Technical Reports

Unicode Technical Standard #35

UNICODE LOCALE DATA MARKUP LANGUAGE (LDML) PART 5: COLLATION

Version	32
Editors	Markus Scherer (markus.icu@gmail.com) and other CLDR committee members

For the full header, summary, and status, see Part 1: Core

Summary

This document describes parts of an XML format (*vocabulary*) for the exchange of structured locale data. This format is used in the Unicode Common Locale Data Repository.

This is a partial document, describing only those parts of the LDML that are relevant for collation (sorting, searching & grouping). For the other parts of the LDML see the main LDML document and the links above.

Status

This document has been reviewed by Unicode members and other interested parties, and has been approved for publication by the Unicode Consortium. This is a stable document and may be used as reference material or cited as a normative reference by other specifications.

> ***A Unicode Technical Standard (UTS)** is an independent specification. Conformance to the Unicode Standard does not imply conformance to any UTS.*

Please submit corrigenda and other comments with the CLDR bug reporting form [Bugs]. Related information that is useful in understanding this document is found in the References. For the latest version of the Unicode Standard see [Unicode]. For a list of current Unicode Technical Reports see [Reports]. For more information about versions of the Unicode Standard, see [Versions].

Parts

The LDML specification is divided into the following parts:

Part 1: Core (languages, locales, basic structure)
Part 2: General (display names & transforms, etc.)
Part 3: Numbers (number & currency formatting)
Part 4: Dates (date, time, time zone formatting)
Part 5: Collation (sorting, searching, grouping)
Part 6: Supplemental (supplemental data)
Part 7: Keyboards (keyboard mappings)

Contents of Part 5, Collation

1 CLDR Collation

Collation is the general term for the process and function of determining the sorting order of strings of characters, for example for lists of strings presented to users, or in databases for sorting and selecting records.

Collation varies by language, by application (some languages use special phonebook sorting), and other criteria (for example, phonetic vs. visual).

CLDR provides collation data for many languages and styles. The data supports not only sorting but also language-sensitive searching and grouping under index headers. All CLDR collations are based on the [UCA] default order, with common modifications applied in the CLDR root collation, and further tailored for language and style as needed.

1.1 CLDR Collation Algorithm

The CLDR collation algorithm is an extension of the Unicode Collation Algorithm.

1.1.1 U+FFFE

U+FFFE maps to a CE with a minimal, unique primary weight. Its primary weight is not "variable": U+FFFE must not become ignorable in alternate handling. On the identical level, a minimal, unique “weight” must be emitted for U+FFFE as well. This allows for Merging Sort Keys within code point space.

For example, when sorting names in a database, a sortable string can be formed with *last_name* + '\uFFFE' + *first_name*. These strings would sort properly, without ever comparing the last part of a last name with the first part of another first name.

For backwards secondary level sorting, text *segments* separated by U+FFFE are processed in forward segment order, and *within* each segment the secondary weights are compared backwards. This is so that such combined strings are processed consistently with merging their sort keys (for example, by concatenating them level by level with a low separator).

Note: With unique, low weights on *all* levels it is possible to achieve `sortkey(str1 + "\uFFFE" + str2) == mergeSortkeys(sortkey(str1), sortkey(str2))` . When that is not necessary, then code can be a little simpler (no special handling for U+FFFE except for backwards-secondary), sort keys can be a little shorter (when using compressible common non-primary weights for U+FFFE), and another low weight can be used in tailorings.

1.1.2 Context-Sensitive Mappings

Contraction matching, as in the UCA, starts from the first character of the contraction string. It slows down processing of that first character even when none of its contractions matches. In some cases, it is preferrable to change such contractions to mappings with a prefix (context before a character), so that complex processing is done only when the less-frequently occurring trailing character is encountered.

For example, the DUCET contains contractions for several variants of L· (L followed by middle dot). Collating ASCII text is slowed down by contraction matching starting with L/l. In the CLDR root collation, these contractions are replaced by prefix mappings (L|·) which are triggered only when the middle dot is encountered. CLDR also uses prefix rules in the Japanese tailoring, for processing of Hiragana/Katakana length and iteration marks.

The mapping is conditional on the prefix match but does not change the mappings for the preceding text. As a result, a contraction mapping for "px" can be replaced by a prefix rule "p|x" only if px maps to the collation elements for p followed by the collation elements for "x if after p". In the DUCET, L· maps to CE(L) followed by a special secondary CE (which differs from CE(·) when · is not preceded by L). In the CLDR root collation, L has no context-sensitive mappings, but · maps to that special secondary CE if preceded by L.

A prefix mapping for p|x behaves mostly like the contraction px, except when there is a contraction that overlaps with the prefix, for example one for "op". A contraction matches only new text (and consumes it), while a prefix matches only already-consumed text.

- With mappings for "op" and "px", only the first contraction matches in text "opx". (It consumes the "op" characters, and there is no context-sensitive mapping for x.)
- With mappings for "op" and "p|x", both the contraction and the prefix rule match in text "opx". (The prefix always matches already-consumed characters, regardless of whether they mapped as part of contractions.)

Note: Matching of discontiguous contractions should be implemented without rewriting the text (unlike in the [UCA] algorithm specification), so that prefix matching is predictable. (It should also help with contraction matching performance.) An implementation that does rewrite the text, as in the UCA, will get different results for some (unusual) combinations of contractions, prefix rules, and input text.

Prefix matching uses a simple longest-match algorithm (op|c wins over p|c). It is recommended that prefix rules be limited to mappings where both the prefix string and the mapped string begin with an NFC boundary (that is, with a normalization starter that does not combine backwards). (In op|ch both o and c should be starters (ccc=0) and NFC_QC=Yes.) Otherwise, prefix matching would be affected by canonical reordering and discontiguous matching, like contractions. Prefix matching is thus always contiguous.

A character can have mappings with both prefixes (context before) and contraction suffixes. Prefixes are matched first. This is to keep them reasonably implementable: When there is a mapping with both a prefix and a contraction suffix (like in Japanese: ぐ|ゞ), then the matching needs to go in both directions. The contraction might involve discontiguous matching, which needs complex text iteration and handling of skipped combining marks, and will consume the matching suffix. Prefix matching should be first because, regardless of whether there is a match, the implementation will always return to the original text index (right after the prefix) from where it will start to look at all of the contractions for that prefix.

If there is a match for a prefix but no match for any of the suffixes for that prefix, then fall back to mappings with the next-longest matching prefix, and so on, ultimately to mappings with no prefix. (Otherwise mappings with longer prefixes would “hide” mappings with shorter prefixes.)

Consider the following mappings.

1. p → CE(p)
2. h → CE(h)
3. c → CE(c)
4. ch → CE(d)
5. p|c → CE(u)
6. p|ci → CE(v)
7. p|ĉ → CE(w)
8. op|ck → CE(x)

With these, text collates like this:

- pc → CE(p)CE(u)
- pci → CE(p)CE(v)
- pch → CE(p)CE(u)CE(h)
- pĉ → CE(p)CE(w)
- pĉ̣ → CE(p)CE(w)CE(U+0323) // discontiguous
- opck → CE(o)CE(p)CE(x)
- opch → CE(o)CE(p)CE(u)CE(h)

However, if the mapping p|c → CE(u) is missing, then text "pch" maps to CE(p)CE(d), "opch" maps to CE(o)CE(p)CE(d), and "pĉ̣" maps to CE(p)CE(c)CE(U+0323)CE(U+0302) (because discontiguous contraction matching extends *an existing match* by one non-starter at a time).

1.1.3 Case Handling

CLDR specifies how to sort lowercase or uppercase first, as a stronger distinction than other tertiary variants (**caseFirst**) or while completely ignoring all other tertiary distinctions (**caseLevel**). See *Section 3.3* Setting Options and *Section 3.13* Case Parameters.

1.1.4 Reordering Groups

CLDR specifies how to do parametric reordering of groups of scripts (e.g., “native script first”) as well as special groups (e.g., “digits after letters”), and provides data for the effective implementation of such reordering.

1.1.5 Combining Rules

Rules from different sources can be combined, with the later rules overriding the earlier ones. The following is an example of how this can be useful.

There is a root collation for "emoji" in CLDR. So use of "-u-co-emoji" in a Unicode locale identifier will access that ordering.

Example, using ICU:

```
collator = Collator.getInstance(ULocale.forLanguageTag("en-u-co-emoji"));
```

However, use of the emoji will supplant the language's customizations. So the above is the equivalent of:

```
collator = Collator.getInstance(ULocale.forLanguageTag("und-u-co-emoji"));
```

The same structure will not work for a language that does require customization, like Danish. That is, the following will fail.

```
collator = Collator.getInstance(ULocale.forLanguageTag("da-u-co-emoji"));
```

For that, a slightly more cumbersome method needs to be employed, which is to take the rules for Danish, and explicitly add the rules for emoji.

```
RuleBasedCollator collator = new RuleBasedCollator(
((RuleBasedCollator) Collator.getInstance(ULocale.forLanguageTag("da"))).getRules() +
((RuleBasedCollator) Collator.getInstance(ULocale.forLanguageTag("und-u-co-emoji")))
.getRules());
```

The following table shows the differences. When emoji ordering is supported, the two faces will be adjacent. When Danish ordering is supported, the ü is after the y.

code point order	,			Z	a	y	ü	☹️	✈️	글	😀
en	,	☹️	✈️	😀	a	ü	y	Z	글		
en-u-co-emoji	,	😀	☹️	✈️	a	ü	y	Z	글		
da	,	☹️	✈️	😀	a	y	**ü**	Z	글		
da-u-co-emoji	,	😀	☹️	✈️	a	**ü**	y	Z	글		
combined rules	,	😀	☹️	✈️	a	y	**ü**	Z	글		

2 Root Collation

The CLDR root collation order is based on the Default Unicode Collation Element Table (DUCET) defined in *UTS #10: Unicode Collation Algorithm* [UCA]. It is used by all other locales by default, or as the base for their tailorings. (For a chart view of the UCA, see Collation Chart [UCAChart].)

