times 16^2) + (F_{16} \times 16^1) + (E_{16} \times 16^0)$ *hex→dec conversion*
$= (12 \times 16^3) + (10 \times 16^2) + (15 \times 16^1) + (14 \times 16^0)$
$= (12 \times 4096) + (10 \times 256) + (15 \times 16) + (14 \times 1)$
$= 49152 + 2560 + 240 + 14$
$= 51966$

A hexit conveniently represents a chunk of four bits since the largest hexadecimal unit F is binary 1111. Binary numbers are thus broken into four-bit chunks corresponding to hexits as shown in (3).

(3) 1101 1110 1010 1101 1011 1110 1110 1111 *bin–hex correspondence*
D E A D B E E F

Because hexadecimal shares graphemes with decimal and binary, there are conventional 'stigmata' to indicate the hexadecimal base (a.k.a. radix). Mathematicians and computer scientists often use a suffix subscript, so e.g. binary 1111_2 = hexadecimal F_{16} = decimal 15_{10}. Programming languages have a variety of other stigmata such as the C language's ⟨0x⟩ prefix, Common Lisp's ⟨#x⟩ prefix, Pascal's ⟨$⟩ prefix, Algol 68 and Smalltalk's ⟨16r⟩ prefix, and Intel assembler's suffix ⟨h⟩ or ⟨H⟩. The same hexadecimal number $FACE_{16} = 64206_{10}$ is shown in (4) using all these notations.

(4) a. $FACE_{16}$
b. 0xFACE
c. #xFACE
d. $FACE
e. 16rFACE
f. FACEh
hexadecimal stigmata

1. Unlike decimal numbers in English, hexadecimal numbers have no complex linguistic morphology. Hexits are simply read linearly: one reads $1D_{16}$ as /wʌn di/ and not as say */di tin/. But hex numbers that look like words can be pronounced as such, e.g. $FADE_{16}$ /feɪd/ versus /ɛf eɪ di i/.

Computers do not normally process bits alone, but instead operate on larger chunks of bits. The most common of these chunks is the BYTE. This is eight bits such as 1111 0010 = $F2_{16}$ = 242_{10}. Historically byte sizes ranged between six bits and nine bits depending on the computer architecture, but the eight-bit byte has become universal. The special term OCTET is used to explicitly denote an eight-bit chunk, often in computer networking where different computers might have different byte sizes.

Larger chunks of bits are used widely but their labels vary and are dependent on particular computer architecutres: 'nybble', 'halfword', 'gawble', 'dword', 'deckle', etc. The term 'quarter' can punningly represent two bits by analogy with the one eighth 'bit' of a historical Spanish dollar (the stereotypical pirate's 'pieces of eight'). A 'word' is specific to a particular architecture, being the number of bits that the processor operates on in a single instruction cycle. Modern word sizes are 32 bits and 64 bits.

In working with digital data, it is essential to keep in mind that everything in a computer is binary and is thus composed entirely of bits and bytes. Digital text is long strings of bytes, digital audio and video are made of complex multidimensional matrices of bytes, computer programs are long strings of bytes representing both instructions and data, and the image on a computer monitor is a two dimensional matrix of triplets (red, green, blue) represented by multiple bytes.

Computers do not understand[2] that there are differences between bits used for one purpose like audio or images and bits used for another purpose like computer code or digital text. Instead it is humans who define and manipulate the contexts in which bits are interpreted. These contexts give meaning to bits by the behaviour of the computer and its user interfaces, and the contexts are implemented in the computer by means of computer programs. A linguistic analogy is the meaninglessness of sounds: a speech sound obtains meaning only by its context in a defined linguistic system.

A harsh consequence follows from the computer's ignorance about meaning. Digital data is entirely useless without a human – or a program designed by a human – to contextualize the data meaningfully. By maintaining computer programs we prolong the useful life of data, but eventually computer architectures change and computer hardware breaks down making data useless. To ensure that data is still processable by other programs on different hardware, we establish standards for how bits are organized into kinds of data. A standard specifies that all computer programs should contextualize the data in the same way, allowing humans who interact with these

2. There is an old joke in computing: don't anthropomorphize computers because they don't like it. Treating computers as particularly stupid humans is a convenient metaphor for thinking about them, even if it is obviously false. We thus ascribe intention, understanding, mental state, etc.

programs to extract similar meanings from their behaviour with the data. Thus an MP3 audio file may be processed by many different programs running on a wide variety of computers, but because the data format is standardized, humans will hear the same audio played by all of these different programs.

4. Characters

I have used the term 'digital text' without defining it. Here I define DIGITAL TEXT as any written linguistic or paralinguistic information represented in a binary code. This depends on the terms 'linguistic' and 'paralinguistic' which I leave for philosophical debate, since symbols like ⟨♀⟩ and ⟨☞⟩ have an arguable status in language. In practice, digital text is anything that can be encoded with a character set like Unicode and which is meant for reading by humans. This includes plain text, most content of PDFs and Word documents, and written material on the World Wide Web.

The smallest unit of digital text is the CHARACTER which can be loosely described as a chunk of bits representing a graph or grapheme. The term 'letter' is inappropriate because characters include numbers, punctuation, diacritics, symbols, whitespace ('blanks'), formatting instructions, and many other entities that would not conventionally be considered 'letters'.

Every character is represented by a unique sequence of bits. The patterns of bits used to represent characters may have some logic in their organization, but they are often arbitrary and opaque to the user. Consider for example the Latin lowercase ⟨a⟩ which is represented by 0110 0001 = 61_{16} = 97_{10}. The lowest bit is intentionally a 1 because ⟨a⟩ is the first letter in the Latin alphabetic ordering, and the uppercase ⟨A⟩ has a similar representation 0100 0001 = 41_{16} = 65_{10} with 1 as the lowest bit. But the choice of ⟨@⟩ as 0100 0000 = 40_{16} = 64_{10} immediately preceding ⟨A⟩ and of ⟨`⟩ as 0110 0000 = 60_{16} = 96_{10} immediately preceding ⟨a⟩ is essentially arbitrary. The higher bits are often used to designate blocks of related characters, an issue I return to later.

Characters are composed of multiple bits but the exact number of bits per character varies depending on the character set. The I Ching (*Yì Jīng*) trigrams use the bit values ⟨–⟩ and ⟨--⟩ to form three-bit characters for a set of 2^3 = 8: {☰, ☱, ☲, ☳, ☴, ☵, ☶, ☷}. Braille is a six-bit system of blanks and dots in a 2×3 matrix: ⟨⠿⠾⠽⠼⠻⠺⠹⠸⟩ = ⟨BRAILLE⟩. Morse code uses a variable number of bits valued ⟨·⟩ and ⟨–⟩ per character with a maximum of six: ⟨–– ––– ·–· ··· ··––·–·⟩ = ⟨MORSE@⟩.

4.1. Character sets and encodings

Each system like the I Ching and Morse code defines a CHARACTER SET (or 'charset') which is a collection of characters and the specification of their properties. A char-

acter set defines what characters should exist in a system, their visual appearances, and how they should behave in various contexts.

A character set does not specify the binary representation of characters. Instead, the binary representation is specified by a CHARACTER ENCODING (usually just 'encoding'). The distinction between character set and character encoding is often ignored because they are usually defined in the same standard documentation. For example, the MS-DOS term CODE PAGE applies to both character sets and their encodings. But when comparing different systems the distinction between character set and encoding is useful, and it is indispensable for understanding Unicode which defines several encodings for a single character set.

