1 ODD SUBSET

1.1 Languages and Character Sets

The documents which users of these Guidelines may wish to encode encompass all kinds of material, potentially expressed in the full range of written and spoken human languages, including the extinct, the non-existent, and the conjectural. Because of this wide scope, special attention has been paid to two particular aspects of the representation of linguistic information often taken for granted: language identification, and character encoding.

Even within a single document, material in many different languages may be encountered. Human culture, and the texts which embody it, is intrinsically multilingual, and shows no sign of ceasing to be so. Traditional philologists and modern computational linguists alike work in a polyglot world, in which code-switching (in the linguistic sense) and accurate representation of differing language systems constitute the norm, not the exception. The current increased interest in studies of linguistic diversity, most notably in the recording and documentation of endangered languages, is one aspect of this long standing tradition. Because of their historical importance, the needs of endangered and even extinct languages must be taken into account when formulating Guidelines and recommendations such as these.

Beyond the sheer number and diversity of human languages, it should be remembered that in their written forms they may deploy a huge variety of scripts or writing systems. These scripts are in turn composed of smaller units, which for simplicity we term here characters. A primary goal when encoding a text should be to capture enough information for subsequent users of it correctly to identify both language, script, and constituent characters. In this chapter we address this requirement, and propose recommended mechanisms to indicate the languages, scripts and characters used in a document or a part thereof.

Identification of language is dealt with in 1.1.1. Language identification. In summary, it recommends the use of pre-defined identifiers for a language where these are available, as they increasingly are, in part as a result of the twin pressures of an increasing demand for language-specific software and an increased interest in language documentation. Where such identifiers are not available or not standardized, these Guidelines recommend a way of documenting language identifiers and their significance, in the same way as other metadata is documented in the TEI Header.

Standardization of the means available to represent characters and scripts has moved on considerably since the publication of the first version of these Guidelines. At that time, it was essential to explicitly document the characters and encoded character sets used by almost any digital resource if it was to have any chance of being usable across different computer platforms or environments, but this is no longer the case. With the availability of the Unicode standard, almost 100,000 different characters representing almost all of the world's current writing systems are available and usable in any XML processing environment without formality. Nevertheless, however large the number of standardized characters, there will always be a need to encode documents which use non-standard characters and glyphs, particularly but not exclusively in historical material. Furthermore, the full potential of Unicode is still not yet realised in all software which users of the Guidelines are likely to encounter. The second part of this chapter therefore discusses in some detail the concepts and practice underlying this standard, and also introduces the methods available for extending beyond it, which are more fully discussed in chapter «CEW06».

1.1.1 Language identification

Identification of the language a document or part thereof is written in is a crucial requirement for many envisioned usages of an electronic document. The TEI therefore accomodates this need in the following way:

- A global attribute `lang` is defined for all TEI elements. Its value identifies the language used.
- The TEI Header has a section set aside for the information about the languages used in a document, for details see «5.4.2 Language Usage».

The value of the attribute `lang` identifies the language using a coded value. For maximal compatibility with existing processes, modelling this value in the following way is recommended (this parallels the modelling of xml:lang):

- The identifier for the language should be constructed as in RTF 3066 or its successor. This *same* identifier has to be used to identify the `<language>` element in the TEI header.

The current draft of Tags for Identifying Languages proposes the following mechanism for constructing an identifier (tag) for languages as administered by the Internet Assigned Numbers Authority (IANA) by assembling this tag from a sequence of subtags separated by the hyphen (-, U+002D) character. It gives the language (possibly further identified with a sublanguage), a script and a region for this language, each possibly followed by a variant subtag.

- The identifier consists of at least one 'primary' subtag, it maybe followed by one or more 'extended' subtags.
- Languages are identified by a language subtag, which may be a two letter code taken from ISO 639-1 or a three letter code taken from ISO 639-2.
- ISO 639-2 reserves for private use codes the range 'qaa' through 'qtz'. These codes should be used for non-registered language subtags.
- A single letter primary subtag "x" indicates that the whole language tag is privately used.
- Extended language subtags must begin with the letter "s". They must follow the primary subtag and precede subtags that do define other properties of the language. The order is significant.
- 4 character subtags are interpreted as script identifiers taken from ISO 15924
- Region subtags can be either two letter country codes taken from ISO 3166 (with exceptions) or 3 digit codes from the UN Standard Country Codes for Statistical Use.
- Variant subtags may follow any of the above, but must precede private use extensions.
- Private use extensions are separated from the other subtags by the single letter subtag "x", which must be followed by at least one subtag. They might consist of several subtags separated with "-", but may not exceed a length of 32 characters.

- - de (German)
  - ja (Japanese)
  - zh (Chinese)
- - zh-Hant (Traditional Chinese)
  - en-Latn (English written in Latin script)
  - sr-Cyrl (Serbian written with Cyrillic script)
- - zh-Hans-CN (Simplified Chinese for the PRC)
  - sr-Latn-891 (Serbian, Latin script, Serbia and Montenegro)
- - zh-SG (Chinese for Singapore)
  - de-DE (German for Germany)
- - zh-CN (Chinese in China, no script given)
  - zh-Latn (Chinese transcribed in the Latin script)
- - de-CH-x-phonebook (phonebook collation for Swiss German)
  - zh-s-min (Min sub-language of Chinese)
  - zh-s-min-s-nan-Hant-CN (Southern variant of Min sublanguage as used in China, written with traditional Characters)
  - zh-Latn-x-pinyin (Chinese transcribed in the Latin script using the Pinyin system)

It should be noted that capitalization given here follows established convention (e.g. capital letters for country coded, small letters for language codes), but RTF 3066 does not ascribed any meaning to differences in capitalization.