Starting with CLDR 1.9, CLDR uses modified tables for the root collation order. The root locale ordering is tailored in the following ways:

2.1 Grouping classes of characters

As of Version 6.1.0, the DUCET puts characters into the following ordering:

- First "common characters": whitespace, punctuation, general symbols, some numbers, currency symbols, and other numbers.
- Then "script characters": Latin, Greek, and the rest of the scripts.

(There are a few exceptions to this general ordering.)

The CLDR root locale modifies the DUCET tailoring by ordering the common characters more strictly by category:

- whitespace, punctuation, general symbols, currency symbols, and numbers.

What the regrouping allows is for users to parametrically reorder the groups. For example, users can reorder numbers after all scripts, or reorder Greek before Latin.

The relative order within each of these groups still matches the DUCET. Symbols, punctuation, and numbers that are grouped with a particular script stay with that script. The differences between CLDR and the DUCET order are:

1. CLDR groups the numbers together after currency symbols, instead of splitting them with some before and some after. Thus the following are put *after* currencies and just before all the other numbers.

   U+09F4 (৴) [No] BENGALI CURRENCY NUMERATOR ONE
   ...
   U+1D371 (𝍱) [No] COUNTING ROD TENS DIGIT NINE

2. CLDR handles a few other characters differently
   1. U+10A7F (𐩿) [Po] OLD SOUTH ARABIAN NUMERIC INDICATOR is put with punctuation, not symbols
   2. U+20A8 (Rs) [Sc] RUPEE SIGN and U+FDFC (﷼) [Sc] RIAL SIGN are put with currency signs, not with R and REH.

2.2 Non-variable symbols

There are multiple Variable-Weighting options in the UCA for symbols and punctuation, including *non-ignorable* and *shifted*. With the *shifted* option, almost all symbols and punctuation are ignored—except at a fourth level. The CLDR root locale ordering is modified so that symbols are not affected by the *shifted* option. That is, by default, symbols are not “variable” in CLDR. So *shifted* only causes whitespace and punctuation to be ignored, but not symbols (like ♥). The DUCET behavior can be specified with a locale ID using the "kv" keyword, to set the Variable section to include all of the symbols below it, or be set parametrically where implementations allow access.

See also:

- *Section 3.3, Setting Options*
- http://www.unicode.org/charts/collation/

2.3 Additional contractions for Tibetan

Ten contractions are added for Tibetan: Two to fulfill well-formedness condition 5, and eight more to preserve the default order for Tibetan. For details see *UTS #10, Section 3.8.2, Well-Formedness of the DUCET*.

2.4 Tailored noncharacter weights

U+FFFE and U+FFFF have special tailorings:

> **U+FFFF:** This code point is tailored to have a primary weight higher than all other characters. This allows the reliable specification of a range, such as “Sch” ≤ X ≤ “Sch\uFFFF”, to include all strings starting with "sch" or equivalent.
>
> **U+FFFE:** This code point produces a CE with minimal, unique weights on primary and identical levels. For details see the *CLDR Collation Algorithm* above.

UCA (beginning with version 6.3) also maps **U+FFFD** to a special collation element with a very high primary weight, so that it is reliably non-variable, for use with ill-formed code unit sequences.

In CLDR, so as to maintain the special collation elements, **U+FFFD..U+FFFF** are not further tailorable, and nothing can tailor to them. That is, neither can occur in a collation rule. For example, the following rules are illegal:

```
&\uFFFF < x
```

```
&x <\uFFFF
```

Note:

- Java uses an early version of this collation syntax, but has not been updated recently. It does not support any of the syntax marked with [...], and its default table is not the DUCET nor the CLDR root collation.

2.5 Root Collation Data Files

The CLDR root collation data files are in the CLDR repository and release, under the path common/uca/.

For most data files there are **_SHORT** versions available. They contain the same data but only minimal comments, to reduce the file sizes.

Comments with DUCET-style weights in files other than allkeys_CLDR.txt and allkeys_DUCET.txt use the weights defined in allkeys_CLDR.txt.

- **allkeys_CLDR** - A file that provides a remapping of UCA DUCET weights for use with CLDR.
- **allkeys_DUCET** - The same as DUCET allkeys.txt, but in alternate=non-ignorable sort order, for easier comparison with allkeys_CLDR.txt.
- **FractionalUCA** - A file that provides a remapping of UCA DUCET weights for use with CLDR. The weight values are modified:
    - The weights have variable length, with 1..4 bytes each. Each secondary or tertiary weight currently uses at most 2 bytes.
    - There are tailoring gaps between adjacent weights, so that a number of characters can be tailored to sort between any two root collation elements.
    - There are collation elements with primary weights at the boundaries between reordering groups and Unicode scripts, so that tailoring around the first or last primary of a group/script results in new collation elements that sort and reorder together with that group or script. These boundary weights also define the primary weight ranges for parametric group and script reordering.

  An implementation may modify the weights further to fit the needs of its data structures.
- **UCA_Rules** - A file that specifies the root collation order in the form of tailoring rules. This is only an approximation of the FractionalUCA data, since the rule syntax cannot express every detail of the collation elements. For example, in the DUCET and in FractionalUCA, tertiary differences are usually expressed with special tertiary weights on all collation elements of an expansion, while a typical from-rules builder will modify the tertiary weight of only one of the collation elements.
- **CollationTest_CLDR** - The CLDR versions of the CollationTest files, which use the tailorings for CLDR. For information on the format, see CollationTest.html in the UCA data directory.
    - CollationTest_CLDR_NON_IGNORABLE.txt
    - CollationTest_CLDR_SHIFTED.txt

2.6 Root Collation Data File Formats

The file formats may change between versions of CLDR. The formats for CLDR 23 and beyond are as follows. As usual, text after a # is a comment.

2.6.1 allkeys_CLDR.txt

This file defines CLDR’s tailoring of the DUCET, as described in *Section 2, Root Collation* .

The format is similar to that of allkeys.txt, although there may be some differences in whitespace.

2.6.2 FractionalUCA.txt

The format is illustrated by the following sample lines, with commentary afterwards.

```
[UCA version = 6.0.0]
```

> Provides the version number of the UCA table.

```
[Unified_Ideograph 4E00..9FCC FA0E..FA0F FA11 FA13..FA14 FA1F FA21 FA23..FA24 FA27..FA29 3400..4DB5 20000..2A6D6 2A700..2B734 2B740..2B81
```

> Lists the ranges of Unified_Ideograph characters in collation order. (New in CLDR 24.) They map to collation elements with implicit (constructed) primary weights.

```
[radical 6=亅亅:亅乚了□-□亇□予氶□-□争□□事□-承□事豫□-□□鵌□]
[radical 210=齊齊:齊□□齋齌幯𪗗齎□□□齍□-□齎□□齏□-□]
[radical 210'=齐齐:齐齑]
[radical end]
```

> Data for Unihan radical-stroke order. (New in CLDR 26.) Following the [Unified_Ideograph] line, a section of `[radical ...]` lines defines a radical-stroke order of the Unified_Ideograph characters.
>
> For Han characters, an implementation may choose either to implement the order defined in the UCA and the [Unified_Ideograph] data, or to implement the order defined by the `[radical ...]` lines. Beginning with CLDR 26, the CJK type="unihan" tailorings assume that the root collation order sorts Han characters in Unihan radical-stroke order according to the `[radical ...]` data. The CollationTest_CLDR files only

contain Han characters that are in the same relative order using implicit weights or the radical-stroke order.

The root collation radical-stroke order is derived from the first (normative) values of the Unihan kRSUnicode field for each Han character. Han characters are ordered by radical, with traditional forms sorting before simplified ones. Characters with the same radical are ordered by residual stroke count. Characters with the same radical-stroke values are ordered by block and code point, as for UCA implicit weights.

There is one `[radical ...]` line per radical, in the order of radical numbers. Each line shows the radical number and the representative characters from the UCD file CJKRadicals.txt, followed by a colon (":") and the Han characters with that radical in the order as described above. A range like 万-丌 indicates that the code points in that range sort in code point order.

The radical number and characters are informational. The sort order is established only by the order of the `[radical ...]` lines, and within each line by the characters and ranges between the colon (":") and the bracket ("]").

Each Unified_Ideograph occurs exactly once. Only Unified_Ideograph characters are listed on `[radical ...]` lines.

This section is terminated with one `[radical end]` line.

```
0000; [,,]     # Zyyy Cc       [0000.0000.0000]        * <NULL>
```

Provides a weight line. The first element (before the ";") is a hex codepoint sequence. The second field is a sequence of collation elements. Each collation element has 3 parts separated by commas: the primary weight, secondary weight, and tertiary weight. The tertiary weight actually consists of two components: the top two bits (0xC0) are used for the *case level*, and should be masked off where a case level is not used.