The term REPERTOIRE is the range of different characters available in a character set. ASCII has a very small repertoire that is essentially only suitable for English. ISO 8859-1 has a slightly larger repertoire that is suitable for several western European languages, but still does not cover many common letters in the Latin script. Unicode has an extremely large repertoire, in fact the largest of all character sets. It is continually growing with the addition of support for more scripts and writing systems.

A CODE POINT is a single element in a specific character set. A code point is usually defined as an element in a two or three dimensional matrix of binary codes, thus denoting a position in the space of the character set. The codes of code points are implicitly a kind of binary encoding, but this is not necessarily the only encoding for a character set. The code point of ⟨é⟩ in the ISO 8859-1 character set is $E9_{16}$ which is also its encoding. The code point of ⟨é⟩ in Unicode is also $E9_{16}$ but it is most commonly represented as $C3A9_{16}$ in the UTF-8 encoding. Unicode thus necessitates a distinction between the encoded form of a character and its code point.

The most widely used character set was ASCII /ˈæsˌki/ until the recent primacy of Unicode in the past decade. ASCII stands for the American Standard Code for Information Interchange and was developed in the early 1960s from earlier teletype systems like Baudot code and ITA2. ASCII is a seven-bit system so it can represent a maximum of $2^7 = 128$ characters. It is sufficient for representing basic English with limited punctuation, but lacks many characters like the en-dash ⟨–⟩ and the section sign ⟨§⟩. ASCII is unsuitable for European languages other than English because it completely lacks diacritics.

An early competitor to ASCII was EBCDIC /ˈɛbsiˌdɪk/, the Extended Binary Code for Information Interchange. This is an eight-bit system used by IBM for its mainframe computers. It suffered from some unfortunate design decisions as well as multiple incompatible versions so it eventually faded away. EBCDIC can still occasionally be encountered in data from old programs in the 1970s and 1980s.

Many eight-bit systems with $2^8 = 256$ possible characters were developed to ex-

tend ASCII. Some of these include the Mac Roman character set for the original Macintosh, CP 437 used by MS-DOS, Windows-1252 designed for Microsoft Windows, the and ISO/IEC 8859-*x* family for representing various subsets of European languages. Many other eight-bit systems were developed for non-Latin scripts like KOI8 (Код Обмена Информацией 8-битного *Kod Obmena Informatsiej 8-bitnogo*) for Cyrillic, ISCII for South Asian scripts like Devanagari, VISCII for Vietnamese, and TIS-620 for Thai. Although these systems generally share ASCII 7-bit code points, their 8-bit "high half" code points are largely incompatible.

As an example of the distinction between character set and encoding, consider the character that appears as ⟨a⟩ in the Latin script. This is defined as 61_{16} in ISO 8859-1b , Windows-1252, and Mac Roman. In contrast the character ⟨é⟩ is $E9_{16}$ in ISO 8859-1 and Windows-1252 but it is $8E_{16}$ in Mac Roman. Different character sets may have the same inventory of characters but they may entail encodings where a logically identical character has different binary representations. As I discuss in section 6, Unicode defines several incompatible encodings for a single character set, with each encoding meant to address different technological problems.

4.2. Conversion

The process of CONVERSION is the transformation of characters from one system to another. Conversion can be in the same character set from one encoding to another, e.g. UTF-8 to UTF-9 for Unicode, or between different character sets and encodings like ISO 2022-JP to Big5. Conversion can be ONE-WAY where some information in the original system is lost in the result, or ROUND-TRIP where the conversion is bijective and so can be reversed without loss of information. ⟦Fixme: ref⟧ Unicode is designed with round-trip conversion in mind, including some duplicate or superfluous characters solely to facilitate round-trip conversion between Unicode and other character sets. ⟦Fixme: ref⟧

One-way conversion may be INCOMPLETE, where most but not all of the text is converted successfully. This leaves the output in a potentially inconsistent state, with some parts (usually most) converted to the target character set or encoding, and some remainder not converted. Incomplete conversion is essentially always undesirable. There are terms for particularly common results of incomplete conversion. A DROP-OUT is a character which is present in the input but entirely absent in the output of the conversion. The example in (5a) shows how a character ⟨∗⟩ in the input is absent in the output. Contrast this with the conversion in (5b) that loses the ⟨∗⟩ but replaces this with whitespace ⟨ ⟩. The output in (5b) retains more information than the output in (5a), namely that there is a character in the input in the same position as ⟨ ⟩. The more rare DROP-IN is a character that does not exist in the input

but erroneously appears in the output, as with the ⟨¡⟩ in (5c).

(5) a. ⟨drop∗out⟩ → ⟨dropout⟩ *drop-out*
b. ⟨drop∗out⟩ → ⟨drop out⟩ *lossy conversion*
c. ⟨drop∗in⟩ → ⟨drop¡∗in⟩ *drop-in*

Another result of incomplete conversion is variously known as a GLITCH, TURD, WART, BARF, or several other uncomplementary names. Encoding conversion glitches arise when encoding elements of the input that are normally invisible wrongly survive conversion and appear visible in the output. The ISO 2022-JP encoding for example users invisible sequences of characters to switch between Latin and Japanese scripts. The input in (6) has the invisible sequence 1B 36 42 immediately preceding the ⟨誤⟩ character encoded as 386D and the invisible sequence 1B 28 42 immediately following. These sequences are visually represented as ⟨ESC$B⟩ and ⟨ESC(B⟩ respectively, and are known as ESCAPE SEQUENCES because they contain the control character 1B called 'escape'. An improper conversion from ISO 2022-JP could leave one of these escape sequences in the output, such as ⟨ESC$B⟩ in (6). This encoding conversion glitch is distinct from a drop-in because the data feeding the glitch is actually present in the input even though it is invisible to the user. Drop-ins in contrast are not present in the input data and are erroneously added by the conversion process.

(6) ⟨error 誤 mistake⟩ → ⟨error ESC$B誤 mistake⟩ *encoding conversion glitch*

Some conversion glitches are common enough to have colloquial names. The DREAD QUESTION MARK DISEASE ⟦FIXME: ref HDict⟧ was very common in the early era of the World Wide Web. In this textual ailment, input with non-ASCII characters is transformed to output with question marks as illustrated in (7). This is usually a result of converting from Windows-1252 encoding to ISO 8859-1 encoding using a Microsoft Office product to generate HTML web pages.

(7) ⟨“Microsoft® Word™”⟩ → ⟨?Microsoft? Word??⟩ *dread question mark disease*

Totally incomprehensible output is sometimes called MOJIBAKE /ˈmoʊdʒi̩ˌbɑkeɪ/, a term borrowed from Japanese ⟨文字化け⟩ 'character mutation'. A typical source of mojibake is erroneous representation of UTF-8 encoded Unicode in Windows-1252, as illustrated in (8) below. Mojibake like ⟨Ãƒâ€šÃ‚Â£⟩ are also known simply as GARBAGE or derisively as UNICRUD when particularly associated with Unicode mishandling. Other languages have developed terms for the same concept such as Mandarin Chinese ⟨亂碼⟩ / ⟨乱码⟩ *luànmǎ* 'chaos code', Korean ⟨외계어⟩ *oigyeeo* 'alien language', Russian ⟨крακозябры⟩ *krakozjabry* 'childish scribbles', and German *Zeichensalat* 'character salad'.

