<draft-ietf-acap-langtag-00.txt> University of Zurich
Expires in six months June 1997

Two Alternative Proposals for Language Taging in ACAP

Status of this Memo

Abstract

For various computing applications, it is helpful to know the language of the text being processed. This can be the case even if otherwise only pure character sequences (so-called plain text) are handled. From several sides, the need for such a scheme for ACAP has been claimed. One specific scheme, called MLSF, has also been proposed, see draft-ietf-acap-mlsf-01.txt for details. This document proposes two alternatives to MLSF. One alternative is using text/enriched-like markup. The second alternative is using a special tag-introduction character. Advantages and disadvantages of the various proposals are discussed. Some general comments about the topic of language tagging are given in the introduction.

1. Introduction

This introduction contains some considerations about language information that should help to better understand why and where language information can be beneficial. They are intended for general information, and are not directly related to the specifics of the proposals made in this document.

1.1 Multilingual Text

It is sometimes claimed that text, in order to be multilingual, has to contain language information of some kind or another. This is definitely not the case. Multilingual text has existed for centuries on paper and other writing materials, and for decades in computers, without language information. A huan reader with the necessary language background is always able to understand a multilingual text without explicitly being told which language each word or character belongs to. In some cases, there may be ambiguities, but this is either intended, such as in a pun or joke, it is because the reader is not fully familliar with the involved languages, or it is because the writer was no precise enough.

The overwhelming majority of characters always has been used in various languages, and a character per se therefore cannot be associated to a single language. This likewise applies to words (and sometimes even phrases) out of context.

1.2 Language Taging

While the human reader does not need special language information, such information can be useful for the purpose of automatic processing of various kinds. These in particular include indexing and searching, text-to-speech conversion, other conversion operations such as case conversion and transliteration, spelling and grammar checks, and high-quality typography.

Two other operations are frequently mentionned as benefiting from language information: Sorting and machine translation. However, in the case of sorting, this has to occur according to the expectations of the viewer, frequently encapsulated as a so-called locale. The language of each of the items being sorted is not relevant. In the case of machine translation, the knowledge and effort to translate a language is by magnitudes higher than the knowledge needed to decide whether a certain word or sentence belong to a given language. Explicit language information is therefore of marginal importance. This also applies to other operations, such as text-to-speech conversion, in particular for high quality and for languages with a complicated relationship between spelling and pronounciation (such as English).

1.2 CJK(V) Glyph Disambiguation

The reason one hears most for the necessity of language information is the need to disambiguate CJK(V) ideographic glyphs, i.e. to select specific typographic variants of certain ideographic characters, variants which can differ somewhat between Chinese (simplified or traditional), Japanese, Korean, or classical Vietnamese. Some such distinctions can indeed be made by using language information to indicate typographic tradition. However, the usefullness of this approach is limited by a series of facts that are not all very widely known:

- Even if a glyph is by mistake taken from another typographic tradition, readability, in particular in context, is never affected.

- In running text, the differences resulting from the use of different fonts (e.g. Song-style font vs. Mincho-style) as well as from different weights of the same font are much more visible than the differences resulting from glyph variant details.

- In many fonts, glyphs vary, for sund aestetic and historic reasons, to a similar or higher degree than that exhibited in the "reference" glyphs for each typographic tradition as given in [ISO 10646]. This applies to print, but even more to handwriting.

- National standards explicitly or implicitly allow for a certain variance of glyph shapes. In particular, the newest edition of the basic Japanese character standard, JIS X 0208-1997 [JIS1997], explicitly mentionnes a large number of permitted variants (pp. 12-22) for Mincho fonts only. It also explicitly allows a list of 29 much wider-reaching variants as a consequence of some unfortunate changes to the standard in 1983 (p. 22).

- Long-standing typographic practice does not use special glyph variants for representing short inclusions of foreign origin (such as names of persons, places, or institutions) in native text.

- Some glyph variants are seen by some persons as explicit properties of their names. Identifying a name by a particular language and assuming that this implies a particular typographic tradition can in some cases lead to the desired result. However, the results cannot be guaranteed due to design differences between different fonts used in the same typographic tradition, and due to the fact that even national standards glyph standards considerably unify glyph variants.