As can be seen, both RTF 3066 and ISO 639-2 provide extensions that can be employed by private convention. The constructs mentioned above can thus be used to generate identifiers for any language, past and present, in any used in any area of the World. If such private extensions are used within the context of the TEI, they should be documented within the `<language>` element of the TEI header, which might also provide a prose description of the language described by the language tag.

While language, region and script can be adequately identified using this mechanism, there is only very rough provision to express a dimension of time for the language of a document; those codes provided (e.g. "grc" for "Greek, Ancient (to 1453)" in ISO 639-2) might not reflect the segments appropriate for a text at hand. Text encoders might express the time window of the language used in the document by means of the extension mechanism defined in RTF 3066 and relate that to a `<date>` or `<dateRange>` in the corresponding `<language>` sectio of the TEI header.

Equivalences to language identifiers by other authorities can be given in the `<language>` section as well, but no formal mechanism for doing so has been defined.

The scope of the language identification is extending to the whole subtree of the document anchored at the element that carries the `lang` attribute, including all elements and all attributes where a language might apply.This will exclude all attributes where a non-textual data type has been specified, for example tokens, boolean values or predefined value lists.References

Phillips, Addison.Davis, Mark, Tags for Identifying Languages2004-04-08, Internet Draft, proposed revision for RTF3066 `http://xml.coverpages.org/draft-phillips-langtags-02a.txt`

Cover, Robin: Language Identifiers in the Markup Context`http://xml.coverpages.org/languageIdentifiers.html`

Tim Bray Jean Paoli C. M. Sperberg-McQueen Eve Maler - Second Edition Francois Yergeau - Third Edition: Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation 04 February 2004 `http://www.w3.org/TR/2004/REC-xml-20040204/`

1.1.2 Characters and Character Sets

All document encoding has to do with representing one thing by another in an agreed and systematic way. Applied to the smallest distinctive units in any given writing system, which for the moment we may loosely call ‘characters’, such representation raises surprisingly complex and troublesome issues. The reasons are partly historical and partly to do with conceptual unclarities about what is involved in identifying, encoding, processing and rendering the characters of a natural language.

Historical considerations When the first methods of representing text for storage or transmission by machines were devised, long before the development of computers, the overriding aim was to identify the smallest set of symbols needed to convey the essential semantic content, and to encode that symbol set in the most economical way that the storage or transmission media allowed. The initial outcome were systems that encoded only such content as could be expressed in upper case letters in the Latin script, plus a few punctuation marks and some ‘control characters’ needed to regulate the storage and transmission devices. Such encodings, originally developed for telegraphy, strongly influenced the way the pioneers of computing conceived of and implemented the handling of text, with consequences that are with us still.

For many years after the invention of computers, the way they represented text continued to be constrained by the imperative to use expensive resources with maximal efficiency. Even when storage and processing costs began their dramatic fall, the Anglo-centric outlook of most hardware designers and software engineers hampered initiatives to devise a more generous and flexible model for text representation. The wish to retain compatability with ‘legacy’ data was an additional disincentive. Eventually, tension in East Asia between commitment to technological progress and the inability of existing computers to cope with local writing systems led to decisive developments. Japanese, Korean and Chinese standards bodies, who long before the advent of computers had been engaged in the specification of character sets, joined with computer manufacturers and software houses to devise ways of mapping those character sets to numeric encodings and processing the resulting text data.

Unfortunately, in the early years there was little or no co-ordination among either the national standards bodies or the manufacturers concerned, so that although commercial necessity dictated that these various local standards were all compatible with the representation of US-American English, they were not straightforwardly compatible with one another. Even within Japan itself there emerged a number of mutually incompatible systems, thanks to a mixture of commercial rivalry, disagreements about how best to manage certain intractable problems, and the fact that such pioneering work inevitably involved some false starts, leading to incompatibilities even between successive products of the same bodies. Roughly at the same time, and for similar reasons, multiple and incompatible ways of representing languages that use Cyrillic scripts were devised, along with methods of encoding ancient writing systems which inevitably could not aim for compatibility with other writing systems apart from basic Latin script. Many of the earliest projects that fed into the TEI were shaped in this developmental phase of the computerised representation of texts, and it was also the context in which SGML was devised and finalized.

SGML had of necessity to offer ways of coping with multiple writing systems in multiple representations; or rather, it provided a framework within which SGML-compliant applications capable of handling such multiple representations might be developed by those with sufficient financial and personnel resources (such as are seldom found in academia). Earlier editions of these Guidelines offered advice on character set and writing system issues addressed to the condition of those for whom SGML was the only feasible option. That advice must now be substantially altered because of two closely-related developments: the availability of the ISO/Unicode character set as an international standard, and the emergence of XML and related technologies which are committed to the theory and practice of character representation which Unicode embodies.

Terminology and key concepts Before the significance of Unicode and the implications of the association between XML and Unicode can be adequately explained, it is necessary to clarify some key concepts and attempt to establish an adequately precise terminology for them.