A weight is either empty (meaning a zero or ignorable weight) or is a sequence of one or more bytes. The bytes are interpreted as a "fraction", meaning that the ordering is 04 < 05 05 < 06. The weights are constructed so that no weight is an initial subsequence of another: that is, having both the weights 05 and 05 05 is illegal. The above line consists of all ignorable weights.

The vertical bar ("|") character is used to indicate context, as in:

```
006C | 00B7; [, DB A9, 05]
```

This example indicates that if U+00B7 appears immediately after U+006C, it is given the corresponding collation element instead. This syntax is roughly equivalent to the following contraction, but is more efficient. For details see the specification of *Context-Sensitive Mappings* above.

```
006C 00B7; CE(006C) [, DB A9, 05]
```

Single-byte primary weights are given to particularly frequent characters, such as space, digits, and a-z. More frequent characters are given two-byte weights, while relatively infrequent characters are given three-byte weights. For example:

```
...
0009; [03 05, 05, 05] # Zyyy Cc       [0100.0020.0002]        * <CHARACTER TABULATION>
...
1B60; [06 14 0C, 05, 05]    # Bali Po       [0111.0020.0002]        * BALINESE PAMENENG
...
0031; [14, 05, 05]    # Zyyy Nd       [149B.0020.0002]        * DIGIT ONE
```

The assignment of 2 vs 3 bytes does not reflect importance, or exact frequency.

```
3041; [76 06, 05, 03]   # Hira Lo       [3888.0020.000D]        * HIRAGANA LETTER SMALL A
3042; [76 06, 05, 85]   # Hira Lo       [3888.0020.000E]        * HIRAGANA LETTER A
30A1; [76 06, 05, 10]   # Kana Lo       [3888.0020.000F]        * KATAKANA LETTER SMALL A
30A2; [76 06, 05, 9E]   # Kana Lo       [3888.0020.0011]        * KATAKANA LETTER A
```

Beginning with CLDR 27, some primary or secondary collation elements may have below-common tertiary weights (e.g., `03`), in particular to allow normal Hiragana letters to have common tertiary weights.

```
# SPECIAL MAX/MIN COLLATION ELEMENTS
FFFE; [02, 05, 05]     # Special LOWEST primary, for merge/interleaving
FFFF; [EF FE, 05, 05]  # Special HIGHEST primary, for ranges
```

The two tailored noncharacters have their own primary weights.

```
F967; [U+4E0D]  # Hani Lo       [FB40.0020.0002][CE0D.0000.0000]        * CJK COMPATIBILITY IDEOGRAPH-F967
2F02; [U+4E36, 10]      # Hani So       [FB40.0020.0004][CE36.0000.0000]        * KANGXI RADICAL DOT
2E80; [U+4E36, 70, 20]  # Hani So       [FB40.0020.0004][CE36.0000.0000][0000.00FC.0004]        * CJK RADICAL REPEAT
```

Some collation elements are specified by reference to other mappings. This is particularly useful for Han characters which are given implicit/constructed primary weights; the reference to a Unified_Ideograph makes these mappings independent of implementation details. This technique may also be used in other mappings to show the relationship of character variants.

The referenced character must have a mapping listed earlier in the file, or the mapping must have been defined via the [Unified_Ideograph] data line. The referenced character must map to exactly one collation element.

`[U+4E0D]` copies U+4E0D's entire collation element. `[U+4E36, 10]` copies U+4E36's primary and secondary weights and specifies a different tertiary weight. `[U+4E36, 70, 20]` only copies U+4E36's primary weight and specifies other secondary and tertiary weights.

FractionalUCA.txt does not have any explicit mappings for implicit weights. Therefore, an implementation is free to choose an algorithm for computing implicit weights according to the principles specified in the UCA.

```
FDD1 20AC;      [0D 20 02, 05, 05]      # CURRENCY first primary
FDD1 0034;      [0E 02 02, 05, 05]      # DIGIT first primary starts new lead byte
FDD0 FF21;      [26 02 02, 05, 05]      # REORDER_RESERVED_BEFORE_LATIN first primary starts new lead byte
FDD1 004C;      [28 02 02, 05, 05]      # LATIN first primary starts new lead byte
FDD0 FF3A;      [5D 02 02, 05, 05]      # REORDER_RESERVED_AFTER_LATIN first primary starts new lead byte
FDD1 03A9;      [5F 04 02, 05, 05]      # GREEK first primary starts new lead byte (compressible)
FDD1 03E2;      [5F 60 02, 05, 05]      # COPTIC first primary (compressible)
```

These are special mappings with primaries at the boundaries of scripts and reordering groups. They serve as tailoring boundaries, so that tailoring near the first or last character of a script or group places the tailored item into the same group. Beginning with CLDR 24, each of these is a contraction of U+FDD1 with a character of the corresponding script (or of the General_Category [Z, P, S, Sc, Nd] corresponding to a special reordering group), mapping to the first possible primary weight per script or group. They can be enumerated for implementations of Collation Indexes. (Earlier versions mapped contractions with U+FDD0 to the last primary weights of each group but not each script.)

Beginning with CLDR 27, these mappings alone define the boundaries for reordering single scripts. (There are no mappings for Hrkt, Hans, or Hant because they are not fully distinct scripts; they share primary weights with other scripts: Hrkt=Hira=Kana & Hans=Hant=Hani.) There are some reserved ranges, beginning at boundaries marked with U+FDD0 plus following characters as shown above. The reserved ranges are not used for collation elements and are not available for tailoring.

Some primary lead bytes must be reserved so that reordering of scripts along partial-lead-byte boundaries can “split” the primary lead byte and use up a reserved byte. This is for implementations that write sort keys, which must reorder primary weights by offsetting them by whole lead bytes. There are reorder-reserved ranges before and after Latin, so that reordering scripts with few primary lead bytes relative to Latin can move those scripts into the reserved ranges without changing the primary weights of any other script. Each of these boundaries begins with a new two-byte primary; that is, no two groups/scripts/ranges share the top 16 bits of their primary weights.

```
FDD0 0034;      [11, 05, 05]    # lead byte for numeric sorting
```

This mapping specifies the lead byte for numeric sorting. It must be different from the lead byte of any other primary weight, otherwise numeric sorting would generate ill-formed collation elements. Therefore, this mapping itself must be excluded from the set of regular mappings. This value can be ignored by implementations that do not support numeric sorting. (Other contractions with U+FDD0 can normally be ignored altogether.)

```
# HOMELESS COLLATION ELEMENTS
FDD0 0063; [, 97, 3D]       # [15E4.0020.0004] [1844.0020.0004] [0000.0041.001F]    * U+01C6 LATIN SMALL LETTER DZ WITH CARON
FDD0 0064; [, A7, 09]       # [15D1.0020.0004] [0000.0056.0004]     * U+1DD7 COMBINING LATIN SMALL LETTER C CEDILLA
FDD0 0065; [, B1, 09]       # [1644.0020.0004] [0000.0061.0004]     * U+A7A1 LATIN SMALL LETTER G WITH OBLIQUE STROKE
```

The DUCET has some weights that don't correspond directly to a character. To allow for implementations to have a mapping for each collation element (necessary for certain implementations of tailoring), this requires the construction of special sequences for those weights. These collation elements can normally be ignored.

Next, a number of tables are defined. The function of each of the tables is summarized afterwards.

```
# VALUES BASED ON UCA
...
[first regular [0D 0A, 05, 05]] # U+0060 GRAVE ACCENT
[last regular [7A FE, 05, 05]] # U+1342E EGYPTIAN HIEROGLYPH AA032
[first implicit [E0 04 06, 05, 05]] # CONSTRUCTED
[last implicit [E4 DF 7E 20, 05, 05]] # CONSTRUCTED
[first trailing [E5, 05, 05]] # CONSTRUCTED
[last trailing [E5, 05, 05]] # CONSTRUCTED
...
```

This table summarizes ranges of important groups of characters for implementations.

```
# Top Byte => Reordering Tokens
[top_byte     00      TERMINATOR ]    #       [0]     TERMINATOR=1
[top_byte     01      LEVEL-SEPARATOR ]       #       [0]     LEVEL-SEPARATOR=1
[top_byte     02      FIELD-SEPARATOR ]       #       [0]     FIELD-SEPARATOR=1
[top_byte     03      SPACE ] #       [9]     SPACE=1 Cc=6 Zl=1 Zp=1 Zs=1
...
```

This table defines the reordering groups, for script reordering. The table maps from the first bytes of the fractional weights to a reordering token. The format is "[top_byte " byte-value reordering-token "COMPRESS"? "]". The "COMPRESS" value is present when there is only one byte in the reordering token, and primary-weight compression can be applied. Most reordering tokens are script values; others are special-purpose values, such as PUNCTUATION. Beginning with CLDR 24, this table precedes the regular mappings, so that parsers can use this information while processing and optimizing mappings. Beginning with CLDR 27, most of this data is irrelevant because single scripts can be reordered. Only the "COMPRESS" data is still useful.

```
# Reordering Tokens => Top Bytes
[reorderingTokens     Arab    61=910 62=910 ]
[reorderingTokens     Armi    7A=22 ]
[reorderingTokens     Armn    5F=82 ]
[reorderingTokens     Avst    7A=54 ]
...
```

This table is an inverse mapping from reordering token to top byte(s). In terms like "61=910", the first value is the top byte, while the second is informational, indicating the number of primaries assigned with that top byte.

```
# General Categories => Top Byte
[categories   Cc      03{SPACE}=6 ]
[categories   Cf      77{Khmr Tale Talu Lana Cham Bali Java Mong Olck Cher Cans Ogam Runr Orkh Vaii Bamu}=2 ]
[categories   Lm      0D{SYMBOL}=25 0E{SYMBOL}=22 27{Latn}=12 28{Latn}=12 29{Latn}=12 2A{Latn}=12...
```

This table is informational, providing the top bytes, scripts, and primaries associated with each general category value.

```
# FIXED VALUES
[fixed first implicit byte E0]
[fixed last implicit byte E4]
[fixed first trail byte E5]
[fixed last trail byte EF]
[fixed first special byte F0]
[fixed last special byte FF]

[fixed secondary common byte 05]
```

```
[fixed last secondary common byte 45]
[fixed first ignorable secondary byte 80]

[fixed tertiary common byte 05]
[fixed first ignorable tertiary byte 3C]
```

The final table gives certain hard-coded byte values. The "trail" area is provided for implementation of the "trailing weights" as described in the UCA.