(8) ⟨文字化け⟩ → ⟨æ–‡å-—åŒ–ã‘⟩ *UTF-8 → Windows-1252 mojibake*

Particular kinds of mojibake can be detected in the input binary patterns implied by the erroneous output characters; the apparent nonsense of mojibake can be seen through to reveal data in some other encoding. In (8) the character ⟨文⟩ is encoded in UTF-8 as E6 96 87 and the character ⟨字⟩ is encoded as E5 AD 97. The Windows-1252 encoding treats each byte as a separate character, so E6 96 87 is ⟨æ–‡⟩ for ⟨文⟩ and E5 AD 97 is ⟨å-—⟩ for ⟨字⟩. The next two characters ⟨化⟩ and ⟨け⟩ are E5 8C 96 and E3 81 91 in the UTF-8 encoding. Notice the predictable four-bit pattern of $E_{16} = 1110$ in the first byte of each sequence. This highest half-byte hexit E is required by UTF-8 for encoding Unicode code points above U+07FF and below U+FFFF, and so the repeated E is a sign of underlying UTF-8 encoded data in the Windows-1252 mojibake. Because of frequent exposure to sequences like ⟨æ…å…å…ã…⟩, many users come to recognize particular kinds of mojibake associated with their scripts. Recovering the data is then a matter of applying the recognized encoding.

A similar but distinct issue is when a font lacks glyphs to display certain characters. The underlying text is encoded correctly in such situations, it is only the display that is erroneous. This is typographic and not a text problem: since the underlying text is unmolested it is only the temporary display of the data that is affected. English speakers do not encounter this phenomenon very often so there is no widely used term. But Google recently commissioned a typeface family called Noto which is designed to cover all the scripts in the Unicode character set.[3] The name is derived from "no more tofu", with TOFU referring to the white square blocks like ⟨▢▢▢▢⟩ or ⟨□□□□⟩ that often appear when a font lacks glyphs to display characters. The term 'tofu' may thus gain popularity in the future.

4.3. Encodings versus file formats

A FILE FORMAT is a specification of how data is stored in a file for use by some programs. This is distinct from a character encoding though the two are sometimes confused. The PLAIN TEXT file format is one of the simplest examples, consisting of lines of text that end with line termination characters known as 'newlines'. Other file formats used for text include the 'docx' format used by recent versions of Microsoft Word and the Rich Text Format known by its abbreviation RTF. Some file formats require a specific encoding, but many file formats have few or no constraints on how text should be encoded within the file. For example one may have a Word document with text in the Windows-1252 encoding and another document with text in the UTF-8 encoding, with the difference between them being largely invisible to the ordinary user.

3. See http://www.google.com/get/noto/.

It is often possible to mix encodings in a single file, although this is almost always a mistake and should be avoided.

Plain text files are actually several different file formats, depending largely on the newlines or line termination characters used. MS-DOS and Windows use a two-byte sequence of 0D 0A to end lines; these characters are known as Carriage Return and Line Feed (CR and LF) respectively. On a teletype terminal like the ASR-33 the CR character originally caused the print head to return to the left side of the print carriage, and then the LF character caused the paper feed mechanism to move the paper up one line. Apple decided to save a character and only use CR for the newline on the original Macintosh. The developers of Unix made a similar decision but selected LF as their newline. Today all three kinds of line terminations are in use, so a plain text file may use any of the three. Both CR and LF are defined as Unicode characters inhereited from ASCII but Unicode does not specify which of these should be used. An annex to the Unicode standard defines an automatic line breaking algorithm but leaves the choice of newline character(s) open (Heninger 2015).

5. The Unicode character set

By the late 1980s there was a bewildering variety of character set standards in use around the world. As well as those already mentioned, there was ANSEL in libraries, KOI8 for Cyrillic writing systems, JIS X 0208 and JIS X 0212 for Japanese, GB 2312 for Simplified Chinese, HKSCS for Cantonese, Big5 for Traditional Chinese, KS X 1001 for Korean, etc., etc., etc. Unicode was designed to replace this alphanumeric soup with a unified standard that could support every single writing system in use.

The project to develop Unicode was officially begun in 1991 but the earliest efforts date to 1988. The initial goal was to halt the proliferation of mutually incompatible standards for East Asian languages. The Chinese characters shared between Chinese, Japanese, and Korean languages are lumped together as the Han script in Unicode jargon, and it was believed that they could be unified into a single set called Unihan. This initial effort was soon expanded to cover every writing system since the East Asian standards include parts of the Latin, Greek, and Cyrillic scripts.

Today the Unicode standard is maintained by the non-profit Unicode Consortium. Membership in the Unicode Consortium is open to the public. Most members of the Consortium are computer hardware and software companies like Apple, Google, IBM, and Microsoft. But there are governmental members like the Government of India and the University of California Berkeley. Some linguistically oriented organizations are also members like SIL International, Evertype, and Tavultesoft. The International Standards Organization adopted Unicode as the ISO/IEC 10646 standard (Allen et al. 2015: 875–887), though this actually lags behind Unicode itself.

Unicode started as a 16 bit character set with $2^{16} = 65536$ possible characters ranging from 0000_{16} to $FFFF_{16}$. The stated goal was to "encompass all the characters of the world's languages", with the bold claim that "in a properly engineered design, 16 bits per character are more than enough for this purpose". They were of course wrong because Chinese alone requires more than 2^{16} characters.

Today Unicode uses a maximal size of 32 bits with $2^{32} \approx 4$ billion, but Unicode actually specifies a complex structure overlaid on this flat 32-bit space which limits the possible characters to about 1.1 million. As of Unicode 8.0 in late 2015, the total number of graphical and formatting characters is 120,672 (Allen et al. 2015: 3). Of this total, 81,390 characters are "Han" and thus part of the Chinese script used throughout East Asia (Allen et al. 2015: 894). In addition there are 137,468 characters designated as "Private Use" meaning that they will never be otherwise specified in future versions of the standard and so can be used outside of the standard without obsolescence. I return to private use characters in section 5.3.

5.1. Code points and character names

All Unicode characters have defined code points. A code point is represented as a hexidecimal number with the prefix 'U+' and at least four hexits, e.g. U+0041 for ⟨A⟩ and U+00E9 for ⟨é⟩. Many characters also have defined names. Character names are officially written in all uppercase as e.g. LATIN CAPITAL LETTER A or in small caps such as Latin Small Letter E With Acute. Table 2 illustrates a variety of characters with their code points and names.

Character names are not arbitrarily constructed; they have a specialized syntax defined in the Unicode standard (Allen et al. 2015: 182, 846, 859). Many character names include the name of their script as the first word in the name, such as 'Latin', 'Armenian', 'Tibetan', 'Linear B', 'Old Persian', 'Egyptian', 'Mathematical', 'Mahjong', and 'Alchemical' in table 2. There are exceptions however, depending on when the character was included in Unicode, on what standards the character might be originally sourced from, and sometimes unpredictably. As such, character names should not be depended upon for indicating their script; instead the code point and its surrounding block (see sec. 5.2) are the only reliable indices for script membership.

Character names are immutable, meaning that once they are established they are never changed in future versions of Unicode (Allen et al. 2015: 182). They are also unique so that a name refers to one and only one character. A character may however have more than one name. Additional names beyond the first are known as character name aliases (Allen et al. 2015: 183) and are defined in the Unicode Character Database (Davis, Iancu, & Whistler 2015). Because character names are immutable, any mistakes in a name cannot be corrected. Instead character name aliases are used

Glyph	*Code point*	*Name*
ʔ	U+0294	Latin Letter Glottal Stop
ա	U+0531	Armenian Letter Ayb
ཀ	U+0F57	Tibetan Letter Ka
≌	U+2246	Approximately But Not Actually Equal To
㉀	U+3240	Parenthesized Ideograph Festival
ffl	U+FB04	Latin Small Ligature FFL
𐄅	U+100C5	Linear B Ideogram B225 Bathtub
𐏉	U+103C9	Old Persian Sign Auramazdaa-2
𓀀	U+13000	Egyptian Hieroglyph A001
𝔚	U+1D582	Mathematical Bold Fraktur Capital W
🀇	U+1F017	Mahjong Tile Eight of Bamboos
😒	U+1F612	Unamused Face
🚶	U+1F6B6	Pedestrian
🜩	U+1F71F	Alchemical Symbol For Regulus of Iron

Table 2: Examples of Unicode characters with code points and names

to document corrections, as well as to define alternate names used by other standards like Horizontal Tabulation for Character Tabulation, abbreviations like Tab for Character Tabulation, and occasionally to name code points that were never actually approved elsewhere ('figments' like U+0081 High Octet Preset).

