The Unicode Cookbook for Linguists

Managing writing systems using orthography profiles

Steven Moran & Michael Cysouw

Change dedication in localmetadata.tex

Contents

1 Writing Systems

1.1 Introduction

Writing systems arise and develop in a complex mixture of cultural, technological and practical pressures. They tend to be highly conservative, in that people who have learned to read and write in a specific way—however impractical or tedious—are mostly unwilling to change their habits. Writers tend to resist spelling reforms. In all literate societies there exists a strong socio-political mainstream that tries to force unification of writing (for example by strongly enforcing "right" from "wrong" writing in schools). However, there is also a large community of users who take as many liberties in their writing as they can get away with.

For example, the writing of tone diacritics in Yoruba is often proclaimed to be the right way to write, although many users of Yoruba orthography seem to be perfectly fine with leaving them out. As pointed out by the proponents of the official rules, there are some homographs when leaving out the tone diacritics (Olúmúyìw 2013: 44). However, writing systems (and the languages they represent) are normally full of homophones, which is normally not a problem at all for speakers of the language. More importantly, writing is not just a purely functional tool, but just as importantly it is a mechanism to signal social affiliation. By showing that you *know the rules* of expressing yourself in writing, others will more easily accept you as a worthy participant in their group. And that just as well holds for obeying to the official rules when writing a job application, as for obeying to the informal rules when writing an SMS to classmates in school. The case of Yoruba writing is an exemplary case, as even after more than a century of efforts to standardize the writing systems, there is still a wide range of variation in daily use (Olúmúyìw 2013).

The sometimes cumbersome and sometimes illogical structure, and the enormous variability of existing writing systems is a fact of life scholars have to accept and should try to adapt to as good as possible. Our plea here is a proposal for a formalization to do exactly that.

When considering the worldwide linguistic diversity, including all lesser-studied and endangered languages, there exist numerous different orthographies using symbols from the same scripts. For example, there are hundreds of orthographies using Latin-based alphabetic scripts. All of these orthographies use the same symbols, but these symbols differ in meaning and usage throughout the various orthographies. To be able to computationally use and compare different orthographies, we need a way to specify all orthographic idiosyncrasies in a computer-readable format (a process called TAILORING in Unicode parlance). We call such specifications ORTHOGRAPHY PROFILES. Ideally, these specifications have to be integrated into so-called Unicode locale descriptions, though we will argue that in practice this is often not the most useful solution for the kind of problems arising in the daily practice of linguistics. Consequently, a central goal of this paper is to flesh out the linguistic challenges for locale descriptions, and work out suggestions to improve their structure for usage in a linguistic context. Conversely, we also aim to improve linguists' understanding and appreciation for the accomplishments of the Unicode Consortium in the development of the Unicode Standard.

The necessity to computationally use and compare different orthographies most forcefully arises in the context of language comparison. Concretely, in our current research our goal is to develop quantitative methods for language comparison and historical analysis in order to investigate worldwide linguistic variation and to model the historical and areal processes that underlie linguistic diversity, cf. Steiner, Stadler & Cysouw (2011); List (2012a,b); List & Moran (2013); Moran & Prokić (2013). In this work, it is crucial to be able to flexibly process across numerous resources with different orthographies. In many cases even different resources on the *same* language use different orthographic conventions. Another orthographic challenge that we encounter regularly in our linguistic practice is electronic resources on a particular language that claim to follow a specific orthographic convention (often a resource-specific convention), but on closer inspection such resources are almost always not consistently encoded. Thus a second goal of our orthography profiles is to allow for an easy specification of orthographic conventions, and use such profiles to check consistency and to report errors to be corrected.

A central step in our proposed solution to this problem is the tailored grapheme separation of strings of symbols, a process we call GRAPHEME TOKENIZATION. Basically, given some strings of symbols (e.g. morphemes, words, sentences) in a specific source, our first processing step is to specify how these strings have to be separated into graphemes, considering the specific orthographic conventions

used in a particular source document. Our experience is that such a graphemic tokenization can be performed without extensive in-depth knowledge about the phonetic and phonological details of the language in question. For example, the specification that <ou> is a grapheme of English is a much easier task than to specify what exactly the phonetic values of this grapheme are in any specific occurrence in English words. Grapheme separation is a task that can be performed relatively reliably and with limited availability of time and resources (compare, for example, the task of creating a complete phonetic or phonological normalization).

Although grapheme tokenization is only one part of the solution, it is an important and highly fruitful processing step. Given a grapheme tokenization, various subsequent tasks become easier, like (a) temporarily reducing the orthography in a processing pipeline, e.g. only distinguishing high versus low vowels; (b) normalizing orthographies across sources (often including temporary reduction of oppositions), e.g. specifying an (approximate) mapping to the International Phonetic Alphabet; (c) using co-occurrence statistics across different languages (or different sources in the same language) to estimate the probability of grapheme matches, e.g. with the goal to find regular sound changes between related languages or transliterations between different sources in the same language.

Before we deal with these proposals, in the first part of this paper (Sections 1.2 through 3) we give an extended introduction to the notion of encoding (Section 1.2) and writing systems, both from a linguistic perspective and from the perspective of the Unicode Consortium (Section **??**). We consider the Unicode Standard to be a breakthrough (and ongoing) development that fundamentally changed the way we look at writing systems, and we aim to provide here a slightly more in-depth survey of the many techniques that are available in the standard. A good appreciation for the solutions that the Unicode Standard also allows for a thorough understanding of possible pitfalls that one might encounter when using it (Section 2). As an example of the current state-of-the-art, we discuss the rather problematic marriage of the International Phonetic Alphabet (IPA) with the Unicode Standard (Section 3).

The second part of the paper (Sections 4 and 5) describes our proposals for how to deal with the Unicode Standard in the daily practice of (comparative) linguists. First, we discuss the challenges of characterizing a writing system. To solve these problems, we propose the notions of orthography profiles, closely related to Unicode locale descriptions (Section 4). Finally, we discuss practical issues with actual examples (Section 5). We provide reference implementation of our proposals in R and in Python, available as open-source libraries.

The following conventions are followed in this paper. All phonemic and phonetic representations are given in the International Phonetic Alphabet (IPA), unless noted otherwise (International Phonetic Association 2005). Standard conventions are used for distinguishing between graphemic < >, phonemic / / and phonetic [ ] representations. For character descriptions, we follow the notational conventions of the Unicode Standard (Unicode Consortium 2014). Character names are represented in small capital letters (e.g. LATIN SMALL LETTER SCHWA) and code points are expressed as U+*n* where *n* is a four to six digit hexadecimal number, e.g. U+0256, which can be rendered as the glyph <ɘ>.

1.2 Encoding

There are many in-depth histories of the origin and development of writing systems (e.g. Robinson 1995; Powell 2012), a story that we therefore will not repeat here. However, the history of turning writing into computer readable code is not so often told, so we decided to offer a short survey of the major developments of such encoding here.[1] This history turns out to be intimately related to the history of telegraphic communication.

Telegraphy

Writing systems have existed for roughly 6000 years, allowing people to exchange messages through time and space. Additionally, to quickly bridge large geographic distances, telegraphic systems of communication (from Greek *τῆλε γράφειν* 'distant writing') have a long and widespread history since ancient times. The most common telegraphic systems worldwide are so-called whistled languages (Meyer 2015), but also drumming languages (Meyer, Dentel & Seifart 2012) and signalling by smoke, fire, flags or even change in water levels through hydraulic pressure have been used as forms of telegraphy.

Telegraphy was reinvigorated in the end of the eighteenth century through the introduction of so-called semaphoric systems by Claude Chapelle to convey

[1] Because of the recent history as summarized in this section, we have used mostly rather ephemeral internet sources. When not references by traditional literature in the bibliography, we have used http://www.unicode.org/history/ and various Wikipedia pages for the information presented here. A useful survey of the historical development of the physical hardware of telegraphy and telecommunication is Huurdeman (2003). Most books that discuss the development of encoding of telegraphic communication focus of cryptography, e.g. Singh (1999), and forego the rather interesting story of "open", i.e. non-cryptographic, encoding that is related here.

messages over large distances. Originally, various specially designed contraptions were used to send messages. Today, descendants of these systems are still in limited use, for example utilizing flags or flashing lights. The "innovation" of those semaphoric systems was that all characters of the written language were replaced one-to-one by visual signals. Since then, all telegraphic systems have taken this principle, namely that any language to be transmitted first has to be turned into some orthographic system, which subsequently is encoded for transmission by the sender, and then turned back into orthographic representation at the receiver side.[2] This of course implies that the usefulness of any such telegraphic encoding completely depends on the sometimes rather haphazard structure of orthographic systems.

In the nineteenth century, electric telegraphy lead to a new approach in which written language characters were encoded by signals sent through a copper wire. Originally, BISIGNAL CODES were used, consisting of two different signals. For example, Carl Friedrich Gauss in 1833 used positive and negative current (Mania 2008: 282). More famous and influential, Samuel Morse in 1836 used long and short pulses. In those bisignal codes each character from the written language was encoded with a different number of signals (between one and five), so two different separators are needed: one between signals and one between characters. For example, in Morse-code there is a short pause between signals and a long pause between characters.[3]

Binary encoding

From those bisignal encodings, true BINARY CODES developed with a fixed length of signals per character. In such systems only a single separator between signals is needed, because the separation between characters can be established by counting until a fixed number of signals has passed.[4] In the context of electric telegraphy, such a binary code system was first established by Émile Baudot in

[2] Sound and video-based telecommunication of course takes a different approach by ignoring the written version of language and directly encode sound waves or light patterns.

[3] Actually, Morse-code also includes an extra long pause between words. Interestingly, it took a long time to consider the written word boundary—using white-space—as a bona-fide character that should simply be encoded with its own code point. This happened only with the revision of the Baudot-code (see below) by Donald Murray in 1901, in which he introduced a specific white-space code. This principle has been followed ever since.

[4] Of course, no explicit separator is needed at all when the timing of the signals is known, which is the principle used in all modern telecommunication systems. An important modern consideration is also how to know where to start counting when you did not catch the start of a message, something that is known in Unicode as SELF SYNCHRONIZATION.

1870, using a fixed combination of five signals for each written character.[5] There are $2^5 = 32$ possible combination when using five binary signals; an encoding today designated as 5-bit. These codes are sufficient for all Latin letters, but of course they do not suffice for all written symbols, including punctuation and digits. As a solution, the Baudot code uses a so-called "shift" character, which signifies that from that point onwards—until shifted back—a different encoding is used, allowing for yet another set of 32 codes. In effect, this means that the Baudot code, and the International Telegraph Alphabet (ITA) derived from it, had an extra bit of information, so the encoding is actually 6-bit (with $2^6 = 64$ different possible characters). For decades, this encoding was the standard for all telegraphy and it is still in limited use today.

To also allow for different uppercase and lowercase letters and for a large variety of control characters to be used in the newly developing technology of computers, the American Standards Association decided to propose a new 7-bit encoding in 1963 (with $2^7 = 128$ different possible characters), known as the American Standard Code for Information Interchange (ASCII), geared towards the encoding of English orthography. With the ascent of other orthographies in computer usage, the wish to encode further variation of Latin letters (like German <ß> or various letters with diacritics, like <è>) led the Digital Equipment Corporation to introduce an 8-bit Multinational Character Set (MCS, with $2^8 = 256$ different possible characters), first used with the introduction of the VT220 Terminal in 1983.

Because 256 characters were clearly not enough for the many different characters needed in the world's writing systems, the ISO/IEC 8859 standard in 1987 extended the MCS to include 16 different 8-bit code pages. For example, part 5 was used for Cyrillic characters, part 6 for Arabic, and part 7 for Greek.[6] This system almost immediately was understood to be insufficient and impractical, so various initiatives to extend and reorganize the encoding started in the 1980s. This led, for example, to various proprietary encodings from Microsoft (e.g. Windows

[5] True binary codes have a longer history, going at least back to the Baconian cipher devised by Francis Bacon in 1605. However, the proposal by Baudot was the quintessential proposal leading to all modern systems.

[6] In effect, because $16 = 2^4$, this means that ISO/IEC 8859 was actually an $8+4 = 12$-bit encoding, though with very many duplicates by design, namely all ASCII codes were repeated in each 8-bit code page. To be precise, ISO/IEC 8859 used the 7-bit ASCII as the basis for each code page, and defined 16 different 7-bit extensions, leading to $(1 + 16) \cdot 2^7 = 2,176$ possible characters. However, because of overlap and not-assigned codes points the actual number of symbols was much smaller.

Latin 1) and Apple (e.g. Mac OS Roman), which one still sometimes encounters today.

More wide-ranging, various people in the 1980s started to develop true international code sets. In the United States, a group of computer scientists formed the Unicode Consortium, proposing a 16-bit encoding in 1991 (with $2^{16} = 65,536$ different possible characters). At the same time in Europe, the International Organization for Standardization (ISO) was working on ISO 10646 to supplant the ISO/IEC 8859 standard. Their first draft of the Universal Character Set (UCS) in 1990 was 31-bit (with theoretically $2^{31} = 2,147,483,648$ possible characters, but because of some technical restrictions only 679,477,248 were allowed). Since 1991, the Unicode Consortium and the ISO jointly develop the unicode standard, or ISO/IEC 10646, leading to the current system including the original 16-bit Unicode proposal as the basic multilingual plane, and 16 additional planes of 16-bit for further extensions (with in total $(1+16) \cdot 2^{16} = 1,114,112$ possible characters). The most recent version of the Unicode Standard (currently at version number 7.0) was published in June 2014 and it defines 112,956 different characters (Unicode Consortium 2014).

In the next section we provide a very brief overview of the linguistic terminology concerning writing systems before turning to the slightly different computational terminology in the next section on the Unicode Standard.

