Unicode in R7RS small Scheme

This is a proposal for the adoption of Unicode in R7RS small Scheme. I am publishing this document to invite wide comment. There is nothing official about it. I acknowledge the kind help of members of the `r6rs-discuss` mailing list in the discussions that led up to this document. However, I retain sole responsibility for it, including all errors.

Principle #1: No small Scheme implementation is required to support any specific Unicode character or repertoire (collection of characters), with the obvious exception of the ASCII repertoire.

Principle #2: Unicode is the predominant character standard today, and a small Scheme implementation's treatment of characters must conform to it, insofar as this does not conflict with Principle #1.

From these principles I draw the following detailed conclusions (where "Scheme" means "small Scheme, as proposed by me"):

1. The `char->integer` procedure must return an exact integer between `0` and `#xD7FF` or between `#xE000` and `#x10FFFF` when applied to a character supported by the implementation and belonging to the Unicode repertoire. This integer must be the Unicode scalar value of the character.

   This is independent of the implementation's internal representation. For example, a Scheme supporting a repertoire of basic Latin and modern Greek characters only might use the ISO 8859-7 encoding internally, in which lower-case lambda is represented as `#xEB`, but `char->integer` must still return `#x03BB` on that character.

   An ASCII-only Scheme satisfies this requirement automatically, provided it does not deliberately scramble the natural result. (Schemes on EBCDIC systems already have ASCII conversion tables readily available.)

   If the implementation supports non-Unicode characters (ones with bucky bits, e.g.), then `char->integer` must return an exact integer greater than `#x10FFFF` when applied to such characters.

2. The `integer->char` procedure, when applied to an exact integer that `char->integer` returns when applied to some character *c*, must return *c*; that is, `(integer->char (char->integer c)) => c` for any character `c`.

   An ASCII-only Scheme also satisfies this requirement automatically, with the same proviso.

3. The `char<?` procedure and its relatives behave consistently with `char->integer`, as R5RS requires.

4. The `char-ci*` case-independent comparison procedures behave as if `char-foldcase` were applied to their arguments before calling the respective non-`ci` procedures.

5. The procedures `char-{alphabetic,numeric,whitespace,upper-case, lower-case}?` return `#t` if their arguments have the Unicode properties Alphabetic, Numeric, White_Space, Uppercase, or Lowercase respectively. Note that many alphabetic characters (though no ASCII ones) are neither upper nor lower case.

6. The `char-downcase` procedure, given an argument that forms the uppercase part of a Unicode upper/lower-case pair, must return the lowercase member of the pair, provided that both characters are supported by the Scheme implementation. Turkic casing pairs are ignored. If the argument is not the uppercase part of such a pair, it is returned.

7. The `char-upcase` procedure works the same way, *mutatis mutandis*. Note that many Unicode lowercase characters don't have uppercase equivalents.

8. The `char-foldcase` procedure (an extension to R5RS) applies the Unicode simple case-folding algorithm to its argument, ignoring the Turkic mappings. Mappings that don't accept or don't produce single characters are ignored.

   In an ASCII-only Scheme, this is equivalent to the `char-downcase` procedure.

9. The `string<?` procedure and its relatives **are not, contrary to R5RS,** required to be a lexicographical extension of the corresponding procedures for characters. That allows strings to be compared in the native representation without conversion to Unicode. It also allows, at the other end of the spectrum, fully internationalized ISO 14651 multilingual sorting.

10. The procedures `string-{up,down,fold}case` (from R6RS) apply the Unicode full uppercasing, lowercasing, and folding algorithms, respectively, to their arguments. This may cause the result to differ in length from the argument. What is more, a few characters have case-mappings that depend on the surrounding context. For example, Greek capital sigma normally downcases to Greek small sigma, but at the end of a word it downcases to Greek small final sigma instead.

    For an ASCII-only Scheme, `string-upcase` is a straightforward application of `string-map` to `char-upcase`, and `string-{down,fold}case` are straightforward applications of `string-map` to `char-downcase`.

11. The `string-ci*` procedures act as if they applied `string-foldcase` to their arguments before calling the non-`ci` versions.

12. In addition to the identifier characters of the ASCII repertoire specified by R5RS, Scheme implementations may permit any additional repertoire of Unicode characters to be employed in identifiers, provided that each character has a Unicode general category of `Lu, Ll, Lt, Lm, Lo, Mn, Mc, Me, Nd, Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So`, or `Co`, or is U+200C or U+200D (the zero-width non-joiner and joiner, respectively, which are needed for correct spelling in Persian, Hindi, and other languages). No non-Unicode characters may be used in identifiers.

13. All Scheme implementations shall permit the sequence "`\x<hexdigits>;`" to appear in Scheme identifiers. If the character with the given Unicode scalar value is supported by the implementation, identifiers containing such a sequence are equivalent to identifiers containing the corresponding character.

Note that what is said of ASCII also applies to ISO 8859-1 (Latin-1), but *not* to Windows code page 1252 or other encodings.