The Han characters like ⟨字⟩ and ⟨語⟩ used by Chinese, Japanese, and Korean languages among others do not have standard character names. This is because they are multilingual and naming them would privilege a particular language over the others. The official term for the Han characters is the "CJK Unified Ideographs" but the term "Han ideographic characters" is also used (Allen et al. 2015: 663). The Unicode standard notes that only a few characters are actually ideographs, the rest having "developed later via composition, borrowing, and other non-ideographic principles" (Allen et al. 2015: 663). Even though the Han characters do not have names there is a Unihan Database (Jenkins, Cook, & Lunde 2015) that documents many properties of Han characters including English definitions, stroke counts, pronunciation in various languages like Cantonese and Vietnamese, and references to standard and lexicographic sources. The text of the Unicode standard also includes several sections devoted to discussing Han characters, including most of chapter 18, a history of Han unification in appendix E, and the documentation of strokes in appendix F

Code point range			*Number*	*Abbrev.*	*Name*
000000	...	00FFFF	Plane 1	BMP	Basic Multilingual Plane
010000	...	01FFFF	Plane 2	SMP	Supplementary Multilingual Plane
020000	...	02FFFF	Plane 3	SIP	Supplementary Ideographic Plane
030000	...	0DFFFF	—	—	*reserved*
0E0000	...	0EFFFF	Plane 14	SSP	Supplementary Special-Purpose Plane
0F0000	...	0FFFFF	Plane 15	—	Supplementary Private Use Area-A
1F0000	...	10FFFF	Plane 16	—	Supplementary Private Use Area-B
110000	...	FFFFFF	—	—	*reserved*

Table 3: Planes (basic divisions) defined in Unicode 8.0

(Allen et al. 2015).

5.2. Structure: Planes and blocks

Unicode is not a flat space of characters one after the next, but rather is subdivided into a series of hierarchical units. The most basic division is a PLANE which is a very large collection of continuous code points (Allen et al. 2015: 44). The most commonly used plane is the Basic Multilingual Plane (BMP) which runs from 0000_{16} to $FFFF_{16}$ and descends from the original 16-bit design of the early 1990s. Table 3 lists all of the planes defined in Unicode 8.0.

Each plane may be divided into smaller BLOCKS which are small, continuous sequences of code points. Blocks in the BMP are small and numerous, and are mostly devoted to specific scripts or subsets of scripts. The Latin script has the largest number of blocks assigned, with around 14 in the BMP (Allen et al. 2015: 45). The largest block assigned so far is the CJK Unified Ideographs Extension B in the SIP, with 42,719 code points. The list in (9) gives a few BMP blocks that are interesting for linguists.

(9)

Range	*Name*	*Range*	*Name*
0000 ... 007F	Basic Latin	1D00 ... 1D7F	Phonetic Extensions
0080 ... 00FF	Latin-1 Supplement	1E00 ... 1EFF	Latin Extended Additional
0100 ... 017F	Latin Extended-A	2190 ... 21FF	Arrows
0180 ... 024F	Latin Extended-B	2700 ... 27BF	Dingbats
0250 ... 02AF	IPA Extensions	2C60 ... 2C7F	Latin Extended-C

Within individual blocks there may be various organizational principles but they are specific to each block and are not generalizable. Characters are often organized according to a script's traditional sorting order, but this is only the case when a new

block is defined and later additions to the block often violate ordering expectations. In some cases the ordering is graphical, defined by some arbitrary measure of visual complexity.

Not all code points in a block are necessarily assigned. Usually a block has a clump of unassigned characters at its end which are reserved for future additions. If there are enough additions made at once in a single revision then the existing reserved code points in a block are not used and instead a new block is defined.

5.3. Character types

The prototypical character is a letter like ⟨A⟩ but there are many different types of characters defined by Unicode. The GRAPHIC CHARACTERS are what most people think of, including not only letters and numbers but also punctuation, weather symbols, emoji, mathematical symbols, box-drawing elements, and many other things. Essentially a graphic character is one that will normally be seen by a human when text is displayed.

The FORMAT CHARACTERS represent mostly invisible formatting commands in plain text. Format characters are meant to be interpreted by display software presenting text to the reader. As well as newlines, other examples include right-to-left and left-to-right print order indicators, invisible hyphens marking where words can be broken, and invisible marks preventing ligature of glyphs. Format characters do not however signal things like 'italic' or 'underlined'; these features are typographic and so are properties of text display rather than the text itself. In Unicode jargon the typographic display of text is called PRESENTATION and is explicitly excluded from Unicode's standardization.

The CONTROL CODE CHARACTERS are a small set of code points inherited mostly from ASCII. These include the previously mentioned U+000D CARRIAGE RETURN and U+000A LINE FEED as welll as others like U+0008 BACKSPACE, U+0009 CHARACTER TABULATION, and U+001B ESCAPE. Control code characters were originally meant to control teletype machines but have come to be used for other purposes. There are graphic characters corresponding to the control code characters in the range U+2400 to U+243F like ⟨ESC⟩ U+241B SYMBOL FOR ESCAPE for U+001B ESCAPE and ⟨BS⟩ U+2408 SYMBOL FOR BACKSPACE for U+0008 BACKSPACE; note the parallel low bytes.

The PRIVATE USE CHARACTERS are code points that have no assigned character and which will never be assigned in the future. The Unicode standard requires that programs process these just like other graphic characters but does not specify how they should appear nor how they should act in any context. Private use characters are meant for representing characters that are not standardized. There are several conflicting specifications for subsets of the private use characters for things like

J.R.R. Tolkien's fictional scripts, medieval manuscript symbols, and corporate logos. Linguists should generally apply to have new characters officially specified in Unicode rather than rely on private use characters, but they are a useful stop-gap during the standardization process.

The SURROGATE CHARACTERS are code points that are designated for use with the 16-bit UTF-16 encoding of the Unicode character set. They have no function alone and instead must occur in what are called SURROGATE PAIRS. I discuss them in more detail in section 6 in the context of Unicode's encoding systems.

The RESERVED CHARACTERS are code points that have no assigned character but which may be assigned in a future version of Unicode. Many reserved characters appear at the end of blocks, leaving space for future characters to be assigned within an existing block rather than in a new block elsewhere. This practice helps improve the locality of related characters in the Unicode character space. Reserved characters should never be used until they are officially assigned, though some see anticipatory use just before a new version of Unicode in which they become official.

The NON-CHARACTERS are code points that are officially specified to not be valid characters in Unicode. The standard guarantees that all future versions will never assign non-character code points, in contrast with reserved characters. In addition, Unicode specifies that non-characters are not acceptable as valid characters, in contrast with the private use characters. Non-characters are meant for solving technological problems in processing digital text. The canonical example of a non-character is the code point U+FFFE which is a BYTE ORDER MARK. This value is used for detecting the order in which a computer processes bytes. A BIG-ENDIAN system processes bytes from highest to lowest ⟦FIXME: ref⟧; one example is the PowerPC architecture. A LITTLE-ENDIAN system processes bytes from lowest to highest ⟦FIXME: ref⟧; an example is the Intel x86 architecture. If a program expects to find $FFFE_{16}$ in a file but instead finds $FEFF_{16}$ then it knows the current system has the opposite byte order from the system on which the file was created. Most other non-characters in Unicode have similar technical purposes.