1.3 Linguistic terminology

Linguistically speaking, a writing system is a symbolic system that uses visible or tactile signs to represent language in a systematic way. The term writing system has two mutually exclusive meanings. First, it may refer to the way a particular language is written. In this sense the term refers to the writing system of a particular language, as, for example, in *'the Serbian writing system uses two scripts: Latin and Cyrillic.'* Second, the term writing system may also refer to a type of symbolic system as used among the world's languages to represent the language, as, for example, in *'alphabetic writing system.'* In this latter sense the term refers to how scripts have been classified according to the way that they encode language, as in, for example, *'the Latin and Cyrillic scripts are both alphabetic writing systems.'* To avoid confusion, this second notion of writing system would more aptly have been called script system.

Writing systems

Focussing on the first sense, we distinguish two different kinds of writing systems used for a particular language, namely transcriptions and orthographies.

First, TRANSCRIPTION is a scientific procedure (and also the result of that procedure) for graphically representing the sounds of human speech at the phonetic level. It incorporates a set of unambiguous symbols to represent speech sounds, including conventions that specify how these symbols should be combined. A transcription system is a specific system of symbols and rules used for transcription of the sounds of a spoken language variety. In principle, a transcription system should be language-independent, in that it should be applicable to all spoken human languages. The INTERNATIONAL PHONETIC ALPHABET (IPA) is a commonly used transcription system that provides a medium for transcribing languages at the phonetic level. However, there is a long history of alternative kinds of transcription systems (see Kemp 2006) and today various alternatives are in widespread use (e.g. X-SAMPA and Cyrillic-based phonetic transcription systems). Many users of IPA do not follow the standard to the letter, and many dialects based on the IPA have emerged, e.g. the Africanist and Americanist transcription systems. Note that IPA symbols are also often used to represent language on a phonemic level. It is important to realize that in this usage the IPA symbols are not a transcription system, but rather an orthography (though with strong links to the pronunciation). Further, a transcription system does not need to be as highly detailed as the IPA. It can also be a system of broad sound classes. Although such an approximative transcription is not normally used in linguistics, it is widespread in technological approaches (Soundex and variants, e.g. Knuth 1973: 391–392; Postel 1969; Beider & Morse 2008), and it is sometimes fruitfully used in automatic approaches to historical linguistics (Dolgopolsky 1986; List 2012c; Brown, Holman & Wichmann 2013).

Second, an ORTHOGRAPHY specifies the symbols, punctuations, and the rules in which a specific language is written in a standardized way. Orthographies are often based on a phonemic analysis, but they almost always include idiosyncrasies because of historical developments (like sound changes or loans) and because of the widely-followed principle of lexical integrity (i.e. the attempt to write the same lexical root in a consistent way, also when synchronic phonemic rules change the pronunciation, as for example with final devoicing in many Germanic languages). All orthographies are language-specific (and often even resource-specific), although individual symbols or rules might be shared between languages. A PRACTICAL ORTHOGRAPHY is a strongly phoneme-based writing system designed for practical use by speakers. The mapping relation between phonemes

and graphemes in practical orthographies is purposely shallow, i.e. there is mostly a systematic and faithful mapping from a phoneme to a grapheme. Practical orthographies are intended to jumpstart written materials development by correlating a writing system with the sound units of a language (cf. Meinhof & Jones 1928). Symbols from the IPA are often used by linguists in the development of such practical orthographies for languages without writing systems, though this usage of IPA symbols should not be confused with transcription (as defined above).

Further, a TRANSLITERATION is a mapping between two different orthographies. It is the process of "recording the graphic symbols of one writing system in terms of the corresponding graphic symbols of a second writing system" (Kemp 2006: 396). In straightforward cases, such a transliteration is simply a matter of replacing one symbol with another. However, there are widespread complications, like one-to-many or many-to-many mappings, which are not always easy, or even possible, to solve without listing all cases individually (cf. Moran 2012: Ch. 2).

Script systems

Different kinds of writing systems are classified into script systems. A SCRIPT is a collection of distinct symbols as employed by one or more orthographies. For example, both Serbian and Russian are written with subsets of the Cyrillic script. A single language, like Serbian or Japanese, can also be written using orthographies based on different scripts. Over the years linguists have classified script systems in a variety of ways, with the tripartite classification of logographic, syllabic, and alphabetic remaining the most popular, even though there are at least half a dozen different types of script systems that can be distinguished (Daniels 1990, 1996).

Breaking it down further, a script consists of GRAPHEMES, and graphemes consist of CHARACTERS. In the linguistic terminology of writing systems, a CHARACTER is a general term for any self-contained element in a writing system.[7] Although in literate societies most people have a strong intuition about what the characters are in their particular orthography or orthographies, it turns out that the separation of an orthography into separate characters is far from trivial. The widespread intuitive notion of a character is strongly biased towards educational traditions, like the alphabet taught at schools, and technological possibilities, like the available type pieces in a printer's job case, the keys on a typewriter, or the

[7] There is a second interpretation of the term CHARACTER, i.e. a conventional term for a unit in the Chinese writing system (Daniels 1996). This interpretation will not be further explored in this paper.

symbols displayed in Microsoft Word's symbol browser. In practice, characters often consist of multiple building blocks, each of which could be considered a character in its own right. For example, although a Chinese character may be considered to be a single basic unanalyzable unit, at a more fine-grained level of analysis the internal structure of Chinese characters is often comprised of smaller semantic and phonetic units that should be considered characters (Sproat 2000). In alphabetic scripts, this problem is most forcefully exemplified by diacritics.

A DIACRITIC is a mark, or series of marks, that may be above, below, or through other characters (Gaultney 2002). Diacritics are sometimes used to distinguish homophonous words, but they are more often used to indicate a modified pronunciation (Daniels & Bright 1996: xli). The central question is whether, for example, <e>, <è>, <a> and <à> should be considered four characters, or different combinations of three characters. In general, multiple characters together can form another character, and it is not always possible to decide on principled grounds what should be the basic building blocks of an orthography.

For that reason, it is better to analyze an orthography as a collection of graphemes. A GRAPHEME is the basic, minimally distinctive symbol of a particular writing system, alike to the phoneme is an abstract representation of a distinct sound in a specific language. The term GRAPHEME was modeled after the term PHONEME and represents a contrastive graphical unit in a writing system (see Kohrt 1986 for a historical overview of the term grapheme). Most importantly, a single grapheme regularly consists of multiple characters, like <th>, <ou> and <gh> in English (note that each character in these graphemes is also a separate grapheme in English). Such complex graphemes are often used to represent single phonemes. So, a combination of characters is used to represent a single phoneme. Note that the opposite is also found in writing systems, in cases in which a single character represents a combination of two or more phonemes. For example, <x> in English orthography represents a combination of the phonemes /k/ and /s/.

Further, conditioned or free variants of a grapheme are called ALLOGRAPHS. For example, the distinctive forms of Greek sigma are conditioned, with <σ> being used word-internally and <ς> being used at the end of a word. In sum, there are many-to-many relationships between phonemes and graphemes as they are expressed in the myriad of language- and resource-specific orthographies.

This exposition of the linguistic terminology involved in describing writing systems has been purposely brief. We have highlighted some of the linguistic notions that are pertinent, yet sometimes confused with, the technological defini-

tions developed for the computational processing of the world's writing systems, which we describe in the next section.

1.4 The Unicode approach

The conceptualization and terminology of writing systems was rejuvenated by the development of the Unicode Standard, with major input from Mark Davis, co-founder and long-term president of the Unicode Consortium. For many years, "exotic" writing systems and phonetic transcription systems on personal computers were constrained by the American Standard Code for Information Interchange (ASCII) character encoding scheme, based on the Latin script, which only allowed for a strongly limited number of different symbols to be encoded. This implied that users could either use and adopt the (extended) Latin alphabet or they could assign new symbols to the small number of code points in the ASCII encoding scheme to be rendered by a specifically designed font (Bird & Simons 2003). In this situation, it was necessary to specify the font together with each document to ensure the rightful display of its content. To alleviate this problem of assigning different symbols to the same code points, in the late 80's and early 90's the Unicode Consortium set itself the ambitious goal of developing a single universal character encoding to provide a unique number, a code point, for every character in the world's writing systems. Nowadays, the Unicode Standard is the default encoding of the technologies that support the World Wide Web and for all modern operating systems, software and programming languages.

The Unicode Standard

The Unicode Standard represents a massive step forward because it aims to eradicate the distinction between universal (ASCII) versus language-particular (font) by adding as much as possible language-specific information into the universal standard. However, there are still language/resource-specific specifications necessary for the proper usage of Unicode, as will be discussed below. Within the Unicode structure many of these specifications can be captured by so-called LOCALE DESCRIPTIONS, so we are moving to a new distinction of universal (Unicode Standard) versus language-particular (locale description). The major gain is a much larger compatibility on the universal level (because Unicode standardizes a much larger portion of writing system diversity), and much better possibilities for automatic processing on the language-particular level (because locale descriptions are computer readable specifications).

Each version of the Unicode Standard (Unicode Consortium 2014, as of writing at version 7) consists of a set of specifications and guidelines that include (i) a core specification, (ii) code charts, (iii) standard annexes and (iv) a character database.[8] The CORE SPECIFICATION is a book directed toward human readers that describes the formal standard for encoding multilingual text. The CODE CHARTS provide a humanly readable online reference to the character contents of the Unicode Standard in the form of PDF files. The UNICODE STANDARD ANNEXES (UAX) are a set of technical standards that describe the implementation of the Unicode Standard for software development, Web standards, and programming languages. The UNICODE CHARACTER DATABASE (UCD) is a set of computer-readable text files that describe the character properties, including a set of rich character and writing system semantics, for each character in the Unicode Standard. In this section, we introduce the basic Unicode concepts, but we will leave out many details. Please consult the above mentioned full documentation for a more detailed discussion. Further note that the Unicode Standard is exactly that, namely a standard. It normatively describes notions and rules to be followed. In the actual practice of applying this standard in a computational setting, a specific implementation is necessary. The most widely used implementation of the Unicode Standard is the INTERNATIONAL COMPONENTS FOR UNICODE (ICU), which offers C/C++ and Java libraries implementing the Unicode Standard.[9]

Character encoding system

The Unicode Standard is a CHARACTER ENCODING SYSTEM which goal it is to support the interchange and processing of written characters and text in a computational setting. Underlyingly, the character encoding is represented by a range of numerical values called a CODE SPACE, which is used to encode a set of characters. A CODE POINT is a unique non-negative integer within a code space (i.e. within a certain numerical range). In the Unicode Standard character encoding system, an ABSTRACT CHARACTER, for example the LATIN SMALL LETTER P, is mapped to a particular code point, in this case the decimal value 112, normally represented in

[8] All documents of the Unicode Standard are available at http://www.unicode.org/versions/latest/. For a quick survey of the use of terminology inside the Unicode Standard, their glossary is particularly useful, available at http://www.unicode.org/glossary/. For a general introduction to the principles of Unicode, Chapter 2 of the core specification, called GENERAL STRUCTURE, is particularly insightful. Different from many other documents of the Unicode Standard, this general introduction is relatively easy to read and illustrated with many interesting examples from various orthography traditions all over the world.

[9] More information about the ICU is available here: http://icu-project.org.

hexadecimal, which then looks in Unicode parlance as U+0070.[10] That encoded abstract character is rendered on a computer screen (or printed page) as a GLYPH, e.g. <p>, depending on the FONT and the context in which that character appears.

In Unicode Standard terminology, an (abstract) CHARACTER is the basic encoding unit. The term CHARACTER can be quite confusing due to its alternative definitions across different scientific disciplines and because in general the word CHARACTER means many different things to different people. It is therefore often preferable to refer to Unicode characters simply as CODE POINTS, because there is a one-to-one mapping between Unicode characters and their numeric representation. In the Unicode approach, a character refers to the abstract meaning and/or general shape, rather than a specific shape, though in code tables some form of visual representation is essential for the reader's understanding. Unicode defines characters as abstractions of orthographic symbols, and it does not define visualizations for these characters (although it does presents examples). In contrast, a GLYPH is a concrete graphical representation of a character as it appears when rendered (or rasterized) and displayed on an electronic device or on printed paper. For example, <g *g* **g** g ***g*** **g**> are different glyphs of the same character, i.e. they may be rendered differently depending on the typography being used, but they all share the same code point. From the perspective of Unicode they are *the same thing*. In this approach, a FONT is then simply a collection of glyphs linked to code points. Allography is not specified in Unicode (expect for a few exceptional cases, due to legacy encoding issues), but can be specified in a font as a CONTEXTUAL VARIANT (a.k.a. presentation form).

Each code point in the Unicode Standard is associated with a set of CHARACTER PROPERTIES as defined by the Unicode character property model.[11] Basically, those properties are just a long list of values for each character. For example, code point U+0047 has the following properties (among many others):

- Name: LATIN CAPITAL LETTER G
- Alphabetic: YES
- Uppercase: YES

[10] Hexadecimal (base-16) 0070 is equivalent to decimal (base-10) 112, which can be calculated by considering that $(0 \cdot 16^3) + (0 \cdot 16^2) + (7 \cdot 16^1) + (0 \cdot 16^0) = 7 \cdot 16 = 112$. Underlyingly, computers will of course treat this code point binary (base-2) as 11100000, as can be seen by calculating that $(1 \cdot 2^7) + (1 \cdot 2^6) + (1 \cdot 2^5) + (0 \cdot 2^4) + (0 \cdot 2^3) + (0 \cdot 2^2) + (0 \cdot 2^1) + (0 \cdot 2^0) = 64 + 32 + 16 = 112$.

[11] The character property model is described in http://www.unicode.org/reports/tr23/, but the actual properties are described in http://www.unicode.org/reports/tr44/. A simplified overview of the properties is available at http://userguide.icu-project.org/strings/properties. The actual code tables listing all properties for all Unicode code points are available at http://www.unicode.org/Public/UCD/latest/ucd/.