The word 'character' will not of itself take us very far towards greater terminological precision. It tends to be used to refer indiscriminately both to the visible symbol on a page and to the letter or ideograph which that symbol represents, two things that it is essential to keep conceptually distinct. The visible symbol obviously has some aspects by which we interpret it as representing one character rather than another; but its appearance may also be significantly determined by features that have no effect on our notion of which character in a writing system it represents. A familiar instance is the lower case 'a', which in printed texts may be represented either by a 'single storey' symbol (cf. figure 1 in the examples from Baskerville SemiBold or Century) or by a 'two storey' version (as in figure 1 in the examples from ArialRegular or Andale Mono Regular). We say that the single and double-storey symbols both represent one and the same the same *abstract character*'a' using two different *glyphs*. Similarly, an uppercase 'A' in a serif typeface has additional strokes that are absent from the same letter when printed using a sans-serif typeface, so that once again we have differing glyphs standing for the same abstract character. The distinction between abstract characters and glyphs is fundamental to all machine processing of documents.

In most scholarly encoding projects, the accurate recording of the abstract characters which make up the text is of prime importance, because it is the essential prerequisite of digitizing and processing the document without semantic loss. In many cases (though there are important exceptions, to be touched on shortly) it may not be necessary to encode the specific glyphs used to render those abstract characters in the original document. An encoding that faithfully registers the abstract characters of a document allows us to search and analyse our document's content, language and structure and access its full semantics. That same encoding, however, may not contain sufficient information to allow an exact visual representation of the glyphs in the source text or manuscript to be recreated.

The importance of this distinction between information content and its visual representation is not always immediately apparent to people unused to the specific complexities of text handling by machine. Such users tend to ask first what (in order of conceptual priority) should actually be their very last question: how do I get a physical image that looks like character x in my source document to appear on to the screen or the output page? Their first question should in fact be: how can I get an abstract representation of character x into my encoded document in a way that will be universally and unambiguously identifiable, no matter what it happens to look like in printout or on any particular display? And occasionally the response they receive as a result of their misguided initial question is a custom 'solution' that satisfies their immediate rendering wishes at the price of making their underlying document unintelligible to other users (or even to the original user in other times and places) because it encodes the abstract character in an idiosyncratic way.

That said, there will certainly be documents or projects where it is a matter of scholarly significance that the compositor or scribe chose to represent a given abstract character using one particular glyph or set of strokes rather than a semantically-equivalent but visually distinct alternative, and in that case the specific appearance of the form will have to be encoded on one way or another. But that encoding need not (and in most cases will not) involve a notation that visually resembles the original, any more than italicised text in an original document will be represented by the use of italic characters in the encoded version.

A collection of the abstract characters needed to represent documents in a given writing system is known as a character set, and the character set or character repertoire of a processing or rendering device is the set of abstract characters that it is equipped to recognise and handle appropriately. There is, however, a subtle distinction between these two parallel uses of the same term, involving one more key concept which it is essential to grasp. The character set of a document (or the writing system in which it is recorded) is purely a collection of abstract characters. But the character set of a computing device is a set of abstract characters which have been mapped in a well-defined way to a set of numbers or code-points by which the device represents those abstract characters internally. It can therefore be referred to as a coded character set, meaning a set of abstract characters each of which has been assigned a numerical code point (or in some instances a sequence of code-points) which unambiguously identifies the character concerned.

It is now possible to use this terminology to say what Unicode is: it is a coded character set, devised and actively maintained by an international public body, where each abstract character is identified by a unique name and assigned a distinctive code-point. Although only Unicode is mentioned here explicitly, it should be noted that the character repertoire and assigned code-points of Unicode and the ISO standard 10646 are identical and maintained in a way that ensures this continues to be the case. Unicode is distinguished from other, earlier and co-existing coded character sets by its (current and potential) size and scope; its built-in provision for (in practical terms) limitless expansion; the range and quality of linguistic and computational expertise on which it draws; the commitment in principle (and to an increasing degree in practice) to implement it by all important providers of hardware and software worldwide; and the stability, authority and accessibility it derives from its status as an international public standard.

Abstract characters, glyphs and encoding scheme design The distinction between abstract characters and glyphs can be crucial when devising an encoding scheme. Users performing text retrieval, searching or concordancing will expect the system to recognise and treat different glyphs as instances of the same character; but when perusing the text itself they may well expect to see glyph variants preserved and rendered. When encoding a pre-existing text, the encoder must determine whether a particular letter or symbol is a character or a glyphic variant. A model of the relationship between characters and glyphs has been developed within the Unicode Consortium and the ISO working group SC251 and will form the base for much future standards work.

The model makes explicit the distinction between two different properties of the components of written language:

- their content, i.e. its meaning and phonetic value (represented by a character)
- their graphical appearance (represented by a glyph)

When searching for information, a system generally operates on the content aspects of characters, with little or no attention to their appearance. A layout or formatting process, on the other hand, must of necessity be concerned with the exact appearance of characters. Of course, some operations (hyphenation for example) require attention to both kinds of feature, but in general the kind of text encoding described in these Guidelines tends to focus on content rather than appearance (see further «6.3 Highlighting and Quotation»).

An encoder wishing to record information about which glyphs are present in a given document may do so at either or both of two levels:

- the level of character encoding, using an appropriate Unicode code point to represent the glyph concerned
- the markup level, with the glyph indicated via appropriate elements and/or attributes

The encoding practice adopted may be guided by, among other things, an assessment of the most frequent uses to which the encoded text will be put. For example, if recognition of identical characters represented by a variety of glyphs is the main priority, it may be advisable to represent the glyph variations at markup level, so that the character value can be immediately exposed to the indexing and retrieval software. Plainly, an encoding project will need to consider such issues carefully and embody the outcome of their deliberations in local manuals of procedure to ensure encoding consistency. Using Unicode code points to represent glyph information requires that such choices be documented in the TEI Header. Such documentation does cannot of itself guarantee proper display of the desired glyph but at least makes the intention of the encoder discoverable.