5.4. CHARACTER APPEARANCE

The appearance of a character is a glyph. Unicode does not actually specify glyphs, but it does supply example forms in the code charts.[4] The description of the code charts in the standard is explicit about how the glyphs shown are not official:

> Each character in these code charts is shown with a representative glyph. A representative glyph is not a prescriptive form of the character, but rather one

4. See http://www.unicode.org/charts/.

> that enables recognition of the intended character to a knowledgeable user and facilitates lookup of the character in the code charts. In many cases, there are more or less well-established alternative glyphic representations for the same character. (Allen et al. 2015: 846)

This means that Unicode characters represent graphemes and not graphs. But there are a few exceptions that can be misleading. In particular, mathematical characters come in a variety of forms reflecting typographic variation. This is because the different forms of the same letter have different meanings in mathematical usage. Thus ⟨*k*⟩ ≠ ⟨k⟩ ≠ ⟨𝓀⟩ ≠ ⟨𝕜⟩ ≠ ⟨𝕜⟩ even though these are all visibly just variants of ⟨k⟩. For similar reasons, Unicode distinguishes ⟨a⟩ from ⟨ɑ⟩ because they have different meanings in the IPA. In contrast, ⟨é⟩ is not distinguished from ⟨*é*⟩ or ⟨é⟩ in Unicode because there are no well-established specialized uses of these different glyphs.

Unicode defines separate characters for some identical graphemes because they have distinct semantics. One example is the difference between ⟨!⟩ U+0021 Exclamation Mark and ⟨!⟩ U+01C3 Latin Letter Retroflex Click, where the former is punctuation and the latter is a letter in the IPA. Another example is ⟨’⟩ U+2019 Right Single Quotation Mark versus ⟨ʼ⟩ U+02BC Modifier Letter Apostrophe, where the difference is again between punctuation and letter. The two characters ⟨µ⟩ U+00B5 Micro Sign and ⟨μ⟩ U+03BC Greek Small Letter Mu represent a technical symbol and a letter, respectively.

Some fonts graphically distinguish what are normally identical glyphs for different characters. Consider SIL International’s Gentium Plus font[5] with the sequence ⟨’ʼ⟩ which is U+2019 Right Single Quotation Mark followed by U+02BC Modifier Letter Apostrophe. The punctuation character is slightly lower and vertically longer than the modifier letter, making the two more easily identified at the end of quoted text like Tlingit ⟨‘yá éil’ʼ⟩ /já ʔéːɬʼ/ ‘this salt’.

5.5. Character properties

Unicode specifies character properties to further capture the semantic differences between characters. Properly supporting the semantics of characters is mandatory (Allen et al. 2015: 80, 159), so that an implementation cannot simply ignore this aspect of the specification. Character properties comprise facts about each character which can be “normative, informative, contributory, or provisional” (Allen et al. 2015: 159). The code charts list many properties in passing, but the definitive source of all character properties is the Unicode Character Database (Davis, Iancu, & Whistler

5. See http://software.sil.org/gentium/.

2015). The list below is repeated from the Unicode standard (Allen et al. 2015: 161), outlining the various character properties specified in the UCD.

- Name
- General Category (basic partition into letters, numbers, symbols, punctuation, etc.)
- Other important general characteristics (whitespace, dash, ideographic, alphabetic, noncharacter, deprecated, etc.)
- Display-related properties (bidirectional class, shaping, mirroring, width, etc.)
- Casing (upper, lower, title, folding – both simple and full)
- Numeric values and types
- Script and block
- Normalization properties (decompositions, decomposition type, canonical combining class, composition exclusions, etc.)
- Age (version of the standard in which the code point was first designated)
- Boundaries (grapheme cluster, word, line, and sentence)

General categories define the primary usage of a character. Each code point is assigned a normative general category (Allen et al. 2015: 174), specified as a two-letter value where the first letter indicates major class and the second letter a subclass. The 'other' subclass is the elsewhere case. Examples of general categories include 'Lu' for 'Letter, uppercase', 'Lm' for 'Letter, modifier', 'Nd' for 'Number, decimal digit', 'Sc' for 'Symbol, currency', and 'Pd' for 'Punctuation, dash'. The general category of a character defines not only its membership but how it will behave in various contexts. For example, the standard line breaking algorithm specifies whether a line break can occur after a character partly on the basis of a character's general category (Heninger 2015), and the standard text segmentation algorithm uses general categories to determine segmentation units (Davis & Iancu 2015).

5.5.1. Case

Case is specified as part of a letter's general category with one of the subclasses of uppercase (Lu), lowercase (Ll), or titlecase (Lt). The term CASE referred originally to a box of type in the days of metal printing. Types were stored in wooden boxes or drawers called 'typecases'. A full set of a single font came in two cases, one for the capital or majuscule letters and the other with the small or miniscule letters. During use the case with capitals was put in a rack above the case with miniscules, and hence the terms UPPERCASE and LOWERCASE that persist today.

CASE PAIRS define mappings between uppercase and lowercase such as ⟨A⟩ ↔ ⟨a⟩ and ⟨É⟩ ↔ ⟨é⟩ but not e.g. ⟨Á⟩ ↔ ⟨a⟩. Not all scripts have case as a feature; examples of case-distinguishing scripts include Latin, Greek, Cyrillic, Coptic, Glagolitic, and Armenian. Japanese kana, Devanagari, Hangul, Ethiopic, and Tifinagh for example are

caseless. Unicode 8.0 added a lowercase set of characters for the Cherokee syllabary which was previously a monocase or 'unicameral' system (Allen et al. 2015: 726).

Case pairs are usually defined in the same block but there are occasionally splits (Allen et al. 2015: 297), particularly with IPA letters that have become part of an orthography such as ⟨Ɣ⟩ U+0194 Latin Capital Letter Gamma in the Latin Extended-B block and ⟨ɣ⟩ U+0263 Latin Small Letter Gamma in the IPA Extensions block. Some letters are caseless but have cased alternatives. The canonical example is the caseless ⟨ʔ⟩ U+0294 Latin Letter Glottal Stop versus the case pair of ⟨Ɂ⟩ U+0241 Latin Capital letter Glottal Stop and ⟨ɂ⟩ U+0242 Latin Small Letter Glottal Stop. Automatic lowercasing of ⟨ʔ⟩ U+0294 does not change this character whereas ⟨Ɂ⟩ U+0241 ...Capital... will always become ⟨ɂ⟩ U+0242 ...Small... when lowercased. Unicode case can be analyzed as two binary features [±lower] and [±upper], although [+lower, +upper] is impossible. Thus ⟨h⟩ is [+lower, –upper], ⟨H⟩ is [–lower, +upper], and ⟨ʔ⟩ is [–lower, –upper].