- Script: LATIN
- Extender: NO
- Simple_Lowercase_Mapping: 0067

These properties contain the basic information of the Unicode Standard and they are necessary to define the correct behavior and conformance required for interoperability in and across different software implementations (as defined in the Unicode Standard Annexes). The character properties assigned to each code point is based on each character's behavior in the real-world writing traditions. For example, the corresponding lowercase character to U+0047 is U+0067 (though note that the relation between uppercase and lowercase is in many situations much more complex than this, and Unicode has further specifications for those cases). Another use of properties is to define the script of a character.[12] In practice, script is simply defined for each character as the explicit SCRIPT property in the Unicode Character Database.

One frequently referenced property is the BLOCK property, which is often used in software applications to impose some structure to the large number of Unicode characters. Each character in Unicode belongs to a specific block. These blocks are basically an organizational structure to alleviate the administrative burden of keeping Unicode up-to-date. Blocks consist of characters that in some way belong together, so that characters are easier to find. Some blocks are connected with a specific script, like the Hebrew block or the Gujarati block. However, blocks are predefined ranges of code points, and often there will come a point after which the range is completely filled. Any extra characters will have to be assigned somewhere else. There is, for example, a block ARABIC, which contains most Arabic symbols. However, there is also a block ARABIC SUPPLEMENT, ARABIC PRESENTATION FORMS-A and ARABIC PRESENTATION FORM-B. The situation with Latin symbols is even more extreme. In general, the names for block should not be taken as a definitional statement. For example, many IPA symbols are not located in the aptly-names block IPA EXTENSIONS, but in other blocks (see Section 3.2).

[12] The Unicode Glossary defines the term SCRIPT as a "collection of letters and other written signs used to represent textual information in one or more writing systems. For example, Russian is written with a subset of the Cyrillic script; Ukrainian is written with a different subset. The Japanese writing system uses several scripts."

Grapheme clusters

There are many cases in which a sequence of characters (i.e. a sequence of more than one code point) represents what a user perceives as an individual unit in a particular orthographic writing system. For this reason the Unicode Standard differentiates between ABSTRACT CHARACTER and USER-PERCEIVED CHARACTER. Sequences of multiple code points that correspond to a single user-perceived characters are called grapheme clusters in Unicode parlance. Grapheme clusters come in two flavors: (default) grapheme clusters and tailored grapheme clusters.

The (default) GRAPHEME CLUSTERS are locale-independent graphemes, i.e. they always apply when a particular combination of characters occurs independent of the writing system in which they are used. These character combinations are defined in the Unicode Standard as functioning as one TEXT ELEMENT.[13] The simplest example of a grapheme cluster is a base character followed by a letter modifier character. For example, the sequence <n> + <˜> (i.e. LATIN SMALL LETTER N at U+006E, followed by COMBINING TILDE at U+0303) combines visually into <ñ>, a user-perceived character in writing systems like that of Spanish. So, what the user perceives as a single character actually involves a multi-code-point sequence. Note that this specific sequence can also be represented with a single so-called PRECOMPOSED code point, LATIN SMALL LETTER N WITH TILDE at U+00F1, but this is not the case for all multi-code point character sequences. The problem that there multiple encodings possible for the same text element has been acknowledged early on in the Unicode Standard (e.g. for <ñ>, the sequence U+006E U+0303 should in all situations be treated identically to the precomposed U+00F1), and a system of CANONICAL EQUIVALENCE is available for such situations. Basically, the Unicode Standard offers different kind of normalizations to either decompose all precomposed characters (called NFD, NORMALIZATION FORM CANONICAL DECOMPOSITION), or compose as much as possible combinations (called NFC, NORMALIZATION FORM CANONICAL COMPOSITION). In current practice of software development, NFC seems to be preferred in most situations and is widely proposed as the preferred canonical form.

More difficult for text processing, because less standardized, is what the Unicode Standard terms TAILORED GRAPHEME CLUSTERS. Tailored grapheme clusters are locale-dependent graphemes, i.e. such combination of characters do not function as text elements in all situations. For example, the sequence <c> + <h> for the Slovak digraph <ch> or the sequence <ky> in the Sisaala practical orthog-

[13] The Unicode Glossary defines text element as: "A minimum unit of text in relation to a particular text process, in the context of a given writing system. In general, the mapping between text elements and code points is many-to-many."

raphy (pronounced as IPA /tʃ/, Moran 2006). These grapheme clusters are TAILORED in the sense that they must be specified on a language-by-language or writing-system-by-writing-system basis. The Unicode Standard provides technological specifications for creating locale specific data in so-called UNICODE LOCALE DESCRIPTIONS, i.e. a set of specification that defines a set of language-specific elements (e.g. tailored grapheme clusters, collation order, capitalization-equivalence), as well as other special information, like how to format numbers, dates, or currencies. Locale descriptions are saved in the COMMON LOCALE DATA REPOSITORY (CLDR),[14] a repository of language-specific definitions of writing system properties, each of which describes specific usages of characters. Each locale can be encoded in a document using the LOCALE DATA MARKUP LANGUAGE (LDML). LDML is an XML format and vocabulary for the exchange of structured locale data. Unicode Locale Descriptions allow users to define language- or even resource-specific writing systems or orthographies.[15] However, there are various drawbacks of locale descriptions for the daily practice of linguistic work in a multilingual setting.

[14] More information about the CLDR can be found here: http://cldr.unicode.org/.

[15] The Unicode Glossary defines WRITING SYSTEM only very loosely, as it is not a central concept in the Unicode Standard. A writing system is, "A set of rules for using one or more scripts to write a particular language. Examples include the American English writing system, the British English writing system, the French writing system, and the Japanese writing system."

2 Unicode pitfalls

2.1 Wrong it is not

In this chapter we describe some of the most common pitfalls that we have encountered when using the Unicode Standard in our own work, or in discussion with other linguists. This section is not meant as a criticism of the decisions made by the Unicode Consortium; on the contrary, we aim to highlight where the technological aspects of the Unicode Standard diverge from many users' intuitions. What have sometimes been referred to as problems or inconsistencies in the Unicode Standard are mostly due to legacy compatibility issues, which can lead to unexpected behavior by linguists using the standard. However, there are also some cases in which the Unicode Standard has made decisions that theoretically could have been taken differently, but for some reason or another (mostly very good reasons) were accepted as they are now. We call behavior that executes without error but does something different than the user expected—often unknowingly—a PITFALL.

In this context, it is important to realize that the Unicode Standard was not developed to solve linguistic problems per se, but to offer a consistent computational environment for written language. In those cases in which the Unicode Standard behaves differently as expected, we think it is important not to dismiss Unicode as "wrong" or "deficient", because our experience is that in almost all cases the behavior of the Unicode Standard has been particularly well thought through. The Unicode Consortium has a more wide-ranging view of matters and often examines important practical use-cases that from a linguistic point of view are normally not considered. Our general guideline for dealing with the Unicode Standard is to accept it as it is, and not to battle windmills. Alternatively, of course, it is possible to actively engage in the development of the standard itself, an effort that is highly appreciated by the Unicode Consortium.

2.2 Pitfall: Characters are not glyphs

A central principle of Unicode is the distinction between character and glyph. A character is the abstract notion of a symbol in a writing system, while a glyph is the concrete drawing of such a symbol. In practice, there is a complex interaction between characters and glyphs. A single Unicode character may of course be rendered as a single glyph. However, a character may also be a piece of a glyph, or vice-versa. Actually, all possible relations between glyphs and characters are attested.

First, a single character may have different contextually determined glyphs. For example, characters in writing systems like Hebrew and Arabic have different glyphs depending on where they appear in a word. Some letters in Hebrew change their form at the end of the word, and in Arabic, primary letters have four contextually-sensitive variants (isolated, word initial, medial and final). Second, a single character may be rendered as a sequence of multiple glyphs. For example, in Tamil one Unicode character may result in a combination of a consonant and vowel, which are rendered as two adjacent glyphs by fonts that supports Tamil. Third, a single glyph may be a combination of multiple characters. For example, the ligature <fi>, a single glyph, is the result of two characters, <f> and <i>, that have undergone glyph substitution by font rendering (see also Section 2.4). Like contextually-determined glyphs, ligatures are (intended) artifacts of text processing instructions. Finally, a single glyph may be a part of a character, as exemplified by diacritics.

Further, the rendering of a glyph is dependent on the font being used. For example, the Unicode character LATIN SMALL LETTER G appears as <g> and <ɡ> in the Linux Libertine and Courier fonts, respectively, because their typefaces are designed differently. Furthermore, font face may change the visual appearance of a character, for example Times New Roman two-story <a> changes to a single-story a in italics <*a*>. This becomes a real problem for some phonetic typesetting (see Section 3.3).

In sum, character-to-glyph mappings are complex technological issues that the Unicode Consortium has had to address in the development of the Unicode Standard, but for the lay user they can be utterly confusing because visual rendering does not (necessarily) indicate logical encoding.

2.3 Pitfall: Missing glyphs

The Unicode Standard is often praised (and deservedly so) for solving many of the perennial problems with the interchange and display of the world's writing systems. However, a common complaint from users is that, while the praise may be true, they mostly just see some boxes on their screen instead of those promised symbols. The problem of course is that users' computers do not have any glyphs installed matching the Unicode code points in the file they are trying to inspect. It is important to realize that internally in the computer everything still works as expected: any handling of Unicode code points works independently of how they are displayed on the screen. So, although a user might only see boxes being displayed, this user should be assured that everything is still in order.

The central problem behind the missing glyphs is that designing actual glyphs includes a lot of different considerations and it is a time-consuming process. Many traditional expectations of how specific characters should look like have to be taken into account when designing glyphs. Those expectations are often not well documented, and it is mostly up to the knowledge and experience of the font designer to try and conform to them as good as possible. Therefore, most designers produce fonts only including glyphs for certain parts of the Unicode Standard, namely for those characters they feel comfortable with. At the same time, the number of characters defined by the Unicode Standard is growing with each new version, so it is neigh impossible for any designer to produce glyphs for all characters. The result of this is that, almost necessarily, each font only includes glyphs for a subset of the characters in the Unicode Standard.

The simple solution to missing glyphs is thus to install additional fonts providing additional glyphs. For the more exotic characters there is often not much choice. There are a few particularly large fonts that might be considered. First, there is the Everson Mono font made by Michael Everson, which currently includes 9,756 different glyphs (not including Chinese) updated up to Unicode 7.0.[1] Already a bit older is the Titus Cyberbit Basic font made by Jost Gippert and Carl-Martin Bunz, which includes 10,044 different glyphs (not including Chinese), but not including newer characters added after Unicode 4.0.[2]

Further, we suggest to always install at least one so-called fall-back font, which provides glyphs that at least show the user some information about the underlying encoded character. Apple Macintoshes have such a font (which is invisible to the user), which is designed by Michael Everson and made available

[1] Everson Mono is available as shareware at http://www.evertype.com/emono/.

[2] Titus Cyberbit Basic is available at http://titus.fkidg1.uni-frankfurt.de/unicode/tituut.asp.

for other systems through the Unicode Consortium.[3] Further, the GNU Unifont is a clever way to produce bitmaps approximating the intended glyph of each available character, updated to Unicode 7.0.[4] Finally, the Summer Institute of Linguistics provides a SIL Unicode BMP Fallback Font, currently available up to Unicode version 6.1. This font does not even attempt to show a real glyph, but only shows the hexadecimal code inside a box for each character, so a user can at least see the Unicode codepoint of the character to be displayed.[5]

2.4 Pitfall: Faulty rendering

A similar complaint to missing glyphs, discussed previously, is that while there might be a glyph being displayed, it does not look right. There are two reasons for unexpected visual display, namely automatic font substitution and faulty rendering. Like missing glyphs, any such problems are independent from the Unicode Standard. The Unicode Standard only includes very general information about characters and leaves the specific visual display to others to decide on. Any faulty display is thus not to be blamed on the Unicode Consortium, but on a complex interplay of different mechanisms happening in a computer to turn Unicode codepoints into visual symbols. We will only sketch a few aspects of this complex interplay here.

Most modern software applications (like Microsoft Word) offer some approach to automatic font substitution. This means that when a text is written in a specific font (e.g. Times New Roman) and an inserted Unicode character does not have a glyph within this font, then the software application will automatically search for another font to display the glyph. The result will be that this specific glyph will look slightly different from the others. This mechanism works differently depending on the software application, and mostly only limited user influence is expected and little feedback is given, which might be rather frustrating to font-aware users.[6]

[3] The Apple/Everson fallback font is available for non-Macintosh users at http://www.unicode.org/policies/lastresortfont_eula.html.

[4] The GNU Unifont is available at http://unifoundry.com/unifont.html.

[5] The SIL Unicode BMP Fallback Font is available at http://scripts.sil.org/UnicodeBMPFallbackFont.

[6] For example, Apple Pages does not give any feedback that a font is being replaced, and the user does not seem to have any influence on the choice of replacement (except by manually marking all occurrences). In contrast, Microsoft Word does indicate the font replacement by showing the name in the font menu of the font replacement. However, Word simply changes

The other problem with visual display is related to the so-called FONT RENDERING. Font rendering refers to the process of the actual positioning of Unicode characters on a page of written text. This positioning is actually a highly complex problem, and many things can go wrong in the process. Well-known rendering problems, like proportional glyph size or ligatures are reasonably well understood. In contrast, the positioning of multiple diacritics relative to a base character is still a widespread problem, even within the Latin script. Especially when more than one diacritic is supposed to be placed above (or below) each other, this often leads to unexpected effects in many modern software applications. The problems arising in Arabic and in many southeast Asian scripts (like Devanagari or Burmese) are even more complex.

To understand where any problems arise it is important to realize that there are basically three different approaches to font rendering. The most widespread is Adobe's and Microsoft's OPENTYPE system. This approach makes it relatively easy for font developers, as the font itself does not include all details about the precise placement of individual characters. For those details, additional script-descriptions are necessary. All of those systems can lead to unexpected behavior.[7] Alternative systems are APPLE ADVANCED TYPOGRAPHY (AAT) and the open-source GRAPHITE system from the Summer Institute of Linguistics (SIL).[8] In both of these systems, a larger burden is placed on the description inside the font.