At present the Unicode Standard does not offer detailed specifications for the encoding of glyph variations. These Guidelines do give some recommendations; some discussion of related matters is given in «Chapter 18 Transcription of Primary Sources», and «Chapter 25 Representation of non-standard Characters and Glyphs» offers some features for the definition of variant glyphs.

Entry of characters. Text characters may be entered into a document using any of three methods, in any convenient combination. First, where suitable input facilities make this possible, the characters concerned may be entered directly into the document, either by normal keystrokes or by the use of one of the Input Method Editors (IMEs) commonly used for the entry of ideographic characters. This is most likely to be convenient where the display used for text entry and/or the printer used to produce output for proof-reading purposes is capable of rendering the characters concerned using correct and readily identifiable glyphs. Where such easily checkable rendering is not available, or where there is no suitable method of inputting certain characters directly, they may be input by one of two possible forms of indirect notation or 'reference'.

The first form of reference is a Numeric Character Reference (NCR), which takes the general form `&#D;` where `D` is an integer representing the code-point of the character in base 10, or `&#xH;`, where `H` is the code-point in hexadecimal notation. This has the advantage that no declaration of what this notation means is required anywhere in the document instance or its DTD. Every XML processor is capable of recognising NCRs and replacing them with the required code-point value without needing access to any additional data. The disadvantage of NCRs as a means of entering, representing and proofing character data is that most human beings find them anything but 'readable' and it is all too easy for the wrong character to be entered in error and retained undetected.

The second form of reference is a Character Entity Reference (though, as explained below, this should not be taken to imply that such entities constitute a 'type' that could be distinctively recognised by a processing system). Character entity references can (and indeed should) have names whose significance is apparent to humans, but each and every entity name has to be associated with its replacement (which as explained below should be a character value, possibly in the form of a NCR) via a formal declaration in the document's internal or external subset. For a large number of characters defined by Unicode and commonly used in documents, there are ISO entity sets declaring mnemonic names which should be used wherever feasible: XML compatible character entity declarations using ISO names and suitable for inclusion into the subset are available on the TEI websites.

Where characters are not defined in Unicode and so have to be assigned both a local code-point and a local entity name of the project's choosing (see 1.1.2.6.2. Non Unicode characters in XML documents below) it is highly desirable to follow the same nomenclature principles as ISO and to emulate the practice in the ISO character entity declarations of appending a string giving the character a unique descriptive name as a comment to the actual entity declaration. In addition, where different groups or projects are working on texts with geographical, historical, linguistic or other similarities that give rise to common issues of character encoding, it is highly advisable in the interests of consistency that they should consult one another when devising entity names. The TEI mailing list may provide a suitable first point of contact for such consultations. Further advice on the matter of locally-defined characters is contained in «Chapter 25 Representation of non-standard Characters and Glyphs.»

Output of characters Rendering of the encoded text is a complicated process that depends largely on the purpose, external requirements, local equipment and so forth, it is thus outside the scope of coverage for these Guidelines.

It might however nevertheless be helpful to put some of the terminology used for the rendering process in the context of the discussion of this chapter. As was mentioned above, Unicode encodes abstract characters, not specific glyphs. For any process that makes characters visible, however, concrete, specifically designed glyph shapes have to be used. For a printing process, for example, these shapes describe exactly at which point ink has to be put on the paper and which areas have to be left blank. If we want to print a character from the Latin script, besides the selection of the overall glyph shape, this process also requires that a specific weight of the font has been selected, a specific size and to what degree the shape should be slanted. Beyond infividual characters, the overall typesetting process also follows specific rules of how to calculate the distance between characters, how much whitespace occurs between words, at which points line breaks might occur and so forth.

If we concern ourselves only with the rendering process of the characters themselves, leaving out all these other parameters, we will realize that of all the information required for this process, only a small amount will be drawn from the encoded text itself. This information is the code-point used to encode the character in the document. With this information, the font selected for printing will be queried to provide a glyph shape for this character. Some modern font formats (e.g. OpenType) do implement a sophisticated mapping from a code-point to the glyph selected, which might take into account surrounding characters (to create ligatures where necessary) and the language or even area this character is printed for to accomodate different typesetting traditions and differences in the usage of glyphs.

A TEI document might provide some of the information that is required for this process for example by identifying the linguistic context with the lang attribute. The selection of fonts and sizes is usually done in a stylesheet, while the actual layout of a page is determined by the typesetting system used. Similarily, if a document is rendered for publication on the Web, information of this kind can be shipped with the document in a stylesheetThe World Wide Web Consortium provides recommendations for two standard stylesheet languages: either CSS or XSL could be used for this purpose..