Note: The particular primary lead bytes for Hani vs. IMPLICIT vs. TRAILING are only an example. An implementation is free to move them if it also moves the explicit TRAILING weights. This affects only a small number of explicit mappings in FractionalUCA.txt, such as for U+FFFD, U+FFFF, and the “unassigned first primary”. It is possible to use no SPECIAL bytes at all, and to use only the one primary lead byte FF for TRAILING weights.

2.6.3 UCA_Rules.txt

The format for this file uses the CLDR collation syntax, see *Section 3, Collation Tailorings* .

3 Collation Tailorings

```
<!ELEMENT collations (alias | (defaultCollation?, collation*, special*)) >
```

```
<!ELEMENT defaultCollation ( #PCDATA ) >
```

This element of the LDML format contains one or more **collation** elements, distinguished by type. Each **collation** contains elements with parametric settings, or rules that specify a certain sort order, as a tailoring of the root order, or both.

Note: CLDR collation tailoring data should follow the CLDR Collation Guidelines.

3.1 Collation Types

Each locale may have multiple sort orders (types). The **defaultCollation** element defines the default tailoring for a locale and its sublocales. For example:

- root.xml: `<defaultCollation>standard</defaultCollation>`
- zh.xml: `<defaultCollation>pinyin</defaultCollation>`
- zh_Hant.xml: `<defaultCollation>stroke</defaultCollation>`

To allow implementations in reduced memory environments to use CJK sorting, there are also short forms of each of these collation sequences. These provide for the most common characters in common use, and are marked with **alt**="**short**".

A collation type name that starts with "private-", for example, "private-kana", indicates an incomplete tailoring that is only intended for import into one or more other tailorings (usually for sharing common rules). It does not establish a complete sort order. An implementation should not build data tables for a private collation type, and should not include a private collation type in a list of available types.

Note:

- There is an on-line demonstration of collation at [LocaleExplorer] that uses the same rule syntax. (Pick the locale and scroll to "Collation Rules", near the end.)
- In CLDR 23 and before, LDML collation files used an XML format. Starting with CLDR 24, the XML collation syntax is deprecated and no longer used. See the *CLDR 23 version of this document* for details about the XML collation syntax.

3.1.1 Collation Type Fallback

When loading a requested tailoring from its data file and the parent file chain, use the following type fallback to find the tailoring.

1. Determine the default type from the <defaultCollation> element; map the default type to its alias if one is defined. If there is no <defaultCollation> element, then use "standard" as the default type.
2. If the request language tag specifies the collation type (keyword "co"), then map it to its alias if one is defined (e.g., "-co-phonebk" → "phonebook"). If the language tag does not specify the type, then use the default type.
3. Use the <collation> element with this type.
4. If it does not exist, and the type starts with "search" but is longer, then set the type to "search" and use that <collation> element. (For example, "searchjl" → "search".)
5. If it does not exist, and the type is not the default type, then set the type to the default type and use that <collation> element.
6. If it does not exist, and the type is not "standard", then set the type to "standard" and use that <collation> element.
7. If it does not exist, then use the CLDR root collation.

Note that the CLDR collation/root.xml contains <defaultCollation>standard</defaultCollation>, <collation type="standard"> (with an empty tailoring, so this is the same as the CLDR root collation), and <collation type="search">.

For example, assume that we have collation data for the following tailorings. ("da/search" is shorthand for "da-u-co-search".)

- root/defaultCollation=standard
- root/standard (this is the same as “the CLDR root collator”)
- root/search
- da/standard
- da/search
- el/standard
- ko/standard

- ko/search
- ko/searchjl
- zh/defaultCollation=pinyin
- zh/pinyin
- zh/stroke
- zh-Hant/defaultCollation=stroke

Sample requested and actual collation locales and types

requested	actual	comment
da/phonebook	da/standard	default type for Danish
zh	zh/pinyin	default type for zh
zh/standard	root/standard	no "standard" tailoring for zh, falls back to root
zh/phonebook	zh/pinyin	default type for zh
zh-Hant/phonebook	zh/stroke	default type for zh-Hant is "stroke"
da/searchjl	da/search	"search.+" falls back to "search"
el/search	root/search	no "search" tailoring for Greek
el/searchjl	root/search	"search.+" falls back to "search", found in root
ko/searchjl	ko/searchjl	requested data is actually available

3.2 Version

The version attribute is used in case a specific version of the UCA is to be specified. It is optional, and is specified if the results are to be identical on different systems. If it is not supplied, then the version is assumed to be the same as the Unicode version for the system as a whole.

> **Note:** For version 3.1.1 of the UCA, the version of Unicode must also be specified with any versioning information; an example would be "3.1.1/3.2" for version 3.1.1 of the UCA, for version 3.2 of Unicode. This was changed by decision of the UTC, so that dual versions were no longer necessary. So for UCA 4.0 and beyond, the version just has a single number.

3.3 Collation Element

```
<!ELEMENT collation (alias | (cr*, special*)) >
```

The tailoring syntax is designed to be independent of the actual weights used in any particular UCA table. That way the same rules can be applied to UCA versions over time, even if the underlying weights change. The following illustrates the overall structure of a collation:

```
<collation type="phonebook">
  <cr><![CDATA[
    [caseLevel on]
    &c < k
  ]]></cr>
</collation>
```

3.4 Setting Options

Parametric settings can be specified in language tags or in rule syntax (in the form `[keyword value]`). For example, `-ks-level2` or `[strength 2]` will only compare strings based on their primary and secondary weights.

If a setting is not present, the CLDR default (or the default for the locale, if there is one) is used. That default is listed in bold italics. Where there is a UCA default that is different, it is listed in bold with (**UCA default**). Note that the default value for a locale may be different than the normal default value for the setting.

Collation Settings

BCP47 Key	BCP47 Value	Rule Syntax	Description
ks	level1	`[strength 1]` (primary)	Sets the default strength for comparison, as described in the [UCA]. *Note that a strength setting of greater than 4 may have the same effect as **identical**, depending on the locale and implementation.*
	level2	`[strength 2]` (secondary)	
	level3	***`[strength 3]`* (tertiary)**	
	level4	`[strength 4]` (quaternary)	
	identic	`[strength I]` (identical)	
ka	noignore	***`[alternate non-ignorable]`***	Sets alternate handling for variable weights, as described in [UCA], where "shifted" causes certain characters to be ignored in comparison. *The default for LDML is different than it is in the UCA. In LDML, the default for alternate handling is **non-ignorable**, while in UCA it is **shifted**. In addition, in LDML only whitespace and punctuation are variable by default.*
	shifted	**`[alternate shifted]` (UCA default)**	
	n/a	*n/a* (blanked)	
kb	true	`[backwards 2]`	Sets the comparison for the second level to be **backwards**, as described in [UCA].
	false	***n/a***	

kk	true	**[normalization on]** **(UCA default)**	If **on**, then the normal [UCA] algorithm is used. If **off**, then most strings should still sort correctly despite not normalizing to NFD first. *Note that the default for CLDR locales may be different than in the UCA. The rules for particular locales have it set to **on**: those locales whose exemplar characters (in forms commonly interchanged) would be affected by normalization.*
	false	***[normalization off]***	
kc	true	[caseLevel on]	If set to **on**, a level consisting only of case characteristics will be inserted in front of tertiary level, as a "Level 2.5". To ignore accents but take case into account, set strength to **primary** and case level to **on**. For details, see *Section 3.14, Case Parameters* .
	false	***[caseLevel off]***	
kf	upper	[caseFirst upper]	If set to **upper**, causes upper case to sort before lower case. If set to **lower**, causes lower case to sort before upper case. Useful for locales that have already supported ordering but require different order of cases. Affects case and tertiary levels. For details, see *Section 3.14, Case Parameters* .
	lower	[caseFirst lower]	
	false	***[caseFirst off]***	
kh	true ***Deprecated:*** Use rules with quaternary relations instead.	[hiraganaQ on]	Controls special treatment of Hiragana code points on quaternary level. If turned **on**, Hiragana codepoints will get lower values than all the other non-variable code points in **shifted**. That is, the normal Level 4 value for a regular collation element is FFFF, as described in [UCA], *Section 3.6, Variable Weighting* . This is changed to FFFE for [:script=Hiragana:] characters. The strength must be greater or equal than quaternary if this attribute is to have any effect.
	false	***[hiraganaQ off]***	
kn	true	[numericOrdering on]	If set to **on**, any sequence of Decimal Digits (General_Category = Nd in the [UAX44]) is sorted at a primary level with its numeric value. For example, "A-21" < "A-123". The computed primary weights are all at the start of the **digit** reordering group. Thus with an untailored UCA table, "a$" < "a0" < "a2" < "a12" < "a⓪" < "aa".
	false	***[numericOrdering off]***	
kr	a sequence of one or more reorder codes: **space, punct, symbol, currency, digit**, or any BCP47 script ID	[reorder Grek digit]	Specifies a reordering of scripts or other significant blocks of characters such as symbols, punctuation, and digits. For the precise meaning and usage of the reorder codes, see *Section 3.13, Collation Reordering.*
kv	space	[maxVariable space]	Sets the variable top to the top of the specified reordering group. All code points with primary weights less than or equal to the variable top will be considered variable, and thus affected by the alternate handling. Variables are ignorable by default in [UCA], but not in CLDR.
	punct	***[maxVariable punct]***	
	symbol	**[maxVariable symbol]** **(UCA default)**	
	currency	[maxVariable currency]	
vt	See *Part 1 Section 3.6.4, U Extension Data Files.* ***Deprecated:*** Use maxVariable instead.	&\u00XX\uYYYY < [variable top] (the default is set to the highest punctuation, thus including spaces and punctuation, but not symbols)	The BCP47 value is described in *Appendix Q: Locale Extension Keys and Types.* Sets the string value for the variable top. All the code points with primary weights less than or equal to the variable top will be considered variable, and thus affected by the alternate handling. An implementation that supports the variableTop setting should also support the maxVariable setting, and it should "pin" ("round up") the variableTop to the top of the containing reordering group. Variables are ignorable by default in [UCA], but not in CLDR. See below for more information.
n/a	*n/a*	*n/a*	match-boundaries: ***none*** | whole-character | whole-word Defined by *Section 8, Searching and Matching* of [UCA].
n/a	*n/a*	*n/a*	match-style: ***minimal*** | medial | maximal Defined by *Section 8, Searching and Matching* of [UCA].