There is mostly no real solution to problems arising from faulty font rendering. Switching to another software application that offers better handling is the only real alternative, but this is normally not an option for daily work. The experience with rendering on the side of the software industry is developing quickly, so we can expect the situation only to get better. In the meantime one can try to correct faulty layout by tweaking baseline and/or kerning (when such option are available).

---

the font completely, so any text written after the replacement is written in a different font as before. Both behaviors leave much to be desired.

[7] For more details about OpenType, see http://www.adobe.com/products/type/opentype.html and http://www.microsoft.com/typography/otspec/. Additional systems for complex text layout are, among others, Microsoft's DirectWrite https://msdn.microsoft.com/library/dd368038.aspx and the open-source project HarfBuzz http://www.freedesktop.org/wiki/Software/HarfBuzz/.

[8] More information about AAT can be found at https://developer.apple.com/fonts/.
SIL's Graphite is described in detail at http://scripts.sil.org/cms/scripts/page.php?site_id=projects&item_id=graphite_home.

2.5 Pitfall: Blocks

The Unicode code space is subdivided into blocks of contiguous code points. For example, the block called Cyrillic runs from U+0400 till U+04FF. These blocks arose as an attempt at ordering the enormous amount of characters in Unicode, but the ideas of blocks very quickly ran into problems. First, the size of a block is fixed, so when a block is full, a new block will have to be instantiated somewhere further in the code space. For example, this led to the blocks Cyrillic Supplement, Cyrillic Extended-A (both of which are also already full) and Cyrillic Extended-B. Second, when a specific character already exists, then it is not duplicated in another block, although the name of the block might indicate that a specific symbol should be available there. In general, names of blocks are just an approximate indication of the kind of characters that will be in the block.

The problem with blocks arises because finding the right character among the thousands of Unicode characters is not easy. Many software applications present blocks as a primary search mechanism, because the block names suggest where to look for a particular character. However, when a user searches for an IPA character in the block IPA Extensions, then many IPA characters will not be found there. For example, the velar nasal <ŋ> is not part of the block IPA Extensions because it was already included as latin small letter eng at U+014B in the block Latin Extensions-A.

In general, finding a specific character in the Unicode Standard is often not trivial. The names of the blocks can help, but they are not (and never were supposed to be) a foolproof structure. It is not the goal nor aim of the Unicode Consortium to provide a user interface to the Unicode Standard. If one often encounters the problem of needing to find a suitable character, there are various other useful services for end-users available.[9]

2.6 Pitfall: Homoglyphs

Homoglyphs are visually indistinguishable glyphs (or highly similar glyphs) that have different code points in the Unicode Standard and thus different character

[9] The Unicode website offers a basic interface to the code charts at http://www.unicode.org/charts/index.html. As a more flexible interface, we particularly like PopChar from Macility, available for both Macintosh and Windows. There are also various free websites that offer search interfaces to the Unicode code tables, like http://unicode-search.net or http://unicode-search.net. A further useful approach for searching characters using shape matching is http://shapecatcher.com.

semantics. As a principle, the Unicode Standard does not specify how a character appears visually on the page or the screen. So in most cases, a different appearance is caused by the specific design of a font, or by user-settings like size or boldface. Taking an example already discussed in Section 2.6, the following symbols <g *g* **g** g ***g*** **g**> are different glyphs of the same character, i.e. they may be rendered differently depending on the typography being used, but they all share the same code point (viz. LATIN SMALL LETTER G at U+0067). In contrast, the symbols <AAAAAAAAA𝗔𝙰> are all different code points, although they look highly similar—in some cases even sharing exactly the same glyph in some fonts. All these different A-like characters include the following code points in the Unicode Standard:

<A> LATIN CAPITAL LETTER A, at U+0041
<A> CYRILLIC CAPITAL LETTER A, at U+0410
<A> GREEK CAPITAL LETTER ALPHA, at U+0391
<A> CHEROKEE LETTER GO, at U+13AA
<A> CANADIAN SYLLABICS CARRIER GHO, at U+15C5
<ᴀ> LATIN SMALL LETTER CAPITAL A, at U+1D00
<A> LISU LETTER A, at U+A4EE
<A> CARIAN LETTER A, at U+102A0
<𝗔> MATHEMATICAL SANS-SERIF CAPITAL A, U+1D5A0
<𝙰> MATHEMATICAL MONOSPACE CAPITAL A, at U+1D670

The existence of such homoglyphs is partly due to legacy compatibility, but for the most part these characters are simply different characters that happen to look similar.[10] Yet, they are supposed to behave different from the perspective of a font designer. For example, when designing a Cyrillic font, the <A> will have different aesthetics and different traditional expectation compared to a Latin <A>.

Such homoglyphs are a widespread problem for consistent encoding. Although for most users it looks like the words <voces> and <voces> are almost identical, in actual fact they do not even share a single code point.[11] For computers these two words are completely different entities. Commonly, when users with Cyrillic or Greek keyboards have to type some Latin-based orthography, they mix similar looking Cyrillic or Greek characters into their text, because those characters are

[10] A particularly nice interface to look for homoglyphs is http://shapecatcher.com, based on the principle of recognizing shapes (Belongie, Malik & Puzicha 2002).

[11] The first words consists completely of Latin characters, namely U+0076, U+006F, U+0063, U+0065 and U+0073, while the second is a mix of Cyrillic and Greek characters, namely U+03BD, U+03BF, U+0041, U+0435 and U+0455.

so much easier to type. Similarly, when users want to enter an unusual symbol, they normally search by visual impression in their favorite software application, and just pick something that looks reasonably alike to what they expect the glyph to look like.

It is really easy to make errors at text entry and add characters that are not supposed to be included. Our proposals for orthography profiles (see Chapter 4) are a method for checking the consistency of any text. In situations in which interoperability is important, we consider it crucial to add such checks in any workflow.

2.7 Pitfall: Canonical equivalence

For some characters, there is more than one possible encoding in the Unicode Standard. This is a possible pitfall, as this would mean that for the computer there exist multiple different entities that for a user are the same. This would, for example, lead to problems with searching, as the computer would search for specific encodings, and not find all expected characters. As a solution, the Unicode Standard includes a notion of CANONICAL EQUIVALENCE. Different encodings are explicitly declared as equivalent in the Unicode Standard code tables. Further, to harmonize all encodings in a specific piece of text, the Unicode Standard proposes a mechanism of NORMALIZATION.

Consider for example the characters and following Unicode code points:

<Å> LATIN CAPITAL LETTER A WITH RING ABOVE U+00C5
<Å> ANGSTROM SIGN U+212B
<Å> LATIN CAPITAL LETTER A U+0041 + COMBINING RING ABOVE U+030A

The character, represented here by glyph <Å>, is encoded in the Unicode Standard in the first two examples by a single-character sequence; each is assigned a different code point. In the third example, the glyph is encoded in a multiple-character sequence that is composed of two character code points. All three sequences are CANONICALLY EQUIVALENT, i.e. they are strings that represent the same abstract character and because they are not distinguishable by the user, the Unicode Standard requires them to be treated the same in regards to their behavior and appearance. Nevertheless, they are encoded differently. For example, if one were to search an electronic text (with software that does not apply Unicode Standard normalization) for ANGSTROM SIGN (U+212B), then the instances of LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5) would not be found.

In other words, there are equivalent sequences of Unicode characters that should be normalized, i.e. transformed into a unique Unicode-sanctioned representation of a character sequence called a NORMALIZATION FORM. Unicode provides a Unicode Normalization Algorithm, which essentially puts combining marks into a specific logical order and it defines decomposition and composition transformation rules to convert each string into one of four normalization forms. We will discuss here the two most relevant normalization forms: NFC and NFD.

The first of the three characters above is considered the NORMALIZATION FORM C (NFC), where C stands for composition. When the process of NFC normalization is applied to the character sequences in 2 and 3, both sequences are normalized into the PRE-COMPOSED character sequence in 1. Thus all three canonical character sequences are standardized into one composition form in NFC.The other central Unicode normalization form is the NORMALIZATION FORM D (NFD), where D stands for decomposition. When NFD is applied to the three examples above, all three, including importantly the single-character sequences in 1 and 2, are normalized into the DECOMPOSED multiple-sequence of characters in 3. Again, all three are then logically equivalent and therefore comparable and syntactically interoperable.

As illustrated, some characters in the Unicode Standard have alternative representations (in fact, many do), but the Unicode Normalization Algorithm can be used to transform certain sequences of characters into canonical forms to test for equivalency. To determine equivalence, each character in the Unicode Standard is associated with a combining class, which is formally defined as a character property called CANONICAL COMBINING CLASS which is specified in the Unicode Character Database. The combining class assigned to each code point is a numeric value between 0 and 254 and is used by the Unicode Canonical Ordering Algorithm to determine which sequences of characters are canonically equivalent. Normalization forms, as very briefly described above, can be used to ensure character equivalence by ordering character sequences so that they can be faithfully compared.

It is very important to note that any software applications that is Unicode Standard compliant is free to change the character stream from one representation to another. This means that a software application may compose, decompose or reorder characters as its developers desire; as long as the resultant strings are canonically equivalent to the original. This might lead to unexpected behavior for users. Various players, like the Unicode Consortium, the W3C, or the TEI recommend NFC in most user-directed situations, and some software applica-

tions that we tested indeed seem to automatically convert strings into NFC.[12] This means in practice that if a user, for example, enters <a> and <`>, i.e. LATIN SMALL LETTER A at U+0061 and COMBINING GRAVE ACCENT at U+0300, this might be automatically converted into <à>, i.e. LATIN SMALL LETTER A WITH GRAVE at U+00E0.[13]

2.8 Pitfall: Absence of canonical equivalence

Although in most cases canonical equivalence will take care of alternative encodings of the same character, there are a some cases in which the Unicode Standard decided against equivalence. This leads to identical characters that are not equivalent, like <ø> LATIN SMALL LETTER O WITH STROKE at U+00F8 and <ø> a combination of LATIN SMALL LETTER O at U+006F with COMBINING SHORT SOLIDUS OVERLAY at U+0037. The general rule followed is that extensions of Latin characters that are connected to the base character are not separated as combining diacritics. For example, characters like <ŋ ɲ ɳ> or <ɖ ɗ> are obviously derived from <n> and <d> respectively, but they are treated like new separate characters in the Unicode Standard. Likewise, characters like <ø> and <ƈ> are not separated into a base character <o> and <c> with an attached combining diacritic.

Interestingly, and somewhat illogically, there are three elements, which are directly attached to their base characters, but which are still treated as separable in the Unicode Standard. Such characters are decomposed (in NFD normalization) in a base character with a combining diacritic. However, it is these cases that should be considered the exceptions to the rule. These three elements are the following:

- the COMBINING CEDILLA at U+0327
  This diacritic is for example attested in the precomposed character <ç> LATIN SMALL LETTER C WITH CEDILLA at U+00E7. This <ç> will thus be decomposed in NFC normalization.
- the COMBINING OGONEK at U+0328
  This diacritic is for example attested in precomposed <ą> LATIN SMALL LET-

[12] See the summary of various recommendation here: http://www.win.tue.nl/~aeb/linux/uc/nfc_vs_nfd.html.

[13] The behavior of software applications can be quite erratic in this respect. For example, Apple's TextEdit does not do any conversion on text entry. However, when you copy and paste some text inside the same document in rich text mode (i.e. RTF-format), it will be transformed into NFC on paste. Saving a document does not do any conversion to the glyphs on screen, but it will save the characters in NFC.

TER A WITH OGONEK at U+0105. This <ą> will thus be decomposed in NFC normalization.

- the COMBINING HORN at U+031B
  This diacritic is for example attested in precomposed <ơ> LATIN SMALL LETTER O WITH HORN at U+01A1. This <ơ> will thus be decomposed in NFC normalization.

There are further combinations that deserve specific care because it is actually possible to produce identical characters in different ways without them being canonically equivalent. In these situations, the general rule holds, namely that characters with attached extras are not decomposed. However, in the following cases the "extras" actually exist as combining diacritics, so there is also the possibility to construct a character by using a base character with those combining diacritics.

- First, there are the combining characters designated as "combining overlay" in the Unicode Standard, like <◌̃> COMBINING TILDE OVERLAY at U+0334 or <◌̵> COMBINING SHORT STROKE OVERLAY at U+0335. There are many characters that look like they are precomposed with such an overlay, for example <ł ƀ đ ꝑ> or <ɨ ɨ̵ ɟ ɾ̵>, or also the example of <ø> given at the start of this section. However, they are not decomposed in NFD normalization.
- Second, the same situation also occurs with combining characters designated as "combining hook", like <◌̡> COMBINING PALATALIZED HOOK BELOW at U+0321. This element seems to occur in precomposed characters like <ɦ ɖ ʄ ɠ>. However, they are not decomposed in NFD normalization.

To harmonize the encoding in these cases it is not sufficient to use Unicode normalization. Additional checks are necessary, for example by using orthography profiles (see Chapter 4).

2.9 Pitfall: File formats

Unicode is a character encoding standard, but these characters of course actually appear inside some kind of computer file. The most basic Unicode-based file format is pure line-based text, i.e. strings of Unicode-encoded characters separated by line breaks (note that these line breaks are what for most people intuitively corresponds to paragraph breaks). Unfortunately, even within this apparently basic setting there exist a multitude of variants. In general, these different possibilities are well-understood in the software industry, and nowadays they normally

do not lead to any problems for the end user. However, there are some situations in which a user is suddenly confronted with cryptic questions in the user interface involving abbreviations like LF, CR, BE, LE or BOM. Most prominently this occurs with exporting or importing data in several software applications from Microsoft. Basically, there are two different issues involved. First, the encoding of line breaks and, second, the encoding of the Unicode characters into code units and the related issue of endianness.