Unicode and XML The devisers of the XML standard took the view that Unicode should be the only means of representing abstract characters which conformant XML processors were obliged to support. That certainly does not preclude the use of other character encoding schemes or character sets in documents which are to be handled by XML processors, but it does mean that all the abstract characters which are encoded as characters (as distinct from being represented indirectly via markup) in an XML document must either possess an assigned code point within the public Unicode standard, or be assigned a code-point devised by and specific to the local project, taken from a reserved range set aside by the standard expressly for this purpose, the so-called Private Use Areas or PUAs. For the vast majority of projects to which these Guidelines are applicable, the Unicode standard will already offer code-points for all the abstract characters their documents employ, and so the requirement that all such characters should be resolvable by XML processors to Unicode code-points will not involve any definition or use of PUA code-points. Indeed, such projects are not obliged by their choice of XML to use Unicode in their documents. Provided they correctly declare at the requisite points any non-Unicode coded character set they may use, ensure that all their XML processors support their declared encoding, and then consistently employ that encoding in strict conformity with their declarations, they need not consciously concern themselves with Unicode unless and until they feel it is appropriate to do so.

Non-Unicode character sets and XML processors There are, however, strict limits to the way conformant XML processors handle documents whose character set is not Unicode, and unless these limits are understood it is likely that projects not yet ready to commit to Unicode across the board will run into unexpected and baffling problems as they attempt to operate with their legacy character encodings. First, it must be repeated that nothing in the XML standard *requires* conformant processors to handle non-Unicode documents. But even if there were any actual processors which on that basis refused to process non-Unicode documents, that would not limit their usefulness as severely as might at first appear. The reason is that there is a way of internally representing Unicode code-points (explained further 1.1.2.9.1. Encoding errors related to UTF-8 below) where there is no detectable difference between a document which is actually encoded in ASCII employing only 7-bit values and one which is encoded in Unicode but which happens to contain only the abstract characters encompassed by the 7-bit ASCII standard. And the XML standard specifies that this way of representing Unicode is the one which processors must assume as the default for any document that does not explicitly declare an encoding. At a stroke, this provision ensures that all pure 7-bit ASCII encoded documents can be processed without further ado by all conformant XML processors. Add to this the provision, also within the XML standard, that allows any Unicode code-point to be indirectly specified using only 7-bit ASCII characters via a Numeric Character Reference (NCR), and the upshot is that all documents in non-Unicode encodings which can be pre-processed to rewrite any characters outside the 7-bit ASCII range as Unicode code-points in NCR notation (a simple batch procedure for which software is readily available) can be handled even by processors which have no inbuilt support for any encoding other than Unicode.

In fact, every XML processor so far released has implemented methods, specified in the standard though not mandatory, which allow the processing of documents in at least some non-Unicode character sets. Such processors include in their documentation a statement of the non-Unicode encodings they support, and the use of such an encoding must be declared to the processor in the correct way.

To avoid confusion when taking advantage of such encoding support, it is first of all essential to grasp that an encoding declaration in an XML document is indeed simply a declaration: it is not an incantation that magically converts the document that follows into the encoding concerned. It is a common error to think that simply declaring a document's encoding to be, say ISO-8859-1 (or for that matter UTF-8 or UTF-16, the representations of Unicode for which support is mandatory) is sufficient to 'make it so'. Such a declaration is useless unless the document that follows actually is encoded strictly in conformance with the declaration. Some of the circumstances in which that may not in fact be the case are outlined in 1.1.2.9. Issues arising from the internal representations of Unicode below. Secondly, an encoding declaration does not somehow switch an XML processor into a mode where it works entirely in the declared encoding for as long as the declaration is in scope. On the contrary, all it does is instruct the processor to pass its input through a filter that immediately converts all the code-points in the declared encoding into their Unicode counterparts; from that point onwards the document as seen by all subsequent stages of processing is actually in Unicode, even though that may not be apparent to the user. Thirdly, this invariable internal conversion has a crucial consequence: the fact that a processor can successfully accept a document in a non-Unicode encoding does not mean that it will necessarily convert any output it may produce back into the declared input encoding. Internally, the document has been converted to and processed in Unicode, and there is nothing in the XML standard that requires the reverse conversion to be performed at the output stage. Most processors go beyond the standard by offering a facility to output in various encodings: but whether it is available and how to use it must be ascertained from the processor's documentation. Should it be unavailable or unreliable, the output may need to be post-processed through a character convertor to restore the original encoding, and again such software is freely available and easy to use.

Non Unicode characters in XML documents In the cases considered in the preceding section, there was a suitable Unicode code-point corresponding to each abstract character contained in the non-Unicode character set of the input document. In such instances, the mandatory internal conversion to Unicode carried out by the processor can be more or less transparent to a user who wishes to continue to work with a non-Unicode character set. Things become rather different when the non-Unicode character set contains abstract characters for which there is no code point in the Unicode standard, or when a project that is attempting to work in Unicode throughout finds that it needs to represent abstract characters not currently provided for in the Unicode standard. Here, a significant difference between SGML and XML emerges in a rather troublesome way.

Following their agenda to devise a subset of SGML that would be significantly easier to implement, the authors of the XML specification decided that one particular type of entity available in SGML, known as an internal SDATA entity, should not be carried over into XML. It would be idle to question that decision here, but its consequences for the handling of abstract characters for which there is no Unicode definition were significant.

The procedures recommended in earlier versions of these Guidelines for encoding, processing and exchanging what we might call locally defined abstract characters were reliant on the availability of entities declared as of type SDATA, but that type is not supported in XML, and there is therefore no ready equivalent for XML-based projects to the recommendations previously offered. In essence, when an SGML parser encounters a reference to an entity of type SDATA, it supplies to the application which it is servicing the name of that entity, as found in the document, plus a pointer to a location somewhere on the local system, and what is present at that location may in turn allow or instruct the application to do one of a number of things, including looking up the entity name in a table and deriving information about the referenced entity which can trigger specific behaviours in the application appropriate to the processing of that abstract character. There is however no way to make an XML parser do anything of the kind in response to an entity reference. Entities in XML are really only of two basic types, parsed and unparsed. Unparsed entities are of no relevance here. References to parsed entities in an XML document result in only one kind of behaviour: when they appear in the parser's input stream, the parser expects to be able to resolve them by locating a declaration in the document's internal or external subset which maps the entity name to its replacement text. The parser then inserts that replacement text into the document in place of the entity reference, which is discarded without trace. The act of replacement is not notified to the application, except where it fails because the entity is undeclared or the declaration is in some way defective (in which case the parser signals a fatal error and stops.)

