Unicode LDML Keyboard Enhancements

M. Hosken, A. Glass, M. Davis, M. Durdin,

and the Unicode CLDR subcommittee

(Working Draft)

Introduction

The Unicode CLDR Committee is planning to enhance the Unicode LDML keyboard specifications. The goal is to be able to represent all the keyboard features necessary to support keyboard layouts from all major providers, allowing the CLDR repository of keyboard layouts to support not only languages in widespread use, but also digitally disadvantaged languages. As a part of this work, keyboards add support more complex scripts, add capabilities for virtual keyboards (especially mobile phones), incorporate features needed on specific platforms, and provide better layouts overall. Keyboards would also be able to import files, reducing maintenance by allowing common features to be shared. For complex scripts, the transform elements are made more powerful, and reorder and backspace transforms are added.

The plan is to incorporate these changes into CLDR v33 (ca. March 2018), and to work thereafter to improve the tooling for the new specification, and streamline the process for submitting new keyboards into CLDR.

The CLDR committee is soliciting feedback on the proposal so that it can make any necessary improvements. The closing date for providing feedback is Feb 1, 2018.

The proposed changes are deltas on top of the current format. As such this document should be read in conjunction with UTS#35. See tr35-keyboards.html for context, and the v32 keyboard DTD. Modifications to that DTD are indicated by blue underline in DTD listings below.

LDML keyboard descriptions are split across two primary structures: the **platform** that describes hardware layouts and the **keyboard** description that aims to be hardware independent and express the mapping of keys to characters and sequences. All of the changes are to the **keyboard** element (ldmlKeyboard.dtd).

Table of Contents

Current DTD

For comparison, here is the current DTD for keyboard. We are not changing any of this except additions, so this is just a reminder.

```
<!ELEMENT keyboard ( version, generation?, names, settings?, keyMap+, transforms? ) >
<!ATTLIST keyboard locale CDATA #REQUIRED >

<!ELEMENT version EMPTY >
<!ATTLIST version platform CDATA #REQUIRED >   <!--@VALUE-->
<!ATTLIST version number CDATA #REQUIRED >   <!--@METADATA-->
<!ATTLIST version cldrVersion CDATA #FIXED "32" >   <!--@METADATA-->

<!ELEMENT generation EMPTY >   <!--@DEPRECATED-->
<!ATTLIST generation date CDATA #REQUIRED >   <!--@VALUE-->   <!--@DEPRECATED-->

<!ELEMENT names ( name+ ) >

<!ELEMENT name EMPTY >
<!ATTLIST name value CDATA #REQUIRED >   <!--@VALUE-->

<!ELEMENT settings EMPTY >   <!--@ORDERED-->
<!ATTLIST settings fallback (omit) #IMPLIED >   <!--@VALUE-->
<!ATTLIST settings transformFailure (omit) #IMPLIED >   <!--@VALUE-->
<!ATTLIST settings transformPartial (hide) #IMPLIED >   <!--@VALUE-->

<!ELEMENT keyMap ( map+ ) >
<!ATTLIST keyMap modifiers CDATA #IMPLIED >

<!ELEMENT map EMPTY >
<!ATTLIST map iso CDATA #REQUIRED >
<!ATTLIST map to CDATA #REQUIRED >   <!--@VALUE-->
```

<!ATTLIST map longPress CDATA #IMPLIED >   <!--@VALUE-->
<!ATTLIST map transform (no) #IMPLIED >   <!--@VALUE-->

<!ELEMENT transforms (transform+) >
<!ATTLIST transforms type CDATA #REQUIRED >

<!ELEMENT transform EMPTY >
<!ATTLIST transform from CDATA #REQUIRED >
<!ATTLIST transform to CDATA #REQUIRED >   <!--@VALUE-->

General

<!ELEMENT keyboard (version, generation?, names?, settings?, import*, keyMap*, displayMap?, layer*, vkeys*, transforms*, reorders?, backspaces?) >

> **Note that vkeys occurs as a child of two different elements: in keyboard it is applicable to the physical platform, and in layer it is applicable to a virtual keyboard.**

---

import

<!ELEMENT import EMPTY>
<!ATTLIST import filePath CDATA #REQUIRED>

One primary design change in the LDML description is to encourage layering. This means that files can reference, and so include, the information from other files. To achieve this we need a new second level element: **import**.

The **import** element references another file of the same type and includes all the subelements of the top level element as though the include element were being replaced by those elements. For example:

> **<import filePath="xxx.xml">**

An imported file may itself contain import elements. Any cycles among the imports, however, make any affected keyboard definition invalid. For example, the following are all invalid because of a cycle.

> **xxx.xml contains <import filePath="yyy.xml">**
>
> **yyy.xml contains <import filePath="zzz.xml">**
>
> **zzz.xml contains <import filePath="xxx.xml">**

The attributes supported by this element are:

> **filePath** contains a relative file path to the ldml file to be included.

The contents of all the imported files are accumulated in order, with later elements winning over earlier ones. The XML paths from the import elements and from the body of the XML file are resolved in in order when interpreting the file. For each distinguishing path, the latest instance will take precedence. Thus consider the following two paths, defined in this order (for example once in an included file and once in the body of the XML file). They both have the same distinguishing path, so the second one overrides the first one.

path1	**//ldml/keyboard/keyMap/map[@iso="E01"][to="1"]**
distinguishing path1	**//ldml/keyboard/keyMap/map[@iso="E01"]**
value attribute1	**[@to="1"]**

path2	**//ldml/keyboard/keyMap/map[@iso="E01"][@to="฿"]**
distinguishing path2	**//ldml/keyboard/keyMap/map[@iso="E01"]**
value attribute2	**[@to="฿"]**

In order to help identify mistakes, it is an error if a single file contains two elements that override each other. All element overrides must come as a result of an <include> element either for the element overridden or the element overriding.