Line breaks

The issue with LINE BREAKS originated with the instructions necessary to direct a printing head of a physical printer to a new line. This involves two movements, known as CARRIAGE RETURN (CR, returning the printing head to the start of the line on the page) and LINE FEED (LF, moving the printing head to the next line on the page). Physically, these are two different events, but conceptually together they form one action. In the history of computing, various encodings of line breaks have been used (e.g. CR+LF, LF+CR, only LF, or only CR). Currently, all Unix and Unix-derived systems use only LF as code for a line break, while software from Microsoft still uses a combination of CR+LF. Today, most software applications recognize both options, and are able to deal with either encoding of line breaks (until rather recently this was not the case, and using the wrong line breaks would lead to unexpected errors). Our impression is that there is a strong tendency in software development to standardize on the simpler "only LF" encoding for line breaks, and we suggest that everybody use this encoding whenever possible.

Code units

The issue with CODE UNITS stems from the question how to separate a stream of binary ones and zero, i.e. bits, into chunks representing Unicode characters. A code unit is the sequence of bits used to encode a single character in an encoding. Depending on different use cases, the Unicode Standard offers three different approaches, called UTF-32, UTF-16 and UTF-8.[14] The details of this issue is extensively explained in section 2.5 of the Unicode Core Specification Unicode Consortium (2014).

[14] The letters UTF stand for UNICODE TRANSFORMATION FORMAT, but the notion of "transformation" is a legacy notion that does not have meaning anymore. Nevertheless, the designation UTF (in capitals) has become an official standard designation, but should probably best be read as simply "Unicode Format."

Basically, UTF-32 encodes each character in 32 bits (32 *bi*nary uni*ts*, i.e. 32 zeros or ones) and is the most disk-space-consuming variant of the three. However, it is the most efficient encoding processing-wise, because the computer simply has to separate each character after 32 bits.

In contrast, UTF-16 uses only 16 bits per character, which is sufficient for the large majority of Unicode characters, but not for all of them. A special system of SURROGATES is defined within the Unicode Standard to deal with these additional characters. The effect is a more disk-space efficient encoding (approximately half the size), while adding a limited computational overhead to manage the surrogates.

Finally, UTF-8 is a more complex system that dynamically encodes each character with the minimally necessary number of bits, choosing either 8, 16 or 32 bits depending on the character. This represents again a strong reduction in space (particularly due to the high frequency of data using erstwhile ASCII characters, which need only 8 bits) at the expense of even more computation necessary to process such strings. However, because of the ever growing computational power of modern machines, the processing overhead is in most practical situations a non-issue, while saving on space is still useful, particularly for sending texts over the Internet. As an effect, UTF-8 has become the dominant encoding on the World Wide Web. We suggest that everybody uses UTF-8 as their default encoding.

A related problem is a general issue about how to store information in computer memory, which is known as ENDIANNESS. The details of this issue go beyond the scope of this book. It suffices to realize that there is a difference between BIG-ENDIAN (BE) storage and LITTLE-ENDIAN (LE) storage. The Unicode Standard offers a possibility to explicitly indicate what kind of storage is used by starting a file with a so-called BYTE ORDER MARK (BOM). However, the Unicode Standard does not require the usage of BOM, preferring other non-Unicode methods to signal to computers which kind of endianness is used. This issue only arises with UTF-32 and UTF-16 encodings. When using the preferred UTF-8, using a BOM is theoretically possible, but strongly dispreferred according to the Unicode Standard. We suggest that everyone tries to prevent the inclusion of BOM in your data.

2.10 Recommendations

Summarizing the pitfalls discussed in this chapter, we propose the following recommendations:

- To prevent strange boxes instead of nice glyphs, always install a few fonts with a large glyph collection and at least one fall-back font (see Section 2.3).
- Unexpected visual impressions of symbols does not necessarily mean that the actual encoding is wrong. It is mostly a problem of faulty rendering (see Section 2.4).
- Do not trust the names of Unicode blocks as a strategy to find specific characters (see Section 2.5)
- To ensure consistent encoding of texts, apply Unicode normalization (NFC or NFD, see Section 2.7).
- To prevent remaining inconsistencies after normalization, for example stemming from homoglyphs (see Section 2.6) or from missing canonical equivalence in the Unicode Standard (see Section 2.8) use orthography profiles (see Chapter 4).
- As a preferred file format, use Unicode Format UTF-8 in Normalization Form Composition (NFC) with LF line endings, but without byte order mark (BOM), whenever possible (see Section 2.9). This last nicely cryptic recommendation has T-shirt potential:

I prefer it
UTF-8 NFC LF no BOM

3 IPA meets Unicode

3.1 The International Phonetic Alphabet (IPA)

The International Phonetic Alphabet (IPA) is a common standard in linguistics to transcribe sounds of spoken language into Latin-based characters (International Phonetic Association 2005). Although IPA is reasonably easily adhered to with pen and paper, it is not trivial to encode IPA characters electronically. Similar to the previous chapter, this chapter will discuss various pitfalls with the encoding of the IPA. The details of the encoding are unimportant as long as the transcription is only directed towards phonetically trained eyes. For a linguist reading an IPA transcription, many of the details that will be discussed in this chapter might seem like hair-splitting trivialities. However, if IPA transcriptions are intended to be used across resources (e.g. searching similar phenomena across different languages) then it becomes crucial that there are strict encoding guidelines. Our main goal in this chapter is to present the encoding issues and propose recommendations for a "strict" IPA encoding.

For a long time, linguists, like all other computer users, were limited to ASCII-encoded 7-bit characters, which only includes Latin characters, numbers and some punctuation and symbols. Restricted to these standard character sets that lacked IPA support or other language-specific graphemes that they needed, linguists devised their own solutions.[1] For example, some chose to represent unavailable graphemes with substitutes, e.g. the combination of <ng> to represent <ŋ>. Tech-savvy linguists redefined selected characters from a character encoding by mapping custom made fonts to those code points. However, one linguist's electronic text would not render properly on another linguist's computer without access to the same font. Further, if two character encodings defined two character sets differently, then data could not be reliably and correctly displayed. This is a common example of the non-interoperability of data and data formats.

[1] Early work addressing the need for a universal computing environment for writing systems and their computational complexity is discussed in Simons (1989). A survey of practical recommendations for language resources, including notes on encoding, can be found in Bird & Simons (2003)

To alleviate this problem, during the late 1980s, SAMPA (Speech Assessment Methods Phonetic Alphabet) was created to represent IPA symbols with ASCII character sequences, e.g. <p\> for [ɸ]. Two problems with SAMPA are that (i) it is only a partial encoding of the IPA and (ii) it encodes different languages in separate data tables, instead of using a universal alphabet, like IPA. SAMPA tables are derived from phonemes appearing in several European languages that were developed as part of a European Commission-funded project to address technical problems like electronic mail exchange (what is now simply called email). SAMPA is essentially a hack to work around displaying IPA characters, but it provided speech technology and other fields a basis that has been widely adopted and used in code. So, SAMPA was a collection of tables to be compared, instead of a large universal table representing all languages. An extended version of SAMPA, called X-SAMPA, set out to include every symbol in the IPA chart including all diacritics (Wells n.d.). X-SAMPA was considered more universally applicable because it consisted of one table that encoded all characters that represented phones/segments in IPA across languages. SAMPA and X-SAMPA have been widely used for speech technology and as an encoding system in computational linguistics. Eventually, ASCII-encoding of the IPA became deprecated through the advent of the Unicode Standard. Note however that many popular software packages used for linguistic analyses still require ASCII input, e.g. RuG/L04 and SplitsTree4.[2]

There are a few pitfalls to be aware of when using the Unicode Standard to encode IPA. As we have said before, from a linguistic perspective it might sometimes look like the Unicode Consortium is making incomprehensible decisions, but it is important to realize that the consortium has tried and is continuing to try to be as consistent as possible across a wide range of use cases, and it does place linguistic traditions above other orthographic possibilities. In general, we strongly suggest linguists not to complain about any decisions in the Unicode Standard, but to try and understand the rationale of the Unicode Consortium (which in our experience is almost always well-conceived) and devise ways to work with any unexpected behavior. Many of the current problems derive from the fact that the IPA is clearly historically based on the Latin script, but different enough from most other Latin-based writing systems to warrant special attention. This ambivalent status of the IPA glyphs (partly Latin, partly special) is unfortunately also attested in the treatment of IPA in the Unicode Standard. In retrospect, it might have been better to consider the IPA (and other transcription systems) to be a special kind of script within the Unicode Standard, and treat

[2] See http://www.let.rug.nl/kleiweg/L04/ and http://www.splitstree.org/, respectively

the obvious similarity to Latin glyphs as a historical relic. All IPA glyphs would then have their own code points, instead of the current situation in which some IPA glyphs have special code points, while others are treated as being identical to the regular Latin characters. Yet, the current situation, however unfortunate, is unlikely to change, so as linguists we will have to learn to deal with the specific pitfalls of IPA within the Unicode Standard. In this section, we will describe these pitfalls in some detail.

3.2 Pitfall: No complete IPA code block

The ambivalent nature of IPA glyphs arises because, on the one hand, the IPA uses Latin-based glyphs like <a>, <b> or <p>. From this perspective, the IPA seems to be just another orthographic tradition using Latin characters, all of which do not get a special treatment within the Unicode Standard (just like e.g. the French, German, or Danish orthographic traditions do not have a special status). On the other hand, the IPA uses many special symbols (like turned <ɐ>, mirrored <ɘ> and/or extended <ɧ> Latin glyphs) not found in any other Latin-based writing system. For this reason a special block with code points, called IPA Extensions was included already in the first version of the Unicode Standard (Version 1.0 from 1991).

As explained in Section 2.5, the Unicode Standard code space is subdivided into character blocks, which generally encode characters from a single script. However, as is illustrated by the IPA, characters that form a single writing system may be dispersed across several different character blocks. With its diverse collection of symbols from various scripts and diacritics, the IPA is spread across 13 blocks in the Unicode Standard:[3]

- Basic Latin (30 characters), e.g. <a b c d e>
- Latin-1 Supplement (4 characters): <æ ç ð ø>
- Latin Extended-A (3 characters): <ħ ŋ œ>
- Latin Extended-B (5 characters): <ǀ ǁ ǂ ! ɳ>
- IPA Extensions (70 characters), e.g. <ɐ ɑ ɔ>
- Spacing Modifier Letters (20 characters), e.g. <ʰ ʷ ˥>
- Combining Diacritical Marks (33 characters), e.g. <◌̟ ◌̥ ◌̺ >
- Greek and Coptic (3 characters): <β θ χ>

[3] This number of blocks depends on whether only IPA-sanctioned symbols are counted or if the phonetic symbols commonly found in the literature are also included, see Moran 2012: Appendix C.

- Phonetic Extensions (2 characters): <ᴅ ᴴ>
- Phonetic Extensions Supplement (3 characters): <ᶑ ᶾ ᶣ>
- Superscripts and Subscripts (1 character): <ⁿ>
- Arrows (4 characters): <↑ ↓ ↗ ↘>
- Latin Extended-C (1 character): <ⱱ>

3.3 Pitfall: IPA homoglyphs in Unicode

Another problem is the large number of homoglyphs, i.e. different characters that have highly similar glyphs (or even completely identical glyphs, depending on the font rendering). For example, a user of a Cyrillic computer keyboard should ideally not use the <а> CYRILLIC SMALL LETTER A at code point U+0430 for IPA transcriptions, but instead use the <a> LATIN SMALL LETTER A at code point U+0061, although visually they are mostly indistinguishable, and the Cyrillic character is more easily typed on a Cyrillic keyboard.

Furthermore, even linguists are unlikely to distinguish between the <ə> LATIN SMALL LETTER SCHWA at code point U+0259 and <ǝ> LATIN SMALL LETTER TURNED E at U+01DD.

Conversely, non-linguists are unlikely to distinguish any semantic difference between an open back unrounded vowel <ɑ> LATIN SMALL LETTER ALPHA at U+0251, and the open front unrounded vowel <a> LATIN SMALL LETTER A at U+0061. But even among linguists this distinction leads to problems. For example, as pointed out by Mielke (2009), there is a problem stemming from the fact that about 75% of languages are reported to have a five-vowel system (Maddieson 1984). Historically, linguistic descriptions tend not to include precise audio recording and measurements of formants, so this may lead one to ask if the many <a> characters that are used in phonological description reflects a transcriptional bias. The common use of <a> in transcriptions could be in part due to the ease of typing the letter on an English keyboard (or for older descriptions, the typewriter). We found it to be exceedingly rare that a linguist uses <ɑ> for a low back unrounded vowel.[4] They simply use <a> as long as there is no opposition to <ɑ>.[5]

[4] One example is **Vidal2001a** in which the author states: "The definition of Pilagá /a/ as [+back] results from its behavior in certain phonological contexts. For instance, uvular and pharyngeal consonants only occur around /a/ and /o/. Hence, the characterization of /a/ and /o/ as a natural class of (i.e., [+back] vowels), as opposed to /i/ and /e/."

[5] See Thomason's Language Log post, "Why I don't love the International Phonetic Alphabet" at: http://itre.cis.upenn.edu/~myl/languagelog/archives/005287.html.