All the above facts clearly limit the usefullness of language tags for CJK(V) glyph variant selection. Language taging should therefore not be advertized as a comprehensive solution to the various problems of CJK(V) glyph variant selection.

2. A Text/Enriched-like Notation for Language Tags (TELT)

This section specifies a text/enriched-like notation for language tags, leading to a format simmilar to text/enriched. It can be used with any character encoding that contains the necessary subset of the US-ASCII character repertoire.

Language tags are of the form "<LANG=xxxxx>" where xxxxx is a language tag as defined in [RFC1766], with all letters written in upper case. No whitespace of any kind is allowed between "<" and ">".

Language alternatives are started by "<ALTLANG>". Again, no whitespace is allowed between "<" and ">".

The use of the character sequences "<LANG=" and "<ALTLANG>" is not allowed in the text itself. Code to convert from this notation to MLSF and back and to test for false positives in plain text search is given in an appendix.

3. Language Tags using a Start Tag Character (STLT)

This method of language taging is only useable with character encodings that can represent the BMP of the Universal Character Set [ISO10646]. For the purpose of illustration, the character PILCROW SIGN (paragraph sign, U+00B6) is used as the tag start character. It would be preferable to officially define a currently unused code point exclusively for this purpose, but such a definition is outside of the scope of this document and outside of the scope of IETF work. If this solution is seriously considered for adoption by the IETF for use in some of it's protocols, a request for such a codepoint should be made through the appropriate channels.

For possible future expansions, tag syntax after the start tag character is kept very simple and general. Tags are defined to start with a tag start character, contain only characters from the US-ASCII repertoire (U+0021 through U+007E, inclusive), excluding the tag end character, and end with a tag end character. The character "#" is choosen as a tag end character.

Language tags proper are formed by a start tag character, a language tag according to [RFC1766], with all letters in upper case, and a "#" as an end tag character, without any intervening white space.

Language alternatives are marked by a sequence of a start tag character, a "%", and a "#" as an end tag character, again without any intevening white space. Code to convert from this notation to MLSF and back and to test for false positives in plain text search is given in an appendix.

4. Conformance

Conforming protocols using either of the solutions proposed above MUST clearly define in which places they do so, and in which places they don't. If there are other mechanisms in the procotol that can be used for language taging, these mechanisms should be considered and used. In particular, storing language information separate from the actual text is beneficial in many cases because it allows the protocol to treat language information and language alternatives in a way appropriate to the protocol, i.e. only selecting and transmitting language alternatives desired by the client, and so on.

Conforming protocols and their implementations MUST at all costs avoid that language tags leak into parts of the protocol where they are not allowed or into other channels where they are not allowed. In the absence of specific information to the countrary, a protocol or implementation MUST assume that another protocol or implementation does not allow language tags.

In interfaces to protocols and formats that use other ways of language taging (for an example HTML, see [RFC2070]), conforming protocols SHOULD convert language tags appropriately or MAY eliminate them.

If text including language tags as defined in this document leaks outside the protocol positions where it is explicitly allowed, it should be treated in the same way other text is treated, with no special processing.

5. Discussion

Two alternative forms for language taging have been proposed in this document. Because they are very simillar, only one of them should finally be choosen. Compared to [MLSF], their main advantages are that they can be used with character encodings other than UTF-8, that they are easily distinguished from UTF-8 by implementors and users, and that they are advantageous in case of debuging and initial string composition.

The MLSF proposal has a number of interesting properties that makes it very suitable for efficient internal processing in certain scenarios. We therefore in particular give conversion functions between MLSF and our proposals in the appendices.

MLSF continues a long tradition of utilizing unused bit combinations for internal processing speedups. Exposing such methods to the outside of an implementation, however, can lead to serious restrictions and undesired biases towards certain implementations.

The main difference between the two proposals given here is that TELT has to exclude certain character sequences from the untagged text, whereas STLT has a potential to use a special, newly defined, codepoint, that is guaranteed not to appear in text per se.

Acknowledgements

The motivation to write this document came from Harald Alvestran. Further acknowledgements go to Lisa Moore, Mark Davis, Ken Whistler, Glenn Adams, and others from the UTC (Unicode Technical Committee), to Rob Pike, and to Chris Newman, Ned Freed, and Mark Crispin from the IETF ACAP working group.