Though for explanatory convenience much XML-related documentation, including these Guidelines, refers specifically to Character Entities and Character Entity References, a character entity in XML is not a distinct 'type' in the sense that 'type' is understood in Computer Science terminology, for example when referring to the type of an attribute. Hence there is no way in which editing or other software can check that the replacement to be inserted is indeed a single character or its equivalent rather than an arbitrary chunk of text, possibly including markup. A character entity is simply a general entity whose replacement text happens to be declared as a character value or a NCR representing that value. This has two important consequences if it is proposed to use such an entity reference to stand for a character that has no Unicode equivalent. First, the entity name reference will disappear at an early stage in the parse and be replaced by the declared value of the entity, so that no processing which requires access in the parsed document to the entity reference as originally entered is possible. Secondly, if a character entity is to be used as a true equivalent to a normal character, and consequently be employed at all points in a document where a single character could legitimately occur (apart from in element and attribute names, where no references of any kind are allowed) then it is essential that its replacement value indeed be pure character data. If the replacement value of the entity were to contain any markup, or a processing instruction, there would be many places in a document where simple character data would be legitimate, but where the substitution of markup or some other replacement could cause the document to become invalid or malformed. Taken together, these considerations mean that the transparent use of a CER to stand for a non-Unicode character in an XML document is simply not possible.

Special aspects of Unicode character definitions

Compatibility characters The principles of Unicode are judiciously tempered with pragmatism. This means, among other things, that the actual repertoire of characters which the standard encodes, especially those parts dating from its earlier days, include a number of items which on a strict interpretation of the Unicode Consortium's theoretical approach should not have been regarded as abstract characters in their own right. Some of these characters are grouped together into a code-point regions assigned to compatibility characters. Ligatures are a case in point. Ligatures (.e.g. the joining of adjacent lower-case letters 's' and 't' or 'f' and 'i' in Latin scripts, whether produced by a scribal practice of not lifting the pen between strokes or dictated by the aesthetics of a type design) are representational features with no added semantic value beyond that of the two letters they unite (though for historians of typography their presence and form in a given edition may be of scholarly significance). However, by the time the Unicode standard was first being debated, it had become common practice to include single glyphs representing the more common ligatures in the repertoires of some typesetting devices and high-end printers, and for the coded character sets built into those devices to use a single code point for such glyphs, even though they represent two distinct abstract characters. So as to increase the acceptance of Unicode among the makers and users of such devices, it was agreed that some such pseudo-characters should be incorporated into the standard. Nevertheless, if a project requires the presence of such ligatured forms to be encoded, this should normally be done via markup, not by the use of a compatibility character. That way, the presence of the ligature can still be identified (and if desired, rendered visually) where appropriate, but indexing and retrieval software will treat the code-points in the document as a simple sequential occurrence of the two constituent characters concerned and so correctly align their semantics with non-ligatured equivalents. Such ligatures should not be confused with digraphs (usually) indicating diphthongs, as in the French word "cœur". Digraphs are atomic orthographic units representing abstract characters in their own right, not purely glyphic amalgamations, and indexing and retrieval software must treat them as such. Where a digraph occurs in a source text, it should normally be encoded using the appropriate code-point for the single abstract character which it indeed represents, either by direct entry of the character concerned of through the appropriate CER or NCR.

Precomposed and combining characters and normalization The treatment of characters with diacritical marks within Unicode shows a similar combination of rigour and pragmatism. It is obvious enough that it would be feasible to represent many characters with diacritical marks in Latin and some other scripts by a sequence of code-points, where one code-point designated the base character and the remainder represented one or more diacritical marks that were to be combined with the base character to produce an appropriate glyphic rendering of the abstract character concerned. From its earliest phase, the Unicode Consortium espoused this view in theory but was prepared in practice to compromise by assigning single code-points to precomposed characters which were already commonly assigned a single distinctive code-point in existing encoding schemes. This means, however, that for quite a large number of commonly-occurring abstract characters, Unicode has two different, but logically and semantically equivalent encodings: a precomposed single code point, and a code-point sequence of a base character plus one or more combining diacritics. Scripts more recently added to Unicode no longer exhibit this code-point duplication (in current practice no new precomposed characters are defined where the use of combining characters is possible) but this does nothing to remove the problem caused by the duplications permanently embodied in older strata of the character set. Together with essentially analogous issues arising from the encoding of certain East Asian ideographs, this duplication gives rise to the need to practice normalization of Unicode documents. Normalization is the process of ensuring that a given abstract character is represented in one way only in a given Unicode document or document collection. The Unicode Consortium provides four standard normalization forms, of which the Normalization Form C (NFC) seems to be most appropriate for text encoding projects. The World Wide Web Consortium has produced a document entitled Character Model for the World Wide Web 1.0Available at `http://www.w3.org/TR/charmod`., which among other things discusses normalization issues and outlines some relevant principles. An authoritative reference is Unicode Standard Annex #15 Unicode Normalization Formsavailable at `http://www.unicode.org/reports/tr15/`. Individual projects will have to decide how far their decisions on normalization need be influenced by the fact that at present, by no means all hardware or software can correctly render (or even consistently identify) abstract characters encoded using combining symbols. It should be noted however, that normalization as discussed in the documents above does not cover the problems mentioned above with East-Asian characters, except for issues connected with composed characters in Hangul.