Making things even more problematic, there is an old typographic tradition that the double-story <a> uses a single-story <ɑ> in italics. This leads to the unfortunate effect that in most well-designed fonts the italics of <a> and <ɑ> use the same glyph. If this distinction has to be kept upright in italics, the only solution we can currently offer is to use SLANTED glyphs (i.e. artificially italicized glyphs) instead of real italics (i.e. special italics glyphs designed by a typographer).[6]

Some other homoglyphs related to encoding IPA in the Unicode Standard are:

- The uses of the apostrophe has led to long discussions on the Unicode Standard email list. An English keyboard inputs < ' > APOSTROPHE at U+0027, although the preferred Unicode apostrophe is the < ’ > RIGHT SINGLE QUOTATION MARK at U+2019.
- The glottal stop/glottalization/ejective marker is another completely different character <ʼ>, the MODIFIER LETTER APOSTROPHE at U+02BC, but unfortunately looks mostly highly similar to U+2019.
- Another problem is the <ˁ> MODIFIER LETTER REVERSED GLOTTAL STOP at U+02C1 vs. the <ˠ> MODIFIER LETTER SMALL REVERSED GLOTTAL STOP at U+02E4. Both appear in various resources representing phonetic data online. This is thus a clear example for which the Unicode Standard does not solve the linguistic standardization problem.
- There is at least one case in which the character name assigned by the Unicode Consortium does not match the IPA's description. In the Unicode Standard the <!> at U+01C3 is labeled LATIN LETTER RETROFLEX CLICK, but in IPA that glyph is used for an alveolar or postalveolar click (not retroflex). This naming is probably best seen as a simple error in the Unicode Standard. This character is of course often simply typed as <!> EXCLAMATION MARK at U+0021.

[6] For example, the widely used IPA font Doulos SIL (http://scripts.sil.org/cms/scripts/page.php?item_id=DoulosSIL) does not have real italics. This leads some word-processing software, like Microsoft Word, to produce slanted glyphs instead. That particular combination of font and software application will thus lead to the desired effect distinguishing <a> from <ɑ> in italics. However, note that when the text is transferred to another font (i.e. one that includes real italics) and/or to another software application (like Apple Pages, which does not perform slanting), then this visual appearance will be lost. In this case we are thus still in the pre-Unicode situation in which the choice of font and rendering software actually matters. The ideal solution from a linguistic point of view would be the introduction of a new IPA code point for a different kind of <a>, which explicitly specifies that it should still be rendered as a double-story character when italicized. After informal discussion with various Unicode players, our impression is that this highly restricted problem is not sufficiently urgent to introduce even more <a> homographs in Unicode (which already lead to much confusion, see Section 2.6).

3.4 Pitfall: Homoglyphs in IPA

It is not just the Unicode Standard that offers multiple options for encoding the IPA. Even the IPA specification itself offers some flexibility in how transcriptions have to be encoded. There are a few cases in which the IPA explicitly allows for different options of transcribing the same phonetic content. This is understandable from a transcriber's point of view, but it is not acceptable for interoperability between resources written in IPA. We consider it crucial to distinguish between "lax" IPA, for which it is sufficient that any phonetically-trained reader is able to understand the transcription, and "strict" IPA, which should be standardized on a single unique encoding for each sound, so search will work across resources. We are aware of the following double options in the IPA, which will be discussed in turn below:

- The marking of tone
- The marking of <g>
- The diacritic for voiceless
- The tie bar to indicate a close bond between sounds

The first case in which the IPA allows for different encodings is the question of how to transcribe tone. There is an old tradition to use diacritics on vowels to mark different tone levels, e.g. <ə̏ə̀ə́ə̋>.[7] The IPA also proposes a less widespread used, but more consistent option of tone letters, e.g. <˩˨˧˦˥>. Tone letters in the IPA have five different levels, and sequences of these letters can be used to indicate contours. Well-designed fonts will even merge a sequence of tone letters into a contour. For example, compare the font Linux Libertine, which does not merge tone letters <˩˧˥>, with the font CharisSIL, which merges this sequence of four tone letters into a single contour <˩˧˥>. For strict IPA encoding we propose to standardize on tone letters.

Second, we commonly encounter the use of <g> LATIN SMALL LETTER G at U+0067, instead of the Unicode Standard IPA character for the voiced velar stop <ɡ> LATIN SMALL LETTER SCRIPT G at U+0261. One begins to question whether this issue is at all apparent to the working linguist, or if they simply use the U+0067 because it is easily keyboarded and thus saves time, whereas the latter must be cumbersomely inserted as a special symbol in most software. This issue was recently addressed by The International Phonetic Association has taken

[7] There are at least two different Unicode homoglyphs for the low and high level tones, namely <̏> COMBINING GRAVE TONE MARK at U+0340 vs. <̀> COMBINING GRAVE ACCENT at U+0300 for low tone, and <́> COMBINING ACUTE TONE MARK at U+0341 vs. <́> COMBINING ACUTE ACCENT at U+0301 for high tone.

the stance that both the keyboard LATIN SMALL LETTER G and the LATIN SMALL LETTER SCRIPT G are valid input characters for the voiced velar plosive. Unfortunately, this decision further introduces ambiguity for linguists trying to adhere to a strict Unicode Standard IPA encoding. For strict IPA encoding we propose to standardize on the more idiosyncratic LATIN SMALL LETTER SCRIPT G at U+0261.

Third, for marking marking of voiceless pronunciation of voiced segments the IPA uses the ring diacritic. Originally, the ring should be placed below the base character, like in <m̥>, using the COMBINING RING BELOW at U+0325. However, in letters with long descenders the IPA also allows to put the ring above the base, like in <ŋ̊>, using the COMBINING RING ABOVE at U+030A. Yet, proper font design does not have any problem with rendering the ring below the base character, like in <ŋ̥>, so for strict IPA encoding we propose to standardize on the ring below.

Finally, the tie bar to indicate affricates, doubly articulated consonants or diphthongs can be placed either below or on top of the base characters, e.g. <t͡s> or <t͜s>. For strict IPA encoding we propose to standardize on the tie bar above the base characters, using Unicode COMBINING DOUBLE INVERTED BREVE at U+0361. Note that this Unicode character is placed between the two base characters to be combined. In principle, it is possible to combine more than two base characters by repeating the tie bar, like in <a͡o͡u>. If really necessary, we consider this possible, even though the rendering will never look good.

3.5 Pitfall: Ligatures and digraphs

One important distinction to acknowledge is the difference between multigraphs and ligatures. Multigraphs are groups of characters (in the context of IPA e.g. <tʃ> or <ou>) while ligatures are single characters (e.g. <ʧ> LATIN SMALL LETTER TESH DIGRAPH at U+02A7). Ligatures arose in the context of printing easier-to-read texts, and are included in the Unicode Standard for reasons of legacy encoding. However, their usage is discouraged by the Unicode core specification. Specifically related to IPA, various phonetic combinations of characters (typically affricates) are available as single code-points in the Unicode Standard, but are designated as LIGATURES or DIGRAPHS (confusingly both names appear interchangeably). Such glyphs might be used by software to produce a pleasing display, but they should not be hard-coded into the text itself. In the context of IPA, characters like the following ligatures should thus *not* be used. Instead a combination of two characters is preferred:

<ʣ> LATIN SMALL LETTER DZ DIGRAPH at U+02A3 (use <dz>)

<ʤ> LATIN SMALL LETTER DEZH DIGRAPH at U+02A4 (use <dʒ>)
<ʥ> LATIN SMALL LETTER DZ DIGRAPH WITH CURL at U+02A5 (use <dʑ>)
<ʦ> LATIN SMALL LETTER TS DIGRAPH at U+02A6 (use <ts>)
<ʧ> LATIN SMALL LETTER TESH DIGRAPH at U+02A7 (use <tʃ>)
<ʨ> LATIN SMALL LETTER TC DIGRAPH WITH CURL at U+02A8 (use <tɕ>)
<ʩ> LATIN SMALL LETTER FENG DIGRAPH at U+02A9 (use <fŋ>)

However, there are a few Unicode characters that are historically ligatures, but which are today considered as simple characters in the Unicode Standard and thus should be used when writing IPA, namely:

<ɮ> LATIN SMALL LETTER LEZH at U+026E
<œ> LATIN SMALL LIGATURE OE at U+0153
<ɶ> LATIN LETTER SMALL CAPITAL OE at U+0276
<æ> LATIN SMALL LETTER AE at U+00E6

3.6 Pitfall: Missing decomposition

Although many combinations of base character with diacritic are treated as canonical equivalent with precomposed characters, there are a few combinations in IPA that allow for multiple, apparently identical, encodings that are not canonical equivalent. The following elements should not be treated as diacritics when encoding IPA in Unicode:

<◌ʲ> COMBINING PALATALIZED HOOK BELOW at U+0321
<◌ᶭ> COMBINING RETROFLEX HOOK BELOW at U+0322
<◌-> COMBINING SHORT STROKE OVERLAY at U+0335
</> COMBINING SHORT SOLIDUS OVERLAY at U+0337
<◌~> COMBINING TILDE OVERLAY at U+0334
<ˠ> MODIFIER LETTER RHOTIC HOOK at U+02DE

There turn out to be a lot of characters in the IPA that could be conceived as using any of these elements, like <ɲ>, <ɳ>, <ɨ>, <ø>, <ɬ> or <ɑˠ>. However, all such characters exist as well as "precomposed" combination in Unicode, and these precomposed characters should preferably be used. When combinations of a base character with diacritic are used, then these combinations are not canonical equivalent to the precomposed combinations. This means that any search will not find both at the same time.

Reversely, <ç> LATIN SMALL LETTER C WITH CEDILLA at U+00E7 will be decomposed into <c> LATIN SMALL LETTER C at U+0063 with <◌̧> COMBINING CEDILLA at

U+0327, also if such a decomposition is not intended, because it is meaningless in IPA.

3.7 Pitfall: Different notions of diacritics

Another pitfall relates the concept of what are diacritics. The problem is that the meaning of the term diacritics as used by the IPA is not the same as is used in the Unicode Standard. Specifically, diacritics in the IPA-sense are either so-called COMBINING DIACRITICAL MARKS or SPACING MODIFIER LETTERS in the Unicode Standard. Crucially, Combining Diacritical Marks are by definition combined with the character before them (to form so-called default grapheme clusters, see Section 1.4). In contrast, Spacing Modifier Letters are by definition *not* combined into grapheme clusters with the preceding character, but simply treated as separate letters. In the context of the IPA, the following IPA-diacritics are actually Spacing Modifier Letters in the Unicode Standard:

Length marks, namely:

<ː> MODIFIER LETTER TRIANGULAR COLON at U+02D0

<ˑ> MODIFIER LETTER HALF TRIANGULAR COLON at U+02D1

Tone letters, like:

<˥> MODIFIER LETTER EXTRA-HIGH TONE BAR at U+02E5

<˧> MODIFIER LETTER MID TONE BAR at U+02E7

and others like this

Superscript letters, like:

<ʰ> MODIFIER LETTER SMALL H at U+02B0

<ˤ> MODIFIER LETTER SMALL REVERSED GLOTTAL STOP at U+02E4

and many more like this

The rhotic hook:

<˞> MODIFIER LETTER RHOTIC HOOK at U+02DE

Although linguists might expect these characters to belong together with the character in front of them, at least for <ʰ> MODIFIER LETTER SMALL H at U+02B0 the Unicode Consortium's decision to treat it as a separate character is also linguistically correct, because according to the IPA it can be used both for aspiration (more precisely post-aspiration following the base character) and pre-aspiration (preceding the base character). Note that there exists a mechanism in Unicode

to force separate characters to be combined (namely by using the ZERO WIDTH JOINER at U+200D), but this seems to be a rather impractical, and probably not an enforceable solution to us.

3.8 Pitfall: No unique diacritic ordering

Also related to diacritics is the question of ordering. To our knowledge, the International Phonetic Association does not specify a specific ordering for diacritics that combine with phonetic base symbols; this exercise is left to the reasoning of the transcriber. However, such marks have to be explicitly ordered if sequences of them are to be interoperable and compatible. An example is a labialized aspirated alveolar plosive: <t^{wh}>. There is nothing holding linguists back from using <t^{hw}> instead (with exactly the same intended meaning). However, from a technical standpoint, these two sequences are different, e.g. if both sequences are used in a document, searching for <t^{wh}> will not find any instances of <t^{hw}>, and vice versa. Likewise, a creaky voiced syllabic dental nasal can be encoded in various orders, e.g. <n̪̰̩>, <n̪̩̰> or <n̰̪̩>.

Canonical combining classes

In accordance with the absence of any specification of ordering in the IPA, the Unicode Standard likewise does not propose any standard orderings. Both leave it to the user to be consistent. However, there is one aspect of ordering for which the Unicode Standard does present a canonical solution, which is uncontroversial from a linguistic perspective. Diacritics in the Unicode Standard (i.e. Combining Diacritical Marks, see above) are classified in Canonical Combining Classes. In practice, the diacritics are distinguished by their position relative to the base character.[8] When applying a Unicode normalization (NFC or NFD, see Section 2.7), the diacritics in different positions are put in a specified order. This process therefore harmonizes the difference between different encodings, for example, of a midtone creaky voice vowel <ē̯>. This grapheme cluster can be encoded either as <e> + <◌̄> + <◌̯> or as <e> + <◌̯> + <◌̄>. To prevent this twofold encoding, the Unicode Standard specifies the second ordering as canonical (in this case, diacritics below are put before diacritics above).

When encoding a string according to the Unicode Standard, it is possible to do this either using the NFC (composition) or NFD (decomposition) normalization.

[8] See http://unicode.org/reports/tr44/#Canonical_Combining_Class_Values for a detailed description.

Decomposition implies that precomposed characters (like <á> LATIN SMALL LETTER A WITH ACUTE at U+00E1) will be split into its parts. This might sound preferable for a linguistic analysis, as the different diacritics are separated from the base characters. However, note that most attached elements like strokes (e.g. in the <ɨ>), retroflex hooks (e.g. in <ʐ>) or rhotic hooks (e.g. in <ɻ>) will not be decomposed, but strangely enough a cedilla (like in <ç>) will be decomposed (see Section 3.6). In general, Unicode decomposition does not behave like a feature decomposition as expected from a linguistic perspective. It is thus important to consider Unicode decomposition only as a technical procedure, and not assume that it is linguistically sensible.