3.4.1 Common settings combinations

Some commonly used parametric collation settings are available via combinations of LDML settings attributes:

- “Ignore accents”: **strength=primary**
- “Ignore accents” but take case into account: **strength=primary caseLevel=on**
- “Ignore case”: **strength=secondary**
- “Ignore punctuation” (completely): **strength=tertiary alternate=shifted**
- “Ignore punctuation” but distinguish among punctuation marks: **strength=quaternary alternate=shifted**

3.4.2 Notes on the normalization setting

The UCA always normalizes input strings into NFD form before the rest of the algorithm. However, this results in poor performance.

With **normalization=off**, strings that are in [FCD] and do not contain Tibetan precomposed vowels (U+0F73, U+0F75, U+0F81) should sort correctly. With **normalization=on**, an implementation that does not normalize to NFD must at least perform an incremental FCD check and normalize substrings as necessary. It should also always decompose the Tibetan precomposed vowels. (Otherwise discontiguous contractions across their leading components cannot be handled correctly.)

Another complication for an implementation that does not always use NFD arises when contraction mappings overlap with canonical

Decomposition_Mapping strings. For example, the Danish contraction “aa” overlaps with the decompositions of ‘ä’, ‘å’, and other characters. In the root collation (and in the DUCET), Cyrillic ‘ӛ’ maps to a single collation element, which means that its decomposition “ə+◌̈” forms a contraction, and its second character (U+0308) is the same as the first character in the Decomposition_Mapping of U+0344 ‘◌̈́’=“◌̈+◌́”.

In order to handle strings with these characters (e.g., “aä” and “ə̈́” [which are in FCD]) exactly as with prior NFD normalization, an implementation needs to either add overlap contractions to its data (e.g., “a+ä” and “ə+◌̈́”), or it needs to decompose the relevant composites (e.g., ‘ä’ and ‘◌̈́’) as soon as they are encountered.

3.4.3 Notes on variable top settings

Users may want to include more or fewer characters as Variable. For example, someone could want to restrict the Variable characters to just include space marks. In that case, maxVariable would be set to "space". (In CLDR 24 and earlier, the now-deprecated variableTop would be set to U+1680, see the “Whitespace” UCA collation chart). Alternatively, someone could want more of the Common characters in them, and include characters up to (but not including) '0', by setting maxVariable to "currency". (In CLDR 24 and earlier, the now-deprecated variableTop would be set to U+20BA, see the “Currency-Symbol” collation chart).

The effect of these settings is to customize to ignore different sets of characters when comparing strings. For example, the locale identifier "de-u-ka-shifted-kv-currency" is requesting settings appropriate for German, including German sorting conventions, and that currency symbols and characters sorting below them are ignored in sorting.

3.5 Collation Rule Syntax

```
<!ELEMENT cr #PCDATA >
```

The goal for the collation rule syntax is to have clearly expressed rules with a concise format. The CLDR rule syntax is a subset of the [ICUCollation] syntax.

For the CLDR root collation, the FractionalUCA.txt file defines all mappings for all of Unicode directly, and it also provides information about script boundaries, reordering groups, and other details. For tailorings, this is neither necessary nor practical. In particular, while the root collation sort order rarely changes for existing characters, their numeric collation weights change with every version. If tailorings also specified numeric weights directly, then they would have to change with every version, parallel with the root collation. Instead, for tailorings, mappings are added and modified relative to the root collation. (There is no syntax to *remove* mappings, except via special [suppressContractions [...]] .)

The ASCII [:P:] and [:S:] characters are reserved for collation syntax: `[\u0021-\u002F \u003A-\u0040 \u005B-\u0060 \u007B-\u007E]`

Unicode Pattern_White_Space characters between tokens are ignored. Unquoted white space terminates reset and relation strings.

A pair of ASCII apostrophes encloses quoted literal text. They are normally used to enclose a syntax character or white space, or a whole reset/relation string containing one or more such characters, so that those are parsed as part of the reset/relation strings rather than treated as syntax. A pair of immediately adjacent apostrophes is used to encode one apostrophe.

Code points can be escaped with `\uhhhh` and `\U00hhhhhh` escapes, as well as common escapes like `\t` and `\n` . (For details see the documentation of ICU UnicodeString::unescape().) This is particularly useful for default-ignorable code points, combining marks, visually indistinct variants, hard-to-type characters, etc. These sequences are unescaped before the rules are parsed; this means that even escaped syntax and white space characters need to be enclosed in apostrophes. For example: `&'\u0020'='\u3000'`

The ASCII double quote must be both escaped (so that the collation syntax can be enclosed in pairs of double quotes in programming environments) and quoted. For example: `&'\u0022'<<<x`

Comments are allowed at the beginning, and after any complete reset, relation, setting, or command. A comment begins with a `#` and extends to the end of the line (according to the Unicode Newline Guidelines).

The collation syntax is case-sensitive.

3.6 Orderings

The root collation mappings form the initial state. Mappings are added and removed via a sequence of rule chains. Each tailoring rule builds on the current state after all of the preceding rules (and is not affected by any following rules). Rule chains may alternate with comments, settings, and special commands.

A rule chain consists of a reset followed by one or more relations. The reset position is a string which maps to one or more collation elements according to the current state. A relation consists of an operator and a string; it maps the string to the current collation elements, modified according to the operator.

Specifying Collation Ordering

Relation Operator	Example	Description
`&`	`& Z`	Map Z to collation elements according to the current state. These will be modified according to the following relation operators and then assigned to the corresponding relation strings.

`<`	`& a`
	`< b`
		Make 'b' sort after 'a', as a *primary* (base-character) difference

`<<`	`& a`
	`<< ä`
		Make 'ä' sort after 'a' as a *secondary* (accent) difference

`<<<`	`& a`
	`<<< A`
		Make 'A' sort after 'a' as a *tertiary* (case/variant) difference

`<<<<`	`& か`
	`<<<< カ`
		Make 'カ' (Katakana Ka) sort after 'か' (Hiragana Ka) as a *quaternary* difference

`=`	`& v`
	`= w`
		Make 'w' sort *identically* to 'v'

The following shows the result of serially applying three rules.

	Rules	Result	Comment
1	& a < g	... a $<_1$ g ...	Put g after a.
2	& a < h < k	... a $<_1$ h $<_1$ k $<_1$ g ...	Now put h and k after a (inserting before the g).

3	& h << g	... a $<_1$ h $<_1$ g $<_1$ k ...	Now put g after h (inserting before k).

Notice that relation strings can occur multiple times, and thus override previous rules.

Each relation uses and modifies the collation elements of the immediately preceding reset position or relation. A rule chain with two or more relations is equivalent to a sequence of “atomic rules” where each rule chain has exactly one relation, and each relation is followed by a reset to this same relation string.

Example:

Rules	Equivalent Atomic Rules
& b < q <<< Q & a < x <<< X << q <<< Q < z	& b < q & q <<< Q & a < x & x <<< X & X << q & q <<< Q & Q < z

This is not always possible because prefix and extension strings can occur in a relation but not in a reset (see below).

The relation operator `=` maps its relation string to the current collation elements. Any other relation operator modifies the current collation elements as follows.

- Find the *last* collation element whose strength is at least as great as the strength of the operator. For example, for `<<` find the last primary or secondary CE. This CE will be modified; all following CEs should be removed. If there is no such CE, then reset the collation elements to a single completely-ignorable CE.
- Increment the collation element weight corresponding to the strength of the operator. For example, for `<<` increment the secondary weight.
- The new weight must be less than the next weight for the same combination of higher-level weights of any collation element according to the current state.
- Weights must be allocated in accordance with the UCA well-formedness conditions.
- When incrementing any weight, lower-level weights should be reset to the “common” values, to help with sort key compression.