It is important that every Unicode-based project should agree on, consistently implement and fully document a comprehensive and coherent normalization practice. As well as ensuring data integrity within a given project, a consistently implemented and properly documented normalization policy is essential for successful document interchange.

Character semantics In addition to the Universal Character Set itself, the Unicode Consortium maintains a database of additional character semantics`http://www.unicode.org/ucd/`. This includes names for each character code-point and normative properties for it. Character properties, as given in this database, determine the semantics and thus the intended use of a code-point or character. It also contains information that might be needed for correctly processing this character for different purposes. This database is an important reference in determining which Unicode code-point to use to encode a certain character.

In addition to the printed documentation and lists made available by the Unicode consortium, the information it contains may also be accessed by a number of search systems over the web (e.g. `http://www.eki.ee/letter/`). Examples of character properties included in the database include case, numeric value, directionality, and, where applicable status as a 'compatibility character'At the time of writing (April 2004) there is a draft Unicode Technical Report (#23), The Unicode Character Property Model that goes into greater detail on properties. The current version, which will undergo further revision, is at `http://unicode.org/reports/tr23/tr23-2.html`.. Where a project undertakes local definition of characters with code-point in the PUA, it is desirable that any relevant additional information about the characters concerned should be recorded in an analogous way, as further discussed under «Chapter 25 Representation of non-standard Characters and Glyphs.»

Character entities in non-validated documents An important difference between SGML and XML is that the latter allows for the processing of non-validated documents. Since validity and validation are central TEI concerns, it is unlikely that documents prepared according to these Guidelines will ever be designed or implemented as merely well-formed in the XML sense. However in the domain of XML technologies, even where a document declares a DTD or schema, it is not always necessary or appropriate that the parser perform full validation. XSLT transformation is a common case in point. By the workflow stage at which a document is handed off to an XSLT process for transformation, it is likely that its associated DTD or schema will already have fulfilled its role of integrity assurance and quality control, and so it may be undesirable to add validation to the processing overhead. For this reason, most XSLT processors do not attempt validation by default, even if a DTD is declared and accessible. This can, however, create a problem where parsed entities, (and character entities in particular in the present context) are referenced. A validating parser reads all entity declarations from the DTD (including those for character entities) in the initial phase of processing, so that they can be resolved as and when required. However, where no validation takes place, it cannot automatically be assumed that the parser will be able to resolve such entities in all circumstances. The XML standard requires a non-validating parser to read and act on entity declarations only if they are located within the document's internal subset (which does not, of course, mean that the entity declarations have to be manually merged into the document instance in advance of processing: character entity sets, for instance, count as being in the internal subset if they are placed there via a parameter entity, as is normal TEI practice). Some parsers when in non-validating mode will also access entity declarations in the external subset, but this behaviour is not mandated by the standard and should not be relied upon. Provided these facts are borne in mind, the presence of character entities in a document when parser validation is switched off should not cause any difficulties.

Issues arising from the internal representations of Unicode In theory it should not be necessary for encoders to have any knowledge of the various ways in which Unicode code-points can be represented internally within a document or in the memory of a processing system, but experience shows that problems frequently arise in this area because of mistaken practice or defective software, and in order to recognise the resulting symptoms and correct their causes an outline knowledge of certain aspects of Unicode internal representation is desirable.

Encoding errors related to UTF-8 The code-points assigned by Unicode 3.0 and later are notionally 32-bit integers, and the most straightforward way to represent each such integer in computer storage would be to use 4 eight-bit bytes. However, many of the code-points for characters most commonly used in Latin scripts can be represented in one byte only and the vast majority of the remainder which are in common use (including those assigned from the most frequently used PUA range) can be expressed in two bytes alone. This accounts for the use of UTF-8 and UTF-16 and their special place in the XML standard. UTF-8 and UTF-16 are ways of representing 32-bit code points in an economical way.

UTF-8 is a variable length-encoding: the more significant bits there are in the underlying code-point (or in everyday terminology the bigger the number used to represent the character), the more bytes UTF-8 uses to encode it. What makes UTF-8 particularly attractive for representing Latin scripts, explaining its status as the default encoding in XML documents, is that all code points that can be expressed in seven or fewer bits (the 127 values in the original ASCII character set) are also encoded as the same seven or fewer bits (and therefore in a single byte) in UTF-8. That is why a document which is actually encoded in pure 7-bit ASCII can be fed to an XML processor without alteration and without its encoding being explicitly declared: the processor will regard it as being in the UTF-8 representation of Unicode and be able to handle it correctly on that basis.