Diacritic ordering

Facing the problem of specifying a consistent ordering of diacritics while developing a large database of phonological inventories from the world's languages, Moran (2012: 540) defines a set of diacritic ordering conventions.[9] The conventions are influenced by the linguistic literature, though some ad-hoc decisions had to be taken. The goal was to explicitly define all character sequences so that the vast variety of phonemes found in descriptions of the world's language were normalized into consistent character sequences, e.g. if one language description uses and another, when both are intended to be phonetically equivalent, then a decision to normalize to one form was taken. For example, when a character sequence contains more than one character in Spacing Modifier Letters, the order that is proposed is the following (where <|> indicates *or* and <⟶> indicates *precedes*):

SPACING MODIFIER LETTERS ORDERING:

⟶ (unreleased < ˺ > | lateral release <^{l}> | nasal release <^{n}>)
⟶ (palatalized <^{j}>)
⟶ (labialized <^{w}>)
⟶ (velarized <^{ɣ}>)
⟶ (pharyngealized <^{ʕ}>)
⟶ (aspirated <^{h}> | ejective <ʼ>)
⟶ (long <ː> | half-long <ˑ>)

If a character sequence contains more than one diacritic below the base character, then the place feature is applied first (dental, laminal, apical, fronted, backed,

[9] The most recent version of these conventions is online at http://phoible.github.io/conventions/

lowered, raised), then the laryngeal setting (voiced, voiceless, creaky voice, breathy voice) and finally the syllabic or non-syllabic marker (for vowels, ATR gets put on between the place and laryngeal setting). So:

Combining Diacritical Marks (below) ordering:

→ (dental <◌̪> | laminal <◌̻> | apical <◌̺>)
→ (fronted <◌̟> | backed <◌̠>)
→ (lowered <◌̞> | raised <◌̝>)
→ (ATR <◌̘ ◌̙>)
→ (voiced <◌̬> | voiceless <◌̥> | creaky voice <◌̰> | breathy voice <◌̤>)
→ (syllabic <◌̩> | non-syllabic <◌̯>)

Character sequences with diacritics above the base character were not problematic in Moran (2012) because they include only the centralized, mid-centralized and nasalized combining characters. Moran (2012) marks tones as singletons with Space Modifier Letters, e.g. <˥> for a phonemic high tone, instead of accent diacritics, alleviating potential conflicts. Building on the work of Moran (2012), if a character sequence contains more than one diacritic above the base character, we propose:

Combining Diacritical Marks (above) ordering:

→ (centralized <◌̈> | mid-centralized <◌̆>)
→ (extra short <◌̆>)
→ (tone accents, e.g. <◌̀>)
→ (Spacing Modifier Letters, see above)
→ (tone letters, e.g. <˥>)

3.9 Recommendations

Summary

4 Orthography profiles

4.1 Characterizing writing systems

At this point in the course of rapid ongoing developments, we are left with a situation in which the Unicode Standard offers a highly detailed and flexible approach to deal computationally with writing systems, but it has unfortunately not influenced the linguistic practice very much. In many practical situations, the Unicode Standard is far too complex for the day-to-day practice in linguistics because it does not offer practical solutions for the down-to-earth problems of many linguists. In this section, we propose some simple practical guidelines and methods to improve on this situation.

Our central aims for linguistics, to be approached with a Unicode-based solution, are: (i) to improve the consistency of the encoding of sources, (ii) to transparently document knowledge about the writing system (including transliteration), and (iii) to do all of that in a way that is easy and quick to manage for many different sources with many different writing systems. The central concept in our proposal is the ORTHOGRAPHY PROFILE, a simple tab-separated CSV text file, that characterizes and documents a writing system. We also offer basic implementations in Python and R to assist with the production of such files, and to apply orthography profiles for consistency testing, grapheme tokenization and transliteration. Not only can orthography profiles be helpful in the daily practice of linguistics, they also succinctly document the orthographic details of a specific source, and, as such, might fruitfully be published alongside sources (e.g. in digital archives). Also, in high-level linguistic analyzes in which the graphemic detail is of central importance (e.g. phonotactic or comparative-historical studies), orthography profiles can transparently document the decisions that have been taken in the interpretation of the orthography in the sources used.

Given these goals, Unicode locale descriptions (see Section **??**) might seem like the ideal orthography profiles. However, there are various practical obstacles preventing the use of such locale descriptions in the daily linguistic practice, namely: (i) the XML-structure is too verbose to easily and quickly produce or correct manually, (ii) locale descriptions are designed for a wide scope on information (like

date formats or names of weekdays) most of which is not applicable for documenting writing systems, and (iii) most crucially, even if someone made the effort to produce a technically correct locale description for a specific source at hand, then it is nigh impossible to deploy the description. This is because a locale description has to be submitted to and accepted by the Unicode Common Locale Data Repository. The repository is (rightly so) not interested in descriptions that only apply to a limited set of sources (e.g. only a single specific dictionary).

The major challenge then is developing an infrastructure to identify the elements that are individual graphemes in a source, specifically for the enormous variety of sources using some kind of alphabetic writing system. Authors of source documents (e.g. dictionaries, wordlists, corpora) use a variety of writing systems that range from their own idiosyncratic transcriptions to already well-established practical or longstanding orthographies. Although the IPA is one practical choice as a sound-based normalization for writing systems (which can act as an interlingual pivot to attain interoperability across writing systems), graphemes in each writing system must also be identified and standardized if interoperability across different sources is to be achieved. In most cases, this amounts to more than simply mapping a grapheme to an IPA segment because graphemes must first be identified in context (e.g. is the sequence one sound or two sounds or both?) and strings must be tokenized, which may include taking orthographic rules into account (e.g. between vowels is /n/ and after a vowel but before a consonant is a nasalized vowel /ṽ/). In our experience, data from each source must be individually tokenized into graphemes so that its orthographic structure is identified and its contents can be extracted. To extract data for analysis, a source-by-source approach is required before an orthography profile can be created. For example, almost each available lexicon on the world's languages is idiosyncratic in its orthography and thus requires lexicon-specific approaches to identify graphemes in the writing system and to map graphemes to phonemes, if desired.

Thus, our key proposal for the characterization of a writing system is to use a grapheme tokenization as an inter-orthographic pivot. Basically, any source document is tokenized by graphemes, and only then a mapping to IPA (or any other orthographic conversion) is performed. An orthography profile then is a description of the units and rules that are needed to adequately model a graphemic tokenization for a language variety as described in a particular source document. An orthography profile summarizes the Unicode (tailored) graphemes and orthographic rules used to write a language (the details of the structure and assumptions of such a profile will be presented in the next section).

As an example of graphemic tokenization, note the three different levels of technological and linguistic elements that interact in the hypothetical lexical form <ts^{h}õ̰shi>:

1. code points (10 text elements): t s ^{h} õ̰ ◌́ s h i
2. grapheme clusters (7 text elements): t s ^{h} õ̰ s h i
3. tailored grapheme clusters (4 text elements): ts^{h} õ̰ sh i

In (1), the string <ts^{h}õ̰shi> has been tokenized into ten Unicode code points (using NFD normalization), delimited here by space. Unicode tokenization is required because sequences of code points can differ in their visual and logical orders. For example, <õ̰> is ambiguous to whether it is the sequence of + ◌̰ + ◌̃> or + ◌̃ + ◌̰>. Although these two variants are visually homoglyphs, computationally they are different. Unicode normalization should be applied to this string to reorder the code points into a canonical order, allowing the data to be treated canonically equivalently for search and comparison. In (2), the Unicode code points have been logically normalized and visually organized into grapheme clusters, as specified by the Unicode Standard. The combining character sequence <õ̰> is normalized and visually grouped together. Note that, the MODIFIER LETTER SMALL H at U+02B0, is not grouped with. This is because it belongs to Spacing Modifier Letters category in the Unicode Standard. These characters are underspecified for the direction in which they modify a host character. For example, can indicate either pre- or post-aspiration (whereas the nasalization or creaky diacritic is defined in the Unicode Standard to apply to a specified base character). Finally, to arrive at the graphemic tokenization in (3), tailored grapheme clusters are needed (as for example specified in an orthography profile). For example, this orthography profile would specify that the sequence of characters, and form a single grapheme, and that and form a grapheme. The orthography profile could also specify orthographic rules, e.g. when tokenization graphemes, in say English, the in the forms and should be treated as distinct sequences depending on their contexts.

4.2 Informal description

An orthography profile describes the Unicode code points, characters, graphemes and orthographic rules in a writing system. An orthography profile is a language-specific (and often even resource-specific) description of the units and rules that are needed to adequately model a writing system. An important assumption of our work is that we assume a resource is encoded in Unicode (or

has been converted to Unicode). Any data source that the Unicode Standard is unable to capture, will also not be captured by an orthography profile.

Informally, an orthography profile specifies the graphemes (or, in Unicode parlance, TAILORED GRAPHEME CLUSTERS) that are expected to occur in any data to be analyzed or checked for consistency. These graphemes are first identified throughout the whole data (a step which we call TOKENIZATION), and possibly simply returned as such, possibly including error messages about any parts of the data that are not specified by the orthography profile. Once the graphemes are identified, they might also be changed into other graphemes (a step which we call TRANSLITERATION). When a grapheme has different possible transliterations, then these differences should be separated by contextual specification, possibly down to listing individual exceptional cases.

In practice, we foresee a workflow in which orthography profiles are iteratively refined, while at the same time inconsistencies and errors in the data to be tokenized are corrected. In some more complex use-cases there might even be a need for multiple different orthography profiles to be applied in sequence (see Section 5 on various exemplary use-cases). The result of any such workflow will normally be a cleaned dataset and an explicit description of the orthographic structure in the form of an orthography profile. Subsequently, the orthography profiles can be easily distributed in scholarly channels alongside the cleaned data, for example in supplementary material added to journal papers or in electronic archives.

4.3 Formal specification

The formal specifications of an orthography profile (or simply PROFILE for short) are the following:

1. A PROFILE IS A UNICODE UTF-8 ENCODED TEXT FILE (ideally using NFC, no-BOM, and LF; see Section 2.9, Pitfall: File Formats) that includes the information pertinent to the orthography.
2. A PROFILE IS A TAB-SEPARATED CSV FILE WITH AN OBLIGATORY HEADER LINE. A minimal profile can have just a single column, in which case there will of course no tabs, but the first line will still be the header. For all columns we assume the name in the header of the CSV file to be crucial. The actual ordering of the columns is unimportant.
3. LINES STARTING WITH A HASH <#> ARE IGNORED. Comments and metadata can be included inside the file, but only as complete lines in the profile, to

be marked by lines starting with hash # (NUMBER SIGN at U+0023). Hashes somewhere else in the file are to be treated literally, i.e. hashes are only to be ignored when they occur at the start of a line.[1]

4. METADATA ARE GIVEN IN COMMENTED LINES AT THE BEGINNING OF THE TEXT FILE IN A BASIC TAG: VALUE FORMAT. Metadata about the orthographic description given in the orthography profile includes, minimally, (i) author, (ii) date, (iii) title, (iv) a stable language identifier encoded in BCP 47/ISO 639-3, and (v) bibliographic data for resource(s) that illustrate the orthography described in the profile.

The content of a profile consists of lines, each describing a grapheme of the orthography, using the following columns:

1. A MINIMAL PROFILE CONSISTS OF A SINGLE COLUMN WITH HEADER GRAPHEMES, listing each of the different graphemes in a separate line.
2. OPTIONAL COLUMNS CALLED LEFT AND RIGHT CAN BE USED TO SPECIFY THE LEFT AND RIGHT CONTEXT OF THE GRAPHEME, RESPECTIVELY. The same grapheme can occur multiple times with different contextual specifications, for example to distinguish different pronunciations depending on the context.
3. THE COLUMNS GRAPHEME, LEFT AND RIGHT CAN USE REGULAR EXPRESSION METACHARACTERS. If regular expressions are used, then all literal usage of the special symbols, like full stops <.> or dollar signs <$> (so-called METACHARACTERS) have to be explicitly escaped by adding a backslash before them (i.e. use <.> or <$>). Note that any specification of context automatically expects regular expressions, so it is probably better to always escape all regular expression metacharacters when used literally in the orthography, i.e. the following symbols will need to be preceded by a backslash: [ ] ( ) { } + * . - ! ? ^ $.
4. AN OPTIONAL COLUMN CALLED CLASS CAN BE USED TO SPECIFY CLASSES OF GRAPHEMES, for example to define a class of vowels. Users can simply add ad-hoc identifiers in this column to indicate a group of graphemes, which can then be used in the description of the graphemes or the context. The identifiers should of course be chosen such that they do not conflate with

[1] Comments that belong to specific lines will have to be put in a separate column of the CSV file, e.g. add a column called COMMENTS. Further, if the content of a profile contains a hash at the start of a line, either reorder the columns so the hash does not occur at the start of the line, or add a dummy column in front of the data to not have the data start with a hash.

any symbols used in the orthography themselves. Note that such classes only refer to the graphemes, not to the context.

5. Columns describing transliterations for each graphemes can be added and named at will. Often more than a single possible transliteration will be of interest. Any software application using these profiles should use the names of these columns to select a specific transliteration column.
6. Any other columns can be added freely, but will mostly be ignored by any software application using the profiles. As orthography profiles are also intended to be read and interpreted by humans, it is often highly useful to add extra information on the graphemes in further columns, like for example Unicode codepoints, Unicode names, frequency of occurrence, examples of occurrence, explanation of the contextual restrictions, or comments.

For the automatic processing of the profiles, the following technical standards will be expected:

1. Each line of a profile will be interpreted as a regular expression. Software applications using profiles can also offer to interpret a profile in the literal sense to avoid the necessity for the user to escape regular expressions metacharacters in the profile. However, this only is possible when no contexts or classes are described, so this seems only useful in the most basic orthographies.
2. The class column will be used to produce explicit or chains of regular expressions, which will then be inserted in the graphemes, left and right columns at the position indicated by the class-identifiers. For example, a class V as a context specification might be replaced by a regular expression like: (a|e|i|o|u|ei|au). Only the graphemes themselves are included here, not any contexts specified for the elements of the class.
3. The left and right contexts will be included into the regular expressions by using lookbehind and lookahead. Basically, the actual regular expression syntax of lookbehind and lookahead is simply hidden to the users by allowing them to only specify the contexts themselves. Internally, the contexts in the columns left and right are combined with the column graphemes to form a complex regular expression like: (?<=left)graphemes(?=rig
4. The regular expressions will be applied in the order as specified in the profile, from top to bottom. A software implementation can offer

help in figuring out the optimal ordering of the regular expressions, but should then explicitly report on the order used.