In all cases, even for `=` , the case bits are recomputed according to *Section 3.13, Case Parameters*. (This can be skipped if an implementation does not support the caseLevel or caseFirst settings.)

For example, `&ae<x` maps ‘x’ to two collation elements. The first one is the same as for ‘a’, and the second one has a primary weight between those for ‘e’ and ‘f’. As a result, ‘x’ sorts between “ae” and “af”. (If the primary of the first collation element was incremented instead, then ‘x’ would sort after “az”. While also sorting primary-after “ae” this would be surprising and sub-optimal.)

Some additional operators are provided to save space with large tailorings. The addition of a * to the relation operator indicates that each of the following single characters are to be handled as if they were separate relations with the corresponding strength. Each of the following single characters must be NFD-inert, that is, it does not have a canonical decomposition and it does not reorder (ccc=0). This keeps abbreviated rules unambiguous.

A starred relation operator is followed by a sequence of characters with the same quoting/escaping rules as normal relation strings. Such a sequence can also be followed by one or more pairs of ‘-’ and another sequence of characters. The single characters adjacent to the ‘-’ establish a code point order range. The same character cannot be both the end of a range and the start of another range. (For example, `<a-d-g` is not allowed.)

Abbreviating Ordering Specifications

Relation Operator	Example	Equivalent

`<*`	`& a`
	`<* bcd-gp-s`
		`& a`
		`< b < c < d < e < f < g < p < q < r < s`

`<<*`	`& a`
	`<<* æąɐ`
		`& a`
		`<< æ << ą << ɐ`

`<<<*`	`& p`
	`<<<* ᴘpP`
		`& p`
		`<<< ᴘ <<< p <<< P`

`<<<<*`	`& k`
	`<<<<* qQ`
		`& k`
		`<<<< q <<<< Q`

`=*`	`& v`
	`=* VwW`
		`& v`
		`= V = w = W`

3.7 Contractions

A multi-character relation string defines a contraction.

Specifying Contractions

Example	Description

`& k`
`< ch`
	Make the sequence 'ch' sort after 'k', as a primary (base-character) difference

3.8 Expansions

A mapping to multiple collation elements defines an expansion. This is normally the result of a reset position (and/or preceding relation) that yields multiple collation elements, for example `&ae<x` or `&æ<y` .

A relation string can also be followed by `/` and an *extension string*. The extension string is mapped to collation elements according to the current state, and the relation string is mapped to the concatenation of the regular CEs and the extension CEs. The extension CEs are not modified, not even their case bits. The extension CEs are *not* retained for following relations.

For example, `&a<z/e` maps ‘z’ to an expansion similar to `&ae<x` . However, the first CE of ‘z’ is primary-after that of ‘a’, and the second CE is exactly that of ‘e’, which yields the order ae < x < af < ag < ... < az < z < b.

The choice of reset-to-expansion vs. use of an extension string can be exploited to affect contextual mappings. For example, `&L·=x` yields a second CE for ‘x’ equal to the context-sensitive middle-dot-after-L (which is a secondary CE in the root collation). On the other hand, `&L=x/·` yields a second CE of the middle dot by itself (which is a primary CE).

The two ways of specifying expansions also differ in how case bits are computed. When some of the CEs are copied verbatim from an extension string, then the relation string’s case bits are distributed over a smaller number of normal CEs. For example, `&aE=Ch` yields an uppercase CE and a lowercase CE, but `&a=Ch/E` yields a mixed-case CE (for ‘C’ and ‘h’ together) followed by an uppercase CE (copied from ‘E’).

In summary, there are two ways of specifying expansions which produce subtly different mappings. The use of extension strings is unusual but sometimes necessary.

3.9 Context Before

A relation string can have a prefix (context before) which makes the mapping from the relation string to its tailored position conditional on the string occurring after that prefix. For details see the specification of *Context-Sensitive Mappings*.

For example, suppose that "-" is sorted like the previous vowel. Then one could have rules that take "a-", "e-", and so on. However, that means that every time a very common character (a, e, ...) is encountered, a system will slow down as it looks for possible contractions. An alternative is to indicate that when "-" is encountered, and it comes after an 'a', it sorts like an 'a', and so on.

Specifying Previous Context

Rules

`& a <<< a | '-'`
`& e <<< e | '-'`
`...`

Both the prefix and extension strings can occur in a relation. For example, the following are allowed:

- `< abc | def / ghi`
- `< def / ghi`
- `< abc | def`

3.10 Placing Characters Before Others

There are certain circumstances where characters need to be placed before a given character, rather than after. This is the case with Pinyin, for example, where certain accented letters are positioned before the base letter. That is accomplished with the following syntax.

```
&[before 2] a << à
```

The before-strength can be 1 (primary), 2 (secondary), or 3 (tertiary).

It is an error if the strength of the reset-before differs from the strength of the immediately following relation. Thus the following are errors.

- `&[before 2] a < à # error`
- `&[before 2] a <<< à # error`

3.11 Logical Reset Positions

The CLDR table (based on UCA) has the following overall structure for weights, going from low to high.

Specifying Logical Positions

Name	Description	UCA Examples
first tertiary ignorable ... last tertiary ignorable	p, s, t = ignore	Control Codes Format Characters Hebrew Points Tibetan Signs ...
first secondary ignorable ... last secondary ignorable	p, s = ignore	None in UCA
first primary ignorable ... last primary ignorable	p = ignore	Most combining marks

first variable
...
last variable
	***if** alternate = non-ignorable*
	p != ignore,
	***if** alternate = shifted*
	p, s, t = ignore
		Whitespace,
		Punctuation

first regular ... last regular	p != ignore	General Symbols Currency Symbols Numbers Latin Greek

		...
first implicit ... last implicit	p != ignore, assigned automatically	CJK, CJK compatibility (those that are not decomposed) CJK Extension A, B, C, ... Unassigned
first trailing ... last trailing	p != ignore, used for trailing syllable components	Jamo Trailing Jamo Leading U+FFFD U+FFFF

Each of the above Names can be used with a reset to position characters relative to that logical position. That allows characters to be ordered before or after a *logical* position rather than a specific character.

> **Note:** The reason for this is so that tailorings can be more stable. A future version of the UCA might add characters at any point in the above list. Suppose that you set character X to be after Y. It could be that you want X to come after Y, no matter what future characters are added; or it could be that you just want Y to come after a given logical position, for example, after the last primary ignorable.

Each of these special reset positions always maps to a single collation element.

Here is an example of the syntax:

```
& [first tertiary ignorable] << à
```

For example, to make a character be a secondary ignorable, one can make it be immediately after (at a secondary level) a specific character (like a combining diaeresis), or one can make it be immediately after the last secondary ignorable.

Each special reset position adjusts to the effects of preceding rules, just like normal reset position strings. For example, if a tailoring rule creates a new collation element after `&[last variable]` (via explicit tailoring after that, or via tailoring after the relevant character), then this new CE becomes the new *last variable* CE, and is used in following resets to `[last variable]` .

[first variable] and [first regular] and [first trailing] should be the first real such CEs (e.g., CE(U+0060 `)), as adjusted according to the tailoring, not the boundary CEs (see the FractionalUCA.txt “first primary” mappings starting with U+FDD1).

`[last regular]` is not actually the last normal CE with a primary weight before implicit primaries. It is used to tailor large numbers of characters, usually CJK, into the script=Hani range between the last regular script and the first implicit CE. (The first group of implicit CEs is for Han characters.) Therefore, `[last regular]` is set to the first Hani CE, the artificial script boundary CE at the beginning of this range. For example: `&[last regular]<*亜唖娃阿...`

The [last trailing] is the CE of U+FFFF. Tailoring to that is not allowed.

The `[last variable]` indicates the "highest" character that is treated as punctuation with alternate handling.

The value can be changed by using the maxVariable setting. This takes effect, however, after the rules have been built, and does not affect any characters that are reset relative to the `[last variable]` value when the rules are being built. The maxVariable setting might also be changed via a runtime parameter. That also does not affect the rules.
(In CLDR 24 and earlier, the variable top could also be set by using a tailoring rule with `[variable top]` in the place of a relation string.)

3.12 Special-Purpose Commands

The import command imports rules from another collation. This allows for better maintenance and smaller rule sizes. The source is a BCP 47 language tag with an optional collation type but without other extensions. The collation type is the BCP 47 form of the collation type in the source; it defaults to "standard".

Examples:

- `[import de-u-co-phonebk]` (not "...-co-phonebook")
- `[import und-u-co-search]` (not "root-...")
- `[import ja-u-co-private-kana]` (language "ja" required even when this import itself is in another "ja" tailoring.)

Special-Purpose Elements

Rule Syntax
[suppressContractions [Љ-ґ]]
[optimize [Ά-ώ]]

The *suppress contractions* tailoring command turns off any existing contractions that begin with those characters, as well as any prefixes for those characters. It is typically used to turn off the Cyrillic contractions in the UCA, since they are not used in many languages and have a considerable performance penalty. The argument is a Unicode Set.

The *suppress contractions* command has immediate effect on the current set of mappings, including mappings added by preceding rules. Following rules are processed after removing any context-sensitive mappings originating from any of the characters in the set.

The *optimize* tailoring command is purely for performance. It indicates that those characters are sufficiently common in the target language for the tailoring that their performance should be enhanced.

The reason that these are not settings is so that their contents can be arbitrary characters.