Linguists often make implicit, unconsidered assumptions about case when designing orthographies with dual-case (bicameral) scripts like Latin and Cyrillic. As explained above, case pairs like ⟨H⟩ ↔ ⟨h⟩ are specified in the Unicode standard as character properties. Suppose an orthography has a letter like ⟨Ḧ⟩ which looks ugly when lowercased as ⟨ḧ⟩. An enterprising linguist might recommend using lowercase ⟨ẖ⟩ instead of ⟨ḧ⟩, but this violates the Unicode standard. Since the case pair ⟨Ḧ⟩ ↔ ⟨ẖ⟩ is nonstandard, users would have to constanly struggle against standard-conforming software. An alternative solution might be a font that displays ⟨ẖ⟩ when ⟨ḧ⟩ is in the actual data, but then users are tied to the use of particular fonts; this is another artificial restriction that users will find annoying. The reasonable solutions are either to persist with ⟨ḧ⟩ despite its aesthetic detraction, or to find some other graphic representation that is supported in Unicode like ⟨Ħ⟩ ↔ ⟨ħ⟩ or ⟨Ḥ⟩ ↔ ⟨ḥ⟩.

There are many more details to case, including operations on case, case folding, exceptional pairs, and script-specific issues. See the Unicode standard's index entry for case (Allen et al. 2015: 963) for many references.

5.6. Combining characters and normalization

Normalization is the technical process of avoiding unwanted differences in text. Unicode allows many complex characters to be constructed from simpler characters. The sequences of simpler characters and their complex equivalents are known as equivalent sequences (Allen et al. 2015: 62). Canonically equivalent sequences should display and be interpreted in exactly the same way on all systems. Compatibly equivalent sequences may display and be interpreted differently, and as such are more like suggestions than requirements. The examples of canonically (≡) and com-

patibly (≈) equivalent sequences in (exx:equivalent-sequences) are repeated from the standard. ⟦FIXME: Make these tables for less horizontal space⟧

(10) a. ⟨B⟩ U+0042 + ⟨Ä⟩ U+00C4 ≡ ⟨B⟩ U+0042 + ⟨A⟩ U+0041 + ⟨◌̈⟩ U+0308

b. ⟨LJ⟩ U+01C7 + ⟨A⟩ U+0041 ≈ ⟨L⟩ U+004C + ⟨J⟩ U+004A + ⟨A⟩ U+0041

c. ⟨2⟩ U+0032 + ⟨¼⟩ U+00BC ≈ ⟨2⟩ U+0032 + ⟨1⟩ U+0031 + ⟨⁄⟩ U+2044 + ⟨4⟩ U+0034

The examples of equivalent sequences in (10) illustrate COMBINING CHARACTERS: graphic characters like ⟨◌̈⟩ U+0308 COMBINING DIAERESIS that are intended to appear together with a BASE CHARACTER. Combining characters are mostly diacritics, but also include some things like surrounding circles and slashes for symbols. The Unicode convention is to represent these with a dotted circle hosting the diacritic, so that it is clear where they are positioned on the base character: ⟨◌́⟩, ⟨◌̊⟩, ⟨◌̲⟩, ⟨◌̣⟩, ⟨◌̸⟩, ⟨◌'⟩, ⟨◌̷⟩, ⟨◌̵⟩, ⟨◌̤⟩, ⟨◌ʰ⟩. Unicode's combining characters always follow their base characters in sequence as shown in (11); this is unlike dead-keys in some input methods which precede their base.

(11) ⟨e⟩ U+0065 + ⟨◌́⟩ U+0301 + ⟨a⟩ U+0061 → ⟨éa⟩ never *⟨eá⟩

There is no standard limit on the number of combining characters that can be used with a single base, but in practice more than about three will be illegible or fail to display properly. Even if combining characters display improperly or are unreadable, the text containing them is still functional.

Unicode specifies many PRECOMPOSED CHARACTERS such as ⟨é⟩ U+0039 LATIN SMALL LETTER E WITH ACUTE. These are canonically decomposable into a sequence such as ⟨e⟩ U+0065 LATIN SMALL LETTER E and ⟨◌́⟩ U+0301 COMBINING ACUTE ACCENT. The decomposition of precomposed characters is standardized. But why have precomposed characters at all? Unicode specifies them almost exclusively because they are required for round-trip conversion with other character sets. The usual example of this is VISCII which specifies a large number of precomposed vowels for Vietnamese.

Some apparently precomposed characters are specfied because they are difficult to compose automatically for display purposes, such as ⟨ɬ⟩ U+2C61 LATIN SMALL LETTER L WITH DOUBLE BAR and ⟨ⱦ⟩ U+2C66 LATIN SMALL LETTER T WITH DIAGONAL STROKE. These are technically not precomposed characters however because they do not have a decomposition specified for them. The lack of a canonical decomposition is specifically to prevent implementors from being tempted to represent them with combining diacritics.

In general the Unicode Consortium prefers to handle all new Latin letters with combining characters whenever possible. Avoiding the addition of new precomposed characters has several technological advantages, such as reducing the amount of hidden complexity (precomposed versus decomposed), simplifying the addition of new characters (one combining character instead of several precomposed characters), and conserving code points for the future. Precomposed characters are essentially a historical artifact, existing only to ensure that round-trip conversion with older character sets is possible.

Both precomposed characters and combining characters can be mixed in the same text, though this is usually undesirable. The mixture of precomposed and combining characters is DENORMALIZED in Unicode terms, and fixing this is NORMALIZATION. Unicode specifies two NORMALIZATION FORMS for characters: (i) Normalization Form Canonical Decomposition (NFCD) and (ii) Normalization Form Canonical Composition (NFCC). ⟦FIXME: ref⟧ NFCD has all precomposed characters decomposed to their base + combining character equivalent sequences. NFCC has all sequences of base + combining character converted to their precomposed character equivalents where possible. NFCC is notably preferred by Mac OS X ⟦FIXME: ref⟧ and NFCD is usually used on Windows and Linux ⟦FIXME: refs⟧. Some programs prefer one or the other and may misbehave, but most software does not care. Operating systems usually normalize text automatically, but denormalized text can still occur and should be considered when encountering problems with digital text.

Only potentially precomposed characters are affected by normalization. For example, the glyph ⟨v́⟩ does not have a precomposed character assigned to it so it will always be represented by the sequence of ⟨v⟩ U+0076 LATIN SMALL LETTER V + ⟨◌́⟩ U+0301 COMBINING ACUTE ACCENT. This is regardless of whether the surrounding text is in NFCD or NFCC.

Normalization does not generally modify the order of multiple combining characters; Unicode specifies that combining characters should work in any order. ⟦FIXME: ref⟧ Thus there is no standard difference between (12a) and (12b) below.

(12) a. ⟨a⟩ U+0061 LATIN SMALL LETTER A *acute + ogonek*
+ ⟨◌́⟩ U+0301 COMBINING ACUTE ACCENT
+ ⟨◌̨⟩ U+0328 COMBINING OGONEK
→ ⟨ą́⟩

b. ⟨a⟩ U+0061 LATIN SMALL LETTER A *ogonek + acute*
+ ⟨◌̨⟩ U+0328 COMBINING OGONEK
+ ⟨◌́⟩ U+0301 COMBINING ACUTE ACCENT
→ ⟨ą́⟩

Nonetheless, some fonts and/or display systems may output forms that are different depending on the order of combining characters. This is technically a bug and should be reported to the relevant font designers or software developers. Switching combining character order to whichever displays better is a stopgap measure and should not be maintained over the long term.

5.7. Sorting and collation

The term COLLATION is the official Unicode label for what is generally called SORTING, "the process of ordering units of textual information" (Unicode Consortium 2015). Any user of a Latin script–based writing system is familiar with the alphabetic order of ⟨a, b, c, d, ...⟩. But not all Latin writing systems are ordered in the same way. Swedish for example ends its alphabet with ⟨..., x, y, z, å, ä, ö⟩, but Danish and Norwegian have ⟨..., x, y, z, æ, ø, å⟩. German dictionaries generally ignore the differences between ⟨a⟩ and ⟨ä⟩but German telephone directories traditionally treat ⟨ä⟩ = ⟨ae⟩ which is ordered differently from just ⟨a⟩.