Bibliography

[Unicode2] The Unicode Standard, Version 2, Addison-Wesley, Reading, MA, 1996.

[ISO-10646] ISO/IEC 10646-1:1993. International Standard -- Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane.

[JIS1997] Japanese Industrial Standard, "7-bit and 8-bit Double Byte Coded Kanji Sets for Information Interchange", JIS X 0208:1997.

[MLSF] C. Newman, "Multi-Lingual String Format (MLSF)", draft-ietf-acap-mlsf-01.txt, work in progress, June 1997.

[RFC1766] Alvestran, H., "Tags for the Identification of Languages", RFC 1766.

[RFC2070]   F. Yergeau, G. Nicol, G. Adams, and M. Duerst, "Internationalization of the Hypertext Markup Language", RFC 2070, January 1997.

Author's Address

Martin J. Duerst
Multimedia-Laboratory
Department of Computer Science
University of Zurich
Winterthurerstrasse 190
CH-8057 Zurich
Switzerland

Tel: +41 1 257 43 16
Fax: +41 1 363 00 35
E-mail: mduerst@ifi.unizh.ch

NOTE -- Please write the author's name with u-Umlaut wherever possible, e.g. in HTML as Dürst.

Appendix A. Conversion from TELT to MLSF

This is sample code to convert from text/enriched-style language tagging to MLSF. It is assumed that the source is in UTF-8, and that the output buffer (outp) is long enough to hold the result. The code uses the functions defined in [MLSF] for convenience.

```
#include <string.h>

void TELTtoMLSF (unsigned char *outp, unsigned char *inp)
{
    unsigned char tagbuff[256];
    unsigned char *temp;

    while (*inp) {
```

```
            if (!strncmp(inp, "<ALTLANG>", 9)) {
                inp += 9;
                *outp++ = 0xFE;
            }
            else if (!strncmp(inp, "<LANG=", 6)) {
                inp += 6;
                temp= tagbuff;
                while (*inp != '>')
                    *temp++ = *imp++;
                *temp= 0;
                outp += MLSFlangencode(outp, tagbuff);
            }
            else
                *inp++= *outp++;
        }
        *outp= 0;
    }
```

Appendix B. Conversion from STLT to MLSF

This is sample code to convert from Start Tag Character style language taging to MLSF. It is assumed that the source is in UTF-8, and that the output buffer (outp) is long enough to hold the result. The code uses the functions defined in [MLSF] for convenience.

```
void STLTtoMLSF (unsigned char *outp, unsigned char *inp)
{
    unsigned char tagbuff[256];
    unsigned char *temp;

    while (*inp) {
        if (!strncmp(inp, "\xC2\xA7%#", 4)) {
            inp += 4;
            *outp++ = 0xFE;
        }
        else if (!strncmp(inp, "\xC2\xA7", 2)) {
            inp += 2;
            temp= tagbuff;
            while (*inp != '&')
                *temp++ = *imp++;
            *temp= 0;
            outp += MLSFlangencode(outp, tagbuff);
        }
```

```
            else
                *inp++= *outp++;
        }
        *outp= 0;
    }
```

Appendix C. Conversion from MLSF to TELT

This is sample code to convert from MLSF to text/enriched-style language taging. It is assumed that the output buffer (outp) is long enough to hold the result. The code uses the functions defined in [MLSF] for convenience.

```
void MLSFtoTELT (unsigned char *outp, unsigned char *inp)
{
    unsigned char tagbuff[256];
    unsigned char *temp;
    int len;

    while (*inp) {
     /* for speed, first insert a test (*inp != "<") */
        if (*inp == 0xFE) {
            inp++;
            strcpy (outp, "<ALTLANG>");
            outp+= 9;
        }
        else if (*inp >= 0xC0 && inp[1] > 0xC0) {
            inp+= MLSFlangdecode(tagbuff, inp);
            strcpy (outp, "<LANG=");
            outp+= 6;
            temp= tagbuff;
            while (*temp)
                *outp++ = *temp++;
            *outp++ = ">";
        }
        else { /* maybe just *outp++ = *inp++ is enough here? */
            len = utlen[*inp];
            if (len > 6) break;
            while (len-- && *src)
                *outp++ = *inp++;
        }
    }
    *outp= 0;
}
```