Example:

The following is a simple example that combines portions of different tailorings for illustration. For more complete examples, see the actual locale data: Japanese, Chinese, Swedish, and German (type="phonebook") are particularly illustrative.

```
<collation>
  <cr><![CDATA[
    [caseLevel on]
    &Z
    < æ <<< Æ
    < å <<< Å <<< aa <<< aA <<< Aa <<< AA
    < ä <<< Ä
    < ö <<< Ö << ű <<< Ű
    < ő <<< Ő << ø <<< Ø
    &V <<<* wW
    &Y <<<* üÜ
    &[last non-ignorable]
    # The following is equivalent to <亜<唖<娃...
    <* 亜唖娃阿哀愛挨姶逢葵茜穐悪握渥旭葦芦
    <* 鯵梓圧斡扱
  ]]></cr>
</collation>
```

3.13 Collation Reordering

Collation reordering allows scripts and certain other defined blocks of characters to be moved relative to each other parametrically, without changing the detailed rules for all the characters involved. This reordering is done on top of any specific ordering rules within the script or block currently in effect. Reordering can specify groups to be placed at the start and/or the end of the collation order. For example, to reorder Greek characters before Latin characters, and digits afterwards (but before other scripts), the following can be used:

Rule Syntax	Locale Identifier
`[reorder Grek Latn digit]`	`en-u-kr-grek-latn-digit`

In each case, a sequence of ***reorder_codes*** is used, separated by spaces in the settings attribute and in rule syntax, and by hyphens in locale identifiers.

A ***reorder_code*** is any of the following special codes:

1. **space, punct, symbol, currency, digit** - core groups of characters below 'a'
2. **any script code** except **Common** and **Inherited**.
    - Some pairs of scripts sort primary-equal and always reorder together. For example, Katakana characters are are always reordered with Hiragana.
3. **others** - where all codes not explicitly mentioned should be ordered. The script code **Zzzz** (Unknown Script) is a synonym for **others**.

It is an error if a code occurs multiple times.

It is an error if the sequence of reorder codes is empty in the XML attribute or in the locale identifier. Some implementations may interpret an empty sequence in the `[reorder]` rule syntax as a reset to the DUCET ordering, synonymous with `[reorder others]` ; other implementations may forbid an empty sequence in the rule syntax as well.

Interaction with **alternate=shifted**: Whether a primary weight is “variable” is determined according to the “variable top”, before applying script reordering. Once that is determined, script reordering is applied to the primary weight regardless of whether it is “regular” (used in the primary level) or “shifted” (used in the quaternary level).

3.13.1 Interpretation of a reordering list

The reordering list is interpreted as if it were processed in the following way.

1. If any core code is not present, then it is inserted at the front of the list in the order given above.
2. If the **others** code is not present, then it is inserted at the end of the list.
3. The **others** code is replaced by the list of all script codes not explicitly mentioned, in DUCET order.
4. The reordering list is now complete, and used to reorder characters in collation accordingly.

The locale data may have a particular ordering. For example, the Czech locale data could put digits after all letters, with `[reorder others digit]` . Any reordering codes specified on top of that (such as with a bcp47 locale identifier) completely replace what was there. To specify a version of collation that completely resets any existing reordering to the DUCET ordering, the single code **Zzzz** or **others** can be used, as below.

Examples:

Locale Identifier	Effect
`en-u-kr-latn-digit`	Reorder digits after Latin characters (but before other scripts like Cyrillic).
`en-u-kr-others-digit`	Reorder digits after all other characters.
`en-u-kr-arab-cyrl-others-symbol`	Reorder Arabic characters first, then Cyrillic, and put symbols at the end—after all other characters.
`en-u-kr-others`	Remove any locale-specific reordering, and use DUCET order for reordering blocks.

The default reordering groups are defined by the FractionalUCA.txt file, based on the primary weights of associated collation elements. The file contains special mappings for the start of each group, script, and reorder-reserved range, see *Section 2.6.2, FractionalUCA.txt*.

There are some special cases:

- The **Hani** group includes implicit weights for *Han characters* according to the UCA as well as any characters tailored relative to a Han character, or after `&[first Hani]`.
- Implicit weights for *unassigned code points* according to the UCA reorder as the last weights in the **others** (**Zzzz**) group.
  There is no script code to explicitly reorder the unassigned-implicit weights into a particular position. (Unassigned-implicit weights are used for non-Hani code points without any mappings. For a given Unicode version they are the code points with General_Category values Cn, Co, Cs.)
- The TRAILING group, the FIELD-SEPARATOR (associated with U+FFFE), and collation elements with only zero primary weights are not reordered.
- The TERMINATOR, LEVEL-SEPARATOR, and SPECIAL groups are never associated with characters.

For example, `reorder="Hani Zzzz Grek"` sorts Hani, Latin, Cyrillic, ... (all other scripts) ..., unassigned, Greek, TRAILING.

Notes for implementations that write sort keys:

- Primaries must always be offset by one or more whole primary lead bytes. (Otherwise the number of bytes in a fractional weight may change, compressible scripts may span multiple lead bytes, or trailing primary bytes may collide with separators and primary-compression terminators.)
- When a script is reordered that does not start and end on whole-primary-lead-byte boundaries, then the lead byte needs to be "split", and a reserved byte is used up. The data supports this via reorder-reserved ranges of primary weights that are not used for collation elements.
- Primary weights from different original lead bytes can be reordered to a shared lead byte, as long as they do not overlap. Primary compression ends when the target lead byte differs or when the original lead byte of the next primary is not compressible.
- Non-compressible groups and scripts begin or end on whole-primary-lead-byte boundaries (or both), so that reordering cannot surround a non-compressible script by two compressible ones within the same target lead byte. This is so that primary compression can be terminated reliably (choosing the low or high terminator byte) simply by comparing the previous and current primary weights. Otherwise it would have to also check for another condition (e.g., equal scripts).

3.13.2 Reordering Groups for allkeys.txt

For allkeys_CLDR.txt, the start of each reordering group can be determined from FractionalUCA.txt, by finding the first real mapping (after "xyz first primary") of that group (e.g., `0060; [0D 07, 05, 05] # Zyyy Sk [0312.0020.0002] * GRAVE ACCENT`), and looking for that mapping's character sequence (`0060`) in allkeys_CLDR.txt. The comment in FractionalUCA.txt (`[0312.0020.0002]`) also shows the allkeys_CLDR.txt collation elements.

The DUCET ordering of some characters is slightly different from the CLDR root collation order. The reordering groups for the DUCET are not specified. The following describes how reordering groups for the DUCET can be derived.

For allkeys_DUCET.txt, the start of each reordering group is normally the primary weight corresponding to the same character sequence as for allkeys_CLDR.txt. In a few cases this requires adjustment, especially for the special reordering groups, due to CLDR's ordering the common characters more strictly by category than the DUCET (as described in *Section 2, Root Collation*). The necessary adjustment would set the start of each allkeys_DUCET.txt reordering group to the primary weight of the first mapping for the relevant General_Category for a special reordering group (for characters that sort before 'a'), or the primary weight of the first mapping for the first script (e.g., sc=Grek) of an "alphabetic" group (for characters that sort at or after 'a').

Note that the following only applies to primary weights greater than the one for U+FFFE and less than "trailing" weights.

The special reordering groups correspond to General_Category values as follows:

- punct: P
- symbol: Sk, Sm, So
- space: Z, Cc
- currency: Sc
- digit: Nd

In the DUCET, some characters that sort below 'a' and have other General_Category values not mentioned above (e.g., gc=Lm) are also grouped with symbols. Variants of numbers (gc=No or Nl) can be found among punctuation, symbols, and digits.

Each collation element of an expansion may be in a different reordering group, for example for parenthesized characters.

3.14 Case Parameters

The **case level** is an *optional* intermediate level ("2.5") between Level 2 and Level 3 (or after Level 1, if there is no Level 2 due to strength settings). The case level is used to support two parametric features: ignoring non-case variants (Level 3 differences) except for case, and giving case differences a higher-level priority than other tertiary differences. Distinctions between small and large Kana characters are also included as case differences, to support Japanese collation.

The **case first** parameter controls whether to swap the order of upper and lowercase. It can be used with or without the case level.

Importantly, the case parameters have no effect in many instances. For example, they have no effect on the comparison of two non-ignorable characters with different primary weights, or with different secondary weights if the strength = **secondary (or higher).**

When either the **case level** or **case first** parameters are set, the following describes the derivation of the modified collation elements. It assumes the original levels for the code point are [p.s.t] (primary, secondary, tertiary). This derivation may change in future versions of LDML, to track the case characteristics more closely.

3.14.1 Untailored Characters

For untailored characters and strings, that is, for mappings in the root collation, the case value for each collation element is computed from the tertiary weight listed in allkeys_CLDR.txt. This is used to modify the collation element.

Look up a case value for the tertiary weight x of each collation element:

1. UPPER if x ∈ {08-0C, 0E, 11, 12, 1D}
2. UNCASED otherwise
3. FractionalUCA.txt encodes the case information in bits 6 and 7 of the first byte in each tertiary weight. The case bits are set to 00 for UNCASED and LOWERCASE, and 10 for UPPER. There is no MIXED case value (01) in the root collation.

3.14.2 Compute Modified Collation Elements

From a computed case value, set a weight **c** according to the following.