However, even within the domain of Latin-based scripts, some projects have documents which use characters from 8 bit extensions to ASCII, e.g. those in the ISO-8859-n series of encodings, and the way characters which under ISO-8859-n use all eight bits are encoded in UTF-8 is significantly different, giving rise to puzzling errors. Abstract characters that have a *single* byte code-point where the highest bit is set (that is, they have a decimal numeric representation between 129 and 255) are encoded in ISO-8859-n as a *single*byte with the same value as the code-point. But in UTF-8 code-point values inside that range are expressed as a *two* byte sequence. That is to say, the abstract character in question is no longer represented in the file or in memory by the same number as its code-point value: it is **transformed** (hence the T in UTF) into a sequence of two different numbers. Now as a side-effect of the way such UTF-8 sequences are derived from the underlying code-point value, many of the single-byte eight-bit values employed in ISO-8859-n encodings are illegal in UTF-8.

This complicated situation has a simple consequence which can cause great bewilderment. XML processors will effortlessly handle character data in pure 7-bit ASCII without that encoding needing to be declared to the parser, and will similarly accept documents encoded in an undeclared ISO-8859-n encoding if they happen to use no characters outside the strict ASCII subset of the ISO character sets; but the parse will immediately fail if an eight-bit character from an ISO-8859-n set is encountered in the input stream, unless the document's encoding has been explicitly and correctly declared. Explicitly declaring the encoding ought to solve the problem, and if the file is correctly encoded throughout, it will do so. But since text editors and word processors are currently acquiring different degrees of Unicode support at different rates, projects are likely to find that they have to deal with some files encoded in UTF-8 along with others in, say, ISO-8859-1. Such encoding differences may go unnoticed, especially if the proportion of characters where the internal encodings are distinguishable is relatively small (for example in a long English text with a smattering of French words). If in the process of document preparation two such files have been merged, or intermixed via 'cut and paste' techniques, it is all too possible that the internal encodings of the resulting files will have become mixed as well. Thanks to misplaced notions of 'user friendliness' some current editing software silently corrects such miscodings as it displays the text, so that they remain hidden until the XML parser terminates with a fatal 'invalid character'error.

Where erroneously mixed encodings are the source of such an error, altering the encoding declaration will not solve the problem, though it may obfuscate it. Eight-bit character codes in a file declared as UTF-8 will always stop the parser. More insidiously, UTF-8 sequences in a file declared as ISO-8859-1 will not halt the parse, but will cause data corruption, because the parser will silently but erroneously convert each byte in every UTF-8 sequence into a spurious separate character, introducing semantic errors which may not become apparent until much later in the processing chain.

In projects that routinely handle documents in non-Latin scripts, everyone is well aware of the need to ensure correct and consistent encoding, so in such places mixed encoding problems seldom arise, and when they do are readily identified and remedied. Real confusion tends to arise, however, in projects which have a low awareness of the issues because they employ predominantly unaccented Latin characters, with only thinly-distributed instances of accented letters, or other 'special characters' where the internal representation under ISO-8859-n and UTF-8 are different (such as the copyright symbol, or, a frequent troublemaker where eventual HTML output is envisaged, the 'non-breaking space'). Even, or especially, if such projects view themselves as concerned only with English documents, the close relationship between XML and Unicode means they will need to acquire an understanding of these encoding issues and develop procedures which assure consistency and integrity of encoding and its correct declaration, including the use of appropriate software for transcoding and verification.

Encoding errors related to UTF-16 The advantages of UTF-8 as an internal representation of Unicode code-points outlined above do not obtain where documents are in scripts other than Latin, Cyrillic or Hebrew. Where characters with code-points in the sixteen-bit range (two-byte) predominate, UTF-8 is inappropriate, because it requires three or more bytes to represent each abstract character. Here the preferred representation of Unicode (which all XML-conformant parsers must support) is UTF-16, where each code-point corresponding to an abstract character is represented in two eight-bit bytes The use of 'surrogate' values to represent code-points beyond the 16-bit range is passed over here, since it adds a complication that does not affect the key points at issue. This encoding presents a different hazard, especially while support for Unicode in editing software is relatively uneven and immature. Because the code-points are represented as sixteen-bit integers stored (in most popular computers) in two separate bytes, the order in which those bytes are stored becomes important. This is dependent on the underlying hardware. In the realm of desktop computing, Macintosh machines, for example, store (on disk as well as in memory) byte pairs representing 16-bit integers with the higher-value byte first, whereas PCs using Intel processors store the bytes in the reverse order (this is often referred to with Swiftian nomenclature as big-endian versus little-endian byte order). This means that if a semantically identical plain text file encoded in UTF-16 is prepared on a Macintosh and on a PC, and the two files are then saved to disk, each byte pair in one file will be in the reverse order from the corresponding byte pair in the other file. To avoid the obvious incompatibility problems, the XML standard requires that all documents whose declared encoding is UTF-16 must begin with a special pseudo-character which is not itself part of the document, but merely a Byte Order Marker (BOM) from which the processor can determine the byte order of the document that follows. Now the insertion of a correct BOM and the consistent maintenance of the byte order throughout the file ought to be taken care of transparently by software, but experience, especially from environments where work is distributed across big-endian and little-endian hardware, shows that this cannot always be taken for granted in the current state of software development. As with mixed encoding problems involving UTF-8, inconsistent byte-order in UTF-16 files seems to be the result of merging or cutting and pasting between files using software which does not correctly enforce byte order integrity, and out of misconceived ‘user friendliness’ which conceals byte-order inconsistencies from the user. Once more, the result can be files which look correct in an editor, but which the XML parser either rejects outright or silently passes on in a seriously garbled form. Again, to avoid the consequent errors, projects need to cultivate an informed awareness of relevant encoding issues and devise policies to avoid them in the first place or detect them at an early stage.

1.2 Class catalogue

1.3 Pattern catalogue

1.4 Element catalogue