Appendix D. Conversion from MLSF to Start Tag Character

This is sample code to convert from MLSF to Start Tag Character style language taging. It is assumed that the output buffer (outp) is long enough to hold the result. The code uses the functions defined in [MLSF] for convenience.

```
void MLSFtoSTLT (unsigned char *outp, unsigned char *inp)
{
    unsigned char tagbuff[256];
    unsigned char *temp;
    int len;

    while (*inp) {
        if (*inp == 0xFE) {
            inp++;
            strcpy (outp, "\xC2\xA7%#");
            outp+= 9;
        }
        else if (*inp >= 0xC0 && inp[1] > 0xC0) {
            inp+= MLSFlangdecode(tagbuff, inp);
            strcpy (outp, "\xC2\xA7");
            outp+= 6;
            temp= tagbuff;
            while (*temp)
                *outp++ = *temp++;
            *outp++ = "&";
        }
        else { /* maybe just *outp++ = *inp++ is enough here? */
            len = utlen[*inp];
            if (len > 6) break;
            while (len-- && *src)
                *outp++ = *inp++;
        }
    }
    *outp= 0;
}
```

Appendix E. Elimination of False Positives in TELT

This is sample code to eliminate false positives in TELT (i.e. checking whether a match found by a search routine starts inside a tag). The elimination of false positives in STLT is structurally equivalent and therefore not given explicitly here.

```
int TELTfalse (unsigned char *inp, unsigned char *pos)
{
    while (inp <= pos) {
        if (!strncmp(inp, "<ALTLANG>", 9)) {
            inp += 9;
            if (inp > pos)
                return 1;
        }
        else (!strncmp(inp, "<LANG=", 6)) {
            inp += 6;
            while (*inp++ != '>')  ;
            if (inp > pos)
                return 1;
        }
    }
    return 0;
}
```

Appendix F. Elimination of Tags from TELT

This is sample code to eliminate tags from TELT (in UTF-8), thereby leaving only plain text. The elimination of tags from STLT is structurally equivalent and therefore not given explicitly here.

```
void TELTclean (unsigned char *inp, unsigned char *outp)
{
    while (*inp) {
        if (!strncmp(inp, "<ALTLANG>", 9))
            inp += 9;
        else if (!strncmp(inp, "<LANG=", 6)) {
            while (*inp++ != '>')  ;
        }
        else
            *inp++= *outp++;
    }
    *outp= 0;
}
```

Appendix G. Selection of the "best" alternative from TELT

This is sample code selects the "best" language match from TELT. Assume input language tag has been converted to upper case. Assume language tags won't exceed 256 characters. Returns a pointer to the start of the "best" match. Code for STLT is structurally equivalent and therefore not given explicitly here.

```
unsigned char TELTselect (unsigned char *inp, unsigned char *tag)
{
    unsigned char tagbuff[256];
    unsigned char *match1, match2;
    unsigned char best= str;
    int bestlen= 0;
    int start= 1;
    int mlen;

    if (tag == NULL || !*tag)
        return;

    while (*inp) {
        if (!strncmp(inp, "<ALTLANG>", 9)) {
            inp += 9;
            *outp++ = 0xFE;
            start= 1;
        }
        if (start) {
            mlen= 0;
            /* get tag into tagbuff */
            if (!strncmp(inp, "<LANG=", 6)) {
                inp += 6;
                match1= tagbuff;
                while (*inp != '>')
                    *match1++ = *imp++;
                *match1= 0;
                inp++;
            }
            else *tagbuff= 0;

            /* check match */
            match1= tagbuff;
            match2= tag;
            while (*match1 && *match1++ == *match2++) {
                if (*match2=="-" && (*match2=="-" || !*match2))
                    mlen = match1 - tagbuff;
            }
```

```
               if (!*match2 && (*match1=='-' || !*match1)) {
                   best = str;
                   break;
               }

               if (mlen > bestlen) {
                   best = str;
                   bestlen = mlen;
               }

               /* search next alternative */
               start = 0;
           }
           else
               inp++;
       }

       return best;
   }
```