The ordering of Unicode code points is a crude collation implicit in the structure of Unicode. Because ⟨A⟩ U+0041 precedes ⟨B⟩ U+0042, numerically sorting code points gives the expected alphabetic ordering for English. A side effect is that uppercase letters precede lowercase letters since ⟨a⟩ U+0061 > ⟨Z⟩ U+005A. This ordering goes back to ASCII and is why so many computer programs sort uppercase before lowercase.

Unicode code points are not a sufficient ordering for the whole Latin script. The letter ⟨Á⟩ U+00C1 precedes ⟨É⟩ U+00C9 as expected, but both of these code points are much higher than ⟨z⟩ U+007A. If only code points were used to sort letters, we would have a very strange alphabetic ordering with things like ⟨Ë⟩ U+00CB before ⟨Đ⟩ U+00D0 before ⟨Ā⟩ U+0100. To avoid this, Unicode specifies a standard algorithm for sorting text, the Unicode Collation Algorithm (Davis, Whistler, & Scherer 2015).

The UCA is a multi-level algorithm that successively compares base characters, diacritics, case, punctuation, and finally code point ("identical") (Davis, Whistler, & Scherer 2015: §1.1). The UCA is synchronized with the ISO/IEC 14651 standard for sorting, so one can follow either for the same effect. Language-specific sorting is implemented using lookup tables called Collation Element Tables (Davis, Whistler, & Scherer 2015: §3). Many such tables are specified in the Common Locale Data Repository (CLDR),[6] which also includes other language- and area-specific information like date and currency formats, timezones, names of months and days, spelling of numbers, and much more. The CLDR is open for public contribution so new Collation

6. See http://cldr.unicode.org/.

Element Tables and other locale data can be submitted for review and inclusion.

6. Unicode encodings

An ENCODING is a mapping from code points to bytes used to actually represent characters. The simplest encoding is to use the code points themselves, and for Unicode this is known as the Unicode Transformation Format 32, abbreviated UTF-32. The '32' refers to the 32 bits required per character. UTF-32 is extremely inefficient: most characters in Unicode have only 8 or 10 distinct bits, so the 22 or so remaining are wasted space. Consider ⟨é⟩ U+00E9 Latin Small Letter E With Acute; the full code point is actually U+000000E9 but only the lowest byte E9 is informative. For any Latin script the majority of text only needs one byte per character, meaning that 75% of the space required for UTF-32 is wasted. UTF-32 is used internally for text processing in some Unix-based operating systems because it is algorithmically very simple, but UTF-32 is rarely used for storing text because it is so spatially inefficient.

The UTF-16 encoding is more efficient than UTF-32 because, as its name implies, it only requires 16 bits per character. All characters below U+FFFF are represented directly, covering the entire Basic Multilingual Plane (BMP) and hence the vast majority of languages today. The character ⟨é⟩ U+00E9 is represented as the two bytes 00 and E9, identical to the code point. Above U+FFFF the UTF-16 encoding uses SURROGATE PAIRS, pairs of 16 bit units that uniquely identify characters. The higher value or 'leading' surrogate is a 16 bit unit in in the range D800 ... DBFF and the lower value or 'trailing' surrogate is in the range DC00 ... DFFF. Examples of some UTF-16 surrogate pair sequences are given in table 4, contrasting with directly represented code points.

The Unicode encoding most widely used in practice is UTF-8. This is an 8-bit encoding that is optimized for the most common kinds of digital text. Unlike UTF-16, only one byte is necessary for Basic Latin making the representation identical to ASCII. Many other characters are encoded in two bytes, and the system gradually adds three or four bytes as necessary. The CHARACTER LENGTH or number of bits per character is thus variable between different blocks as indicated in (13).

(13)

Range	*Bytes per character*
U+0000 ... U+007F	1 byte per character
U+0080 ... U+07FF	2 bytes per character
U+0800 ... U+FFFF	3 bytes per character
≥ U+10000	4 bytes per character

A subtle but essential feature of UTF-8 is that no characters above U+007F are ever represented using characters below that value. This ensures that any conversion

Glyph	*Code point*	*Name*	*Representation*			
a	U+0061	Latin Small Letter A	00	61		
ʔ	U+0294	Latin Letter Glottal Stop	02	94		
अ	U+0905	Devanagari Letter A	09	05		
幺	U+2E93	CJK Radical Thread	2E	93		
뫘	U+BB10	Hangul Syllable Moess	BB	10		
	U+FFFD	Replacement Character	FF	FD		
	U+10000	Linear B Syllable B008 A	D8	00	DC	00
	U+103A0	Old Persian Sign A	D8	00	DF	A0
	U+10900	Phoenician Letter Alf	D8	02	DD	00
	U+16800	Bamum Letter Phase-A Ngkue Mfon	D8	1A	DC	00
	U+1F0AE	Playing Card King of Spades	D8	3C	DC	AE
	U+1F442	Ear	D8	3D	DC	42

Table 4: Examples of UTF-16 showing surrogate pairs above U+FFFF

will always preserve ASCII characters even if other characters are mangled. Because most text markup languages like XML and TEX use ASCII for their markup commands, UTF-8 ensures that at least the markup of some text is preserved with incorrect conversion even if other data is not. This makes the result of incorrect conversion displayable, thus allowing humans to see the incorrect result rather than a potentially enigmatic error message.

The most frequently used code points in Unicode require the least number of bytes in UTF-8. This means that UTF-8 is much more space-efficient than ohter Unicode encodings. Some example encodings of characters in () demonstrate how higher code points require more bytes.

(14)

a	U+0061	below U+007F so 1 byte	→ 61
α	U+03B1	below U+07FF so 2 bytes	→ CE B1
ѣ	U+0463	below U+07FF so 2 bytes	→ D1 A3
अ	U+0905	below U+FFFF so 3 bytes	→ E0 A4 85
あ	U+3042	below U+FFFF so 3 bytes	→ E1 81 82
	U+103C9	above U+FFFF so 4 bytes	→ F0 90 8F 89

The algorithm for encoding code points in UTF-8 can be hard to understand for non-specialists. But UTF-8 is so well supported today that users generally need not ever think about how this encoding works. UTF-8 has become the preferred encoding

by far for Unicode text, and Unicode is both the technical and practical preference for all digital text today, so UTF-8 is dominant.

There are a few other unusual encodings for Unicode that are occasionally encountered in the wild. Punycode is an encoding of Unicode that outputs only graphic ASCII characters. ⟦Fixme: ref⟧ Although it is extremely inefficient, Punycode can be processed by anything that can process ASCII. Punycode is particularly used for internationalized domain names in the Domain Name System. ⟦Fixme: ref⟧ There are also old encodings known as UCS-2 (≈ UTF-16) and UCS-4 (≈ UTF-32) which are rare and largely deprecated today in favour of the UTF-series. Finally, the UTF-9 and UTF-18 encodings were designed and implemented as a joke by Mark Crispin for long obsolete computers like the 36-bit PDP-10s and UNIVAC 1100 series (Crispin 2005).

7. Some practical issues for linguists

The Unicode standard (Allen et al. 2015) is freely available and is remarkably readable for a formal standard document. It is however quite large, at nearly 1000 pages for Unicode 8.0, despite excluding the code charts that are separately available online. The editors have taken pains to define nearly all specialized terminology, and a separate glossary covers even more terminological detail (Unicode Consortium 2015). Linguists with even a passing interest in character set and encoding issues should consider reviewing at least the introduction and the second chapter on general structure which lay out most of the basic issues in the representation of digital text, as well as explaining the technical decisions behind Unicode's structure. The chapters on particular scripts and geographic areas provide valuable descriptions of both common and obscure writing systems, as well as some historical and linguistic background. Finally, the code charts[7] and Unicode character database (Jenkins, Cook, & Lunde 2015) include detailed information about the function and purpose of most characters and should be referred to by anyone considering the use of a particular character.