1. If **CaseFirst=UpperFirst**, set **c** = UPPER ? **1** : MIXED ? 2 : **3**
2. Otherwise set **c** = UPPER ? **3** : MIXED ? 2 : **1**

Compute a new collation element according to the following table. The notation *xt* means that the values are numerically combined into a single level, such that xt < yu whenever x < y. The fourth level (if it exists) is unaffected. Note that a secondary CE must have a secondary weight S which is greater than the secondary weight s of any primary CE; and a tertiary CE must have a tertiary weight T which is greater than the tertiary weight t of any primary or secondary CE ([UCA] WF2).

Case Level	Strength	Original CE	Modified CE	Comment
on	**primary**	`0.S.t`	`0.0`	ignore case level weights of primary-ignorable CEs
		`p.s.t`	`p.c`	
	secondary or higher	`0.0.T`	`0.0.0.T`	ignore case level weights of secondary-ignorable CEs
		`0.S.t`	`0.S.c.t`	
		`p.s.t`	`p.s.c.t`	
off	any	`0.0.0`	`0.0.00`	ignore case level weights of tertiary-ignorable CEs
		`0.0.T`	`0.0.3T`	
		`0.S.t`	`0.S.ct`	
		`p.s.t`	`p.s.ct`	

For primary+case, which is used for “ignore accents but not case” collation, primary ignorables are ignored so that a = ä. For secondary+case, which would by analogy mean “ignore variants but not case”, secondary ignorables are ignored for equivalent behavior.

When using **caseFirst** but not **caseLevel**, the combined case+tertiary weight of a tertiary CE must be greater than the combined case+tertiary weight of any primary or secondary CE so that [UCA] well-formedness condition 2 is fulfilled. Since the tertiary CE’s tertiary weight T is already greater than any t of primary or secondary CEs, it is sufficient to set its case weight to UPPER=3. It must not be affected by **caseFirst=upper**. (The table uses the constant 3 in this case rather than the computed c.)

The case weight of a tertiary-ignorable CE must be 0 so that [UCA] well-formedness condition 1 is fulfilled.

3.14.3 Tailored Strings

Characters and strings that are tailored have case values computed from their root collation case bits.

1. Look up the tailored string’s root CEs. (Ignore any prefix or extension strings.) N=number of primary root CEs.
2. Determine the number and type (primary vs. weaker) of CEs a tailored string maps to. M=number of primary tailored CEs.
3. If N<=M (no more root than tailoring primary CEs): Copy the root case bits for primary CEs 0..N-1.
    - If N<M (fewer root primary CEs): Clear the case bits of the remaining tailored primary CEs. (uncased/lowercase/small Kana)
4. If N>M (more root primary CEs): Copy the root case bits for primary CEs 0..M-2. Set the case bits for tailored primary CE M-1 according to the remaining root primary CEs M-1..N-1:
    - Set to uncased/lower if all remaining root primary CEs have uncased/lower.
    - Set to uppercase if all remaining root primary CEs have uppercase.
    - Otherwise, set to mixed.
5. Clear the case bits for secondary CEs 0.s.t.
6. Tertiary CEs 0.0.t must get uppercase bits.
7. Tertiary-ignorable CEs 0.0.0 must get ignorable-case=lowercase bits.

Note: Almost all Cased characters have primary (non-ignorable) root collation CEs, except for U+0345 Combining Ypogegrammeni which is Lowercase. All Uppercase characters have primary root collation CEs.

3.15 Visibility

Collations have external visibility by default, meaning that they can be displayed in a list of collation options for users to choose from. A collation whose type name starts with "private-" is internal and should not be shown in such a list. Collations are typically internal when they are partial sequences included in other collations. See *Section 3.1, Collation Types* .

3.16 Collation Indexes

3.16.1 Index Characters

The main data includes <exemplarCharacters> for collation indexes. See *Part 2 General, Section 3, Character Elements*, for general information about exemplar characters.

The index characters are a set of characters for use as a UI "index", that is, a list of clickable characters (or character sequences) that allow the user to see a segment of a larger "target" list. Each character corresponds to a bucket in the target list. One may have different kinds of index lists; one that produces an index list that is relatively static, and the other is a list that produces roughly equally-sized buckets. While CLDR is mostly focused on the first, there is provision for supporting the second as well.

The index characters need to be used in conjunction with a collation for the locale, which will determine the order of the characters. It will also determine which index characters show up.

The static list would be presented as something like the following (either vertically or horizontally):

… A B C D E F G H CH I J K L M N O P Q R S T U V W X Y Z …

In the "A" bucket, you would find all items that are primary greater than or equal to "A" in collation order, and primary less than "B". The use of the list requires that the target list be sorted according to the locale that is used to create that list. Although we say "character" above, the index character could be a sequence, like "CH" above. The index exemplar characters must always be used with a collation appropriate for the locale. Any characters that do not have primary differences from others in the set should be removed.

Details:

1. The primary weight (according to the collation) is used to determine which bucket a string is in. There are special buckets for before the first character, between buckets of different scripts, and after the last bucket (and of a different script).
2. Characters in the *index characters* do not need to have distinct primary weights. That is, the *index characters* are adapted to the underlying collation: normally Ë is in the E bucket for Russian, but if someone used a variant of Russian collation that distinguished them on a primary level, then Ë would show up as its own bucket.

3. If an *index character* string ends with a single "*" (U+002A), for example "Sch*" and "St*" in German, then there will be a separate bucket for the string minus the "*", for example "Sch" and "St", even if that string does not sort distinctly.
4. An *index character* can have multiple primary weights, for example "Æ" and "Sch". Names that have the same initial primary weights sort into this *index character*'s bucket. This can be achieved by using an upper-boundary string that is the concatenation of the *index character* and U+FFFF, for example "Æ\uFFFF" and "Sch\uFFFF". Names that sort greater than this upper boundary but less than the next index character are redirected to the last preceding single-primary index character (A and S for the examples here).

For example, for index characters `[A Æ B R S {Sch*} {St*} T]` the following sample names are sorted into an index as shown.

- A — Adelbert, Afrika
- Æ — Æsculap, Aesthet
- B — Berlin
- R — Rilke
- S — Sacher, Seiler, Sultan
- Sch — Schiller
- St — Steiff
- T — Thomas

The … items are special: each is a bucket for everything else, either less or greater. They are inserted at the start and end of the index list, *and* on script boundaries. Each script has its own range, except where scripts sort primary-equal (e.g., Hira & Kana). All characters that sort in one of the low reordering groups (whitespace, punctuation, symbols, currency symbols, digits) are treated as a single script for this purpose.

If you tailor a Greek character into the Cyrillic script, that Greek character will be bucketed (and sorted) among the Cyrillic ones.

Even in an implementation that reorders groups of scripts rather than single scripts, for example Hebrew together with Phoenician and Samaritan, the index boundaries are really script boundaries, *not* multi-script-group boundaries. So if you had a collation that reordered Hebrew after Ethiopic, you would still get index boundaries between the following (and in that order):

1. Ethiopic
2. Hebrew
3. Phoenician *// included in the Hebrew reordering group*
4. Samaritan *// included in the Hebrew reordering group*
5. Devanagari

(Beginning with CLDR 27, single scripts can be reordered.)

In the UI, an index character could also be omitted or grayed out if its bucket is empty. For example, if there is nothing in the bucket for Q, then Q could be omitted. That would be up to the implementation. Additional buckets could be added if other characters are present. For example, we might see something like the following:

Sample Greek Index	Contents
Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ Σ Τ Υ Φ Χ Ψ Ω	With only content beginning with Greek letters
… Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ Σ Τ Υ Φ Χ Ψ Ω …	With some content before or after
… 9 Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ Σ Τ Υ Φ Χ Ψ Ω …	With numbers, and nothing between 9 and Alpha
… 9 *A-Z* Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ Σ Τ Υ Φ Χ Ψ Ω …	With numbers, some Latin

Here is a sample of the XML structure:

```
<exemplarCharacters type="index">[A B C D E F G H I J K L M N O P Q R S T U V W X Y Z]</exemplarCharacters>
```

The display of the index characters can be modified with the Index labels elements, discussed in the *Part 2 General, Section 3.3,* Index Labels .

3.16.2 CJK Index Markers

Special index markers have been added to the CJK collations for stroke, pinyin, zhuyin, and unihan. These markers allow for effective and robust use of indexes for these collations.

The per-language index exemplar characters are not useful for collation indexes for CJK because for each such language there are multiple sort orders in use (for example, Chinese pinyin vs. stroke vs. unihan vs. zhuyin), and these sort orders use very different index characters. In addition, sometimes the boundary strings are different from the bucket label strings. For collations that contain index markers, the boundary strings and bucket labels should be derived from those index markers, ignoring the index exemplar characters.

For example, near the start of the pinyin tailoring there is the following:

```
<p> A</p><!-- INDEX A -->
<pc>阿呵□锕□□</pc><!-- ā -->
…
<pc>翱</pc><!-- ao -->
<p> B</p><!-- INDEX B -->
```

These indicate the boundaries of "buckets" that can be used for indexing. They are always two characters starting with the noncharacter U+FDD0, and thus will not occur in normal text. For pinyin the second character is A-Z; for unihan it is one of the radicals; and for stroke it is a character after U+2800 indicating the number of strokes, such as ⠁ . For zhuyin the second character is one of the standard Bopomofo characters in the range U+3105 through U+3129.

The corresponding bucket label strings are the boundary strings with the leading U+FDD0 removed. For example, the Pinyin boundary string "\uFDD0A" yields the label string "A".

However, for stroke order, the label string is the stroke count (second character minus U+2800) as a decimal-digit number followed by 劃

(U+5283). For example, the stroke order boundary string "\uFDD0\u2805" yields the label string "5劃".