The actual implementation of the profile on some text-string will function as follows:

1. ALL GRAPHEMES ARE MATCHED IN THE TEXT BEFORE THEY ARE TOKENIZED OR TRANSLITERATED. In this way, there is no necessity for the user to consider 'feeding' and 'bleeding' situations, in which the application of a rule either changes the text so another rule suddenly applies (feeding) or prevents another rule to apply ('bleeding').
2. THE MATCHING OF THE GRAPHEMES CAN OCCUR EITHER GLOBALLY OR LINEARLY. From a computer science perspective, the most natural way to match graphemes from a profile in some text is by walking linearly through the text-string from left to right, and at each position go through all graphemes in the profile to see which one matches, then go to the position at the end of the matched grapheme and start over. This is basically how a finite state transducer works, which is a well-established technique in computer science. However, from a linguistic point of view, our experience is that most linguists find it more natural to think from a global perspective. In this approach, the first grapheme in the profile is matched everywhere in the text-string first, before moving to the next grapheme in the profile. Theoretically, these approaches will lead to different results, though in practice of actual natural language orthographies they almost always lead to the same result. Still, we suggest that any software application using orthography profiles should offer both approaches (i.e. GLOBAL or LINEAR) to the user. The approach used should be documented in the metadata as TOKENIZATION METHOD.
3. THE MATCHING OF THE GRAPHEMES CAN OCCUR EITHER IN NFC OR NFD. By default, both the profile and the text-string to be tokenized should be treated as NFC (see section 2.7, Pitfall: Canonical equivalence, above). However, in some use-cases it turns out to be practical to treat both text and profile as NFD. This typically happens when very many different combinations of diacritics occur in the data. An NFD-profile can then be used to first check which individual diacritics are used, before turning to the more cumbersome inspection of all combinations. We suggest that any software application using orthography profiles should offer both approaches (i.e. NFC or NFD) to the user. The approach used should be documented in the metadata as UNICODE NORMALIZATION.

4. The text-string is always returned in tokenized form by separating the matched graphemes by a user-specified symbols-string. Any transliteration will be returned on top of the tokenization.
5. Leftover characters (i.e. characters that are not matched by the profile) should be reported to the user as errors. Typically, the unmatched character are replaced in the tokenization by a user-specified symbol-string.

Any software application offering to use orthography profile:

1. should offer user-options to specify:
    1. the name of the column to be used for transliteration (if any).
    2. the symbol-string to be inserted between graphemes. Optionally, a warning might be given if the chosen string includes characters from the orthography itself.
    3. the symbol-string to be inserted for unmatched strings in the tokenized and transliterated output.
    4. the tokenization method, i.e. whether the tokenization should proceed global or linear.
    5. unicode normalization, i.e. whether the text-string and profile should use NFC or NFD.
2. might offer user-options to:
    6. assist in the ordering of the graphemes. In our experience, it makes sense to apply larger graphemes before shorter graphemes, and to apply graphemes with context before graphemes without context. Further, frequently relevant rules might be applied after rarely relevant rules (though frequency is difficult to establish in practice, as it depends on the available data). Also, if this all fails to give any decisive ordering between rules, it seems useful to offer linguists the option to reverse the ordering from any manual specified ordering, because linguists tend to write the more general rule first, before turning to exceptions or special cases.
    7. assist in dealing with upper and lower case characters. It seems practical to offer some basic case matching, so characters like <a> and <A> are treated equally. This will be useful in many concrete cases, although the user should be warned that case matching does not function universally in the same way across orthographies. Ideally, users should prepare orthography profiles with all lowercase

and uppercase variants explicitly mentioned, so by default no case matching should be performed.

8. TREAT THE PROFILE LITERAL, i.e. to not interpret regular expression metacharacters. Matching graphemes literally often leads to strong speed increase, and would allow users to not needing to worry about escaping metacharacters. However, in our experience all actually interesting use-cases of orthography profiles include some contexts, which automatically prevents any literal interpretation, so by default the matching should not be literal.

3. SHOULD RETURN THE FOLLOWING INFORMATION to the user:

9. THE ORIGINAL TEXT-STRINGS TO BE PROCESSED IN THE USED UNICODE NORMALIZATION, i.e. in either NFC or NFD as specified by the user.
10. THE TOKENIZED STRINGS, with additionally any transliterated strings, if transliteration is requested.
11. A SURVEY OF ALL ERRORS ENCOUNTERED, ideally both in which text-strings any errors occurred and which characters in the text-strings lead to errors.
12. A REORDERED PROFILE, when any automatic reordering is offered

5 Use cases

5.1 Introduction

We now present several use cases that have motivated the development of orthography profiles. These include:

- tokenization and error checking
- normalization of orthographic systems to attain interoperability across different source documents,
- cognate identification for detecting the similarity between words from different languages

An important assumption of our work is that input sources are encoded in Unicode (UTF-8 to be precise). Anything that the Unicode Standard is unable to capture, cannot be captured by an orthography profile. We also use Unicode normalization form NFD to decompose all incoming text input into normalized and (Unicode) logically ordered strings. This process is of fundamental importance when working with text data in the Unicode Standard.

5.2 Tokenization and error checking

The most basic use case is tokenization of language data. Tokenization comes in three types:

- Unicode character tokenization - grapheme tokenization - tailored grapheme tokenization

The simplest form of tokenization provided by any Unicode Standard-compliant software will split an input text stream on Unicode character code points. The input text is split on the character bit sequences as they have been encoded by the user. This may be in various Unicode Normalization Forms (see Section **??**) and tokenization split function returns a byte stream sequence given a particular encoding (see Section **??**).

There is increasing support for the regular expression match, commonly “ X”, to identify Unicode graphemes (see Section **??**) and perform tokenization on these sets of Unicode characters. Examples are given in Section **??**.

The orthography profile provides the mechanism for tokenizing an input stream on tailored grapheme clusters. The orthography's formal specification is given in Section ??. To sum, an orthography profile is a list of tailored grapheme clusters and / or orthographic rules that specify how a specific resource of text input should be tokenized.

Once text has been normalized, an additional and straightforward process is Unicode grapheme tokenization. More recently this regular expression, often available as short cut "\X", identifies all sequences of "base" characters followed by 0 or more combining diacritics. For example, the sequence, a :>. For a first pass at identifying grapheme clusters, this is a straightforward tokenization. However, the linguist will note that both the tie bar (COMBINING DOUBLE INVERTED BREVE, a Unicode Combining Modifier) and the length marker (MODIFIER LETTER TRIANGULAR COLON. a Unicode Letter Modifier) do not appear "correctly", i.e. the tie bar is grouped with the first character in the sequence and the length marker appears singly. This is necessary, as defined by the Unicode Standard, because of these character's semantic properties – with Unicode we cannot know if, for example, the MODIFIER LETTER TRIANGULAR COLON should appear before or after the base character that it modifies, cf. the IPA aspiration marker, , the MODIFIER LETTER SMALL H, which is also a Unicode Letter Modifier and for linguists a diacritic that be be used for pre- or post-aspiration. These ambiguities of position in tokenization are not uncommon in IPA, thus orthographic (or source) normalization and tokenization is needed.

5.3 Cross-orthographic analysis of writing systems

Given that orthography profiles are stored in a standard CSV format, we can use tools for converting and working with CSV. One such tool is the command line utility CSVKIT.[1]

[1] https://csvkit.readthedocs.org

Bibliography

Beider, Alexander & Stephen P. Morse. 2008. Beider-morse phonetic matching: an alternative to soundex with fewer false hits. *Avotaynu: the International Review of Jewish Genealogy* 24(2). 12. https://web.archive.org/web/20150513075152/http://stevemorse.org/phonetics/bmpm.htm.

Belongie, Serge, Jitendra Malik & Jan Puzicha. 2002. Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24(4). 509–522.

Bird, Steven & Gary F. Simons. 2003. Seven Dimensions of Portability for Language Documentation and Description. *Language* 79(3). 557–582. http://www.language-archives.org/documents/portability.pdf.

Brown, Cecil H., Eric W. Holman & Søren Wichmann. 2013. Sound correspondences in the world's languages. *Language* 89(1). 4–29.

Daniels, Peter T. 1990. Fundamentals of Grammatology. *Journal of the American Oriental Society* 110(4). 727–731.

Daniels, Peter T. 1996. The Study of Writing Systems. In P. T. Daniels & W. Bright (eds.), *The world's writing systems*. New York, NY: Oxford University Press.

Daniels, Peter T. & William Bright. 1996. *The World's Writing Systems*. New York, NY: Oxford University Press.

Dolgopolsky, A. B. 1986. A probabilistic hypothesis concerning the oldest relationships among the language families of northern eurasia. In Vitalij V. Shevoroshkin (ed. and trans.). Trans. from the Russian by Vitalij V. Shevoroshkin, *Typology, relationship and time: A collection of papers on language change and relationship by soviet linguists*, 27–50. Ann Arbor: Karoma Publisher.

Gaultney, J. Victor. 2002. *Problems of diacritic design for latin script text faces*. SIL International.

Huurdeman, Anton A. 2003. *The worldwide history of telecommunications*. New York, NY: John Wiley & Sons.

International Phonetic Association. 2005. *International Phonetic Alphabet*. Tech. rep. International Phonetic Association. http://www.arts.gla.ac.uk/IPA/.

Kemp, Alan. 2006. Phonetic transcription: history. *The Encyclopedia of Language and Linguistics* 6. 396–410.

Knuth, Donald Ervin. 1973. *The art of programming: sorting and searching*. Vol. 3. Reading, Mass.: Addison-Wesley.

Kohrt, Manfred. 1986. The Term 'Grapheme' in the History and Theory of Linguistics. In Gerhard Augst (ed.), *New trends in graphemics and orthography*, 80–96. Berlin: de Gruyter.

List, Johann-Mattis. 2012a. Multiple sequence alignment in historical linguistics: A sound class based approach. In *Proceedings of console xix*, 241–260.

List, Johann-Mattis. 2012b. SCA. phonetic alignment based on sound classes. In Marija Slavkovik & Dan Lassiter (eds.), *New directions in logic, language, and computation* (LNCS 7415), 32–51. Berlin & Heidelberg: Springer.

List, Johann-Mattis. 2012c. SCA: phonetic alignment based on sound classes. In Daniel Lassiter & Marija Slavkovik (eds.), *New directions in logic, language, and computation*, 32–51. Berlin & Heidelberg: Springer.

List, Johann-Mattis & Steven Moran. 2013. An open source toolkit for quantitative historical linguistics. In *Proceedings of the 51st annual meeting of the association for computational linguistics*, 13–18.

Maddieson, Ian. 1984. *Pattern of Sounds*. Cambridge, UK: Cambridge University Press.

Mania, Hubert. 2008. *Gauss: eine biographie*. Reinbek bei Hamburg: Rowohlt.

Meinhof, Carl & Daniel Jones. 1928. Principles of Practical Orthography for African Languages. *Africa: Journal of the International African Institute* 1(2). 228–239.

Meyer, Julien. 2015. *Whistled languages: a worldwide inquiry on human whistled speech*. Berlin: Springer. http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=957780.

Meyer, Julien, Laure Dentel & Frank Seifart. 2012. A methodology for the study of rhythm in drummed forms of languages: application to bora manguaré of amazon. In *Proceedings of interspeech*, 686–690.

Mielke, Jeff. 2009. Segment Inventories. *Language and Linguistics Compass* 3/2. 700–718.

Moran, Steven. 2006. *A grammatical sketch of western sisaala*. Ypsilanit, MI: Eastern Michigan University MA thesis.

Moran, Steven. 2012. *Phonetics Information Base and Lexicon*. University of Washington PhD thesis.

Moran, Steven & Jelena Prokić. 2013. Investigating the relatedness of the endangered dogon languages. *Literary And Linguistic Computing* 28(4). 676–691. http://llc.oxfordjournals.org/content/28/4/676.

Olúmúyìw, Tèmítọ́pẹ́. 2013. Yoruba writing: standards and trends. *Journal of Arts and Humanities* 2(1). 40–51. http://www.theartsjournal.org/index.php/site/article/view/50.

Postel, Hans Joachim. 1969. Die kölner phonetik: ein verfahren zur identifizierung von personennamen auf der grundlage der gestaltanalyse. *IBM-Nachrichten* 19. 925–931.

Powell, Barry B. 2012. *Writing theory and history of the technology of civilization*. New York, NY: John Wiley & Sons. http://nbn-resolving.de/urn:nbn:de:101:1-201502112957.

Robinson, Andrew. 1995. *The story of writing*. London: Thames & Hudson.

Simons, Gary F. 1989. Working With Special Characters. In *Occasional publications in academic computing* (Occasional Publications in Academic Computing). Dallas, TX: Summer Institute of Linguistics.

Singh, Simon. 1999. *The code book*. New York: Doubleday.

Sproat, Richard. 2000. *A Computational Theory of Writing Systems*. Cambridge, UK: Cambridge University Press.

Steiner, Lydia, Peter F Stadler & Michael Cysouw. 2011. A pipeline for computational historical linguistics. *Language Dynamics and Change* 1(1). 89–127.

Unicode Consortium, The. 2014. *The Unicode Standard, Version 7.0.0*. Tech. rep. Mountain View, CA: The Unicode Consortium.

Wells, John C. n.d. Computer-coding the IPA: A Proposed Extension of SAMPA. Unpublished Manuscript.