The most important thing to keep in mind when using Unicode is to choose the right character for the job. The Unicode standard specifies many different characters that "look the same" but which have distinct semantics. Even though ⟨A⟩ U+0041 Latin Capital Letter A, ⟨A⟩ U+0391 Greek Capital Letter Alpha, and ⟨A⟩ U+0410 Cyrillic Capital Letter A have identical glyphs in many fonts, this does not mean that they are identical characters. Mixing them can cause subtle problems because Unicode specifies them as having different behaviour and different contexts of use.

7. See http://www.unicode.org/charts/.

The Unicode standard often provides guidelines on how to use particular characters. The following quote is taken from the text introducing the Phonetic Extensions Block from U+1D00 to U+1D7F.[8]

> The small capitals, superscript, and subscript forms are for phonetic representations where style variations are semantically important. For general use, use regular Latin, Greek, or Cyrillic letters with markup instead.

This means that even though ⟨ᴀ⟩ U+1D00 Latin Letter Small Capital A looks like it could be used in small caps for formatting a section title or an author name, it is instead intended explicitly as a phonetic symbol (e.g. the predecessor of today's IPA /ʌ/) and should not be used for regular text. Small caps in gloss abbreviations and so forth should be done using typographic or word processor mechanisms, not using the "small capital" Unicode characters.

Probably the worst offence of many linguists is using the wrong character for the null symbol. The character Unicode intends for this purpose is ⟨∅⟩ U+2205 Empty Set. This character has the following comments on its entry in the code chart for the Mathematical Operators block from U+2200 to U+22FF.[9]

> 2205 ∅ EMPTY SET
> = null set
> · used in linguistics to indicate a null morpheme or phonological "zero"
> → 00D8 Ø latin capital letter o with stroke
> → 2300 ⌀ diameter sign

The '=' comment designates a character name alias, here that 'empty set' = 'null set'. The '→' comments are contrastive comparisons, indicating characters that have a similar appearance but different semantics. The comparison with ⟨Ø⟩ U+00D8 implies that it is not equivalent to ⟨∅⟩ even though they are superficially related. The letter ⟨ø⟩ U+00F8 is the lowercase version of ⟨Ø⟩ and likewise is not meant for use as a null symbol. Turning to ⟨ø⟩ U+00F8 in the Latin-1 Supplement block[10] from U+0080 to U+00FF there is a short comment "· Danish, Norwegian, Faroese, IPA". This means that ⟨ø⟩ U+00F8 is meant for use as a letter in phonetic transcriptions as well as in the languages mentioned, but there is no mention of its use as a null symbol.

Linguists may think that because the character looks appropriate it should be acceptable. This is technically incorrect, and text using the wrong character is essentially lying about its content. Other people may search for the null symbol using ⟨∅⟩

8. See http://unicode.org/charts/PDF/U1D00.pdf.
9. See http://unicode.org/charts/PDF/U2200.pdf.
10. See http://unicode.org/charts/PDF/U0080.pdf.

U+2205, but these searches will fail if ⟨Ø⟩ U+00D8 is used instead. Automatic case conversion will switch between ⟨ø⟩ and ⟨Ø⟩ whereas ⟨∅⟩ U+2205 will be unmodified. Aesthetic issues with a particular font can be solved by switching to other fonts with a better appearance, so for example although Gentium Plus has a possibly 'ugly' ⟨∅⟩, Cambria has ⟨∅⟩, Minion Pro has ⟨∅⟩, and STIX Variants has ⟨∅⟩, each of which is more like the 'slashed zero' used by linguists.

Linguists designing orthographies should generally stay within the confines of already defined Unicode characters because the process for defining new characters is long and involved. Proposals for characters that could be represented by base + combining character sequences will usually be rejected in favour of using the corresponding sequence, with the exception being those graphemes that directly modify the outline of a base character (overlapping diacritics like slash and double-bar). Orthography designers should pay careful attention to case and other character properties, and should try to avoid graphically confusable characters like ⟨'⟩ U+02BC Modifier Letter Apostrophe and ⟨'⟩ U+2019 Right Single Quotation Mark. Orthographies based on existing scripts should try to follow existing sorting and collation practices rather than burden users with novel collations that are poorly supported in software. This especially means that Latin and Cyrillic orthographies should prefer alphabetic ordering rather than articulatory ordering since the latter is entirely unsupported by Unicode.

There is an organization devoted to preparing Unicode proposals for new characters and scripts, the Script Encoding Initiative (SEI) of the UC Berkeley Department of Linguistics.[11] Linguists confronted with existing orthographies or characters not defined in Unicode should contact the SEI. They have successfully worked to include over 70 scripts in Unicode, including additions to the Latin and Cyrillic scripts.

Unicode is a complex system meant to solve complex problems. It is the only modern solution for representing digital text around the world. Most of the problems that Unicode confronts are not obvious to the uninitiated, often involving both technical and social compromises. Since linguists work with digital text every day, awareness of Unicode and its basic principles is a professional responsibility. Linguists involved with orthography design have a particularly strong need to understand Unicode since users will inevitably encounter it in use with all modern technology.

References

Allen, Julie D. et al. (eds.). 2015. *The Unicode standard, version 8.0: Core*

11. See http://www.linguistics.berkeley.edu/sei/index.html.

specification. Mountain View, CA: Unicode Consortium. ISBN 978-1-936213-10-8. URL http://www.unicode.org/versions/Unicode8.0.0/.

Crispin, Mark. 2005. UTF-9 and UTF-18 efficient transformation formats of Unicode. RFC 4042. Fremont, CA: Internet Engineering Task Force, Internet Society. URL http://tools.ietf.org/html/rfc4042.

Davis, Mark & Laurenţiu Iancu. 2015. Unicode text segmentation. Unicode Standard Annex 29. Revision 27. Mountain View, CA: Unicode Consortium. URL http://www.unicode.org/reports/tr29/tr29-27.html.

Davis, Mark, Laurenţiu Iancu, & Ken Whistler. 2015. Unicode character database. Unicode Standard Annex 44. Revision 16. Mountain View, CA: Unicode Consortium. URL http://www.unicode.org/reports/tr44/tr44-16.html.

Davis, Mark, Ken Whistler, & Markus Scherer. 2015. Unicode collation algorithm. Unicode Technical Standard 10. Revision 32. Mountain View, CA: Unicode Consortium. URL http://www.unicode.org/reports/tr10/tr10-32.html.

Heninger, Andy. 2015. Unicode line breaking algorithm. Unicode Standard Annex 14. Revision 35. Mountain View, CA: Unicode Consortium. URL http://www.unicode.org/reports/tr14/tr14-35.html.

Jenkins, John H., Richard Cook, & Ken Lunde. 2015. Unicode Han database (Unihan). Unicode Standard Annex 38. Revision 19. Mountain View, CA: Unicode Consortium. URL http://www.unicode.org/reports/tr38/tr38-19.html.

Lunde, Ken. 2009. *CJKV information processing*. 2nd edn. Sebastopol, CA: O'Reilly Media. ISBN 978-0-596-51447-1.

Unicode Consortium. 2015. *Glossary of Unicode terms*. Unicode Consortium, Inc. URL http://www.unicode.org/glossary/.