The following elements are *not* imported from the source file:

- version
- generation
- names
- settings

Importing Ordered Elements

When inheriting ordered elements, the child elements always come before the parent elements. That way when the elements are "executed" (searched linearly for first match), the child elements override the parent elements. Implementations are free, in such cases, to discard elements that cannot be executed. For example:

Child:	e1=v1, e9=v2, e4=v3
Parent:	e1=v4, e7=v5
Resolved:	e1=v1, e9=v2, e4=v3, ~~e1=v4,~~ e7=v5

KeyMap

There are various elements in the **keyMap** element that are expanded, as well as new ones added.

<!ELEMENT keyMap (map | flicks)+ >

map Attribute

<!ATTLIST map multitap (no) #IMPLIED >

<!--@VALUE-->

There is a new attribute **multitap** to the existing **map** element. This is a space-delimited list of strings, where each successive element of the list is produced by the corresponding number of quick taps. For example, two taps on the key C01 will produce a “c” in the following example.

Example

<map iso="C01" to="a" **multitap="b c d"**>

Flicks

<!ELEMENT flicks (flick+)>
<!ATTLIST flicks iso NMTOKENS #REQUIRED>

<!ELEMENT flick EMPTY>
<!ATTLIST flick directions NMTOKENS>
<!ATTLIST flick to CDATA>
<!--@VALUE-->

The directions value is a space-delimited list of keywords, currently restricted to directions {n e s w ne nw se sw}. The to value is the result of (one or more) flicks.

Example

```
<flicks iso="C01">
      <flick directions=“ne s” to=“\uABCD\uDCBA”>
</flicks>
```

displayMap

<!ELEMENT displayMap (display+)>

<!ELEMENT display EMPTY>
<!ATTRIBUTE display mapOutput CDATA #REQUIRED>
<!ATTRIBUTE display keycap CDATA #REQUIRED>
<!-- @VALUE -->

There is a new subelement to keyboard that describes what is to be displayed on the keytops for various keys. For the most part, such explicit information is unnecessary since the @char element from the keyMap/map element can be used. But there are some characters, such as diacritics, that do not display well on their own and so explicit overrides for such characters can help. The displayMap consists of a list of **display** subelements.

displayMaps are designed to be shared across many different keyboard layout descriptions, and included where needed.

display

The display element describes how a character that has come from a keyMap/map element, should be displayed on a keyboard layout where such display is possible. It has the following attributes:

mapOutput specifies the character or character sequence from the keyMap/map element that is to have a special display.

display specifies the character sequence that should be displayed on the keytop for any key that generates the @mapOutput sequence. (It is an NOP if the value of the display attribute is the same as the value of the **to** attribute.)

Example:

```
<keyboard>
        <keyboardMap>
                <map iso="C01" to="a" longPress="\u0301 \u0300"/>
        </keyboardMap>

        <displayMap>
                <display mapOutput="\u0300" keycap="\u02CB"/>
                <display mapOutput="\u0301" keycap="\u02CA"/>
        </displayMap>

</keyboard>
```

To allow **displayMaps** to be shared across descriptions, there is no requirement that @mapOutput matches any @to in any keyMap/map element in the keyboard description.

layer

```
<!ELEMENT layer ( row+, switch*, vkeys* )>
<!ATTRIBUTE layer modifier CDATA #REQUIRED>

<!ELEMENT row EMPTY>
<!ATTRIBUTE row keys CDATA #REQUIRED>

<!ELEMENT switch EMPTY>
<!ATTRIBUTE switch iso CDATA #REQUIRED>
<!ATTRIBUTE switch layer CDATA #REQUIRED>
        <!-- @VALUE -->
<!ATTRIBUTE switch keycap CDATA #IMPLIED>
        <!-- @VALUE -->
```

Note that vkeys occurs as a child of two different elements: in keyboard it is applicable to the physical platform, and in layer it is applicable to a virtual keyboard.

A layer element describes the configuration of keys on a particular layer of a keyboard. It contains **row** elements to describe which keys exist in each row and **switch** elements that describe how keys in the layer switch the layer to another. In addition, for platforms that require a mapping from a key to a virtual key (for example Windows or Mac) there is also a **vkeys** element to describe the mapping.

The layer element takes a single required attribute:

modifier This has two roles. It acts as an identifier for the layer element and also provides the linkage into a keyMap. A modifier is a single modifier combination such that it is matched by one of the modifier combinations in one of the keyMap/@modifiers attribute. To indicate that no modifiers apply the reserved name of "none" is used. For the purposes of fallback vkey mapping, the following modifier components are reserved:

"shift", "ctrl", "alt", "caps", "cmd", "opt" along with the "L" and "R" optional single suffixes for the first 3 in that list. It is an error for there not to be a keyMap/@modifiers attribute that does not match the modifier attribute value.

Matching does not require individual modifiers in a combination to be in the same order as the matching modifier combination. In addition, optional modifiers in a matching combination do not need to be listed in the modifier combination being matched. For example, on the left are some keyMap/@modifiers values and on the right a possible minimal layer/@modifier that would be matched.

keymap/@modifiers	**layer/@modifier**
shift caps+shiftR+shiftL?	**shift**
caps+cmd?	**caps**
caps+shiftL	**caps+shiftL**
opt+caps? cmd+optR+caps+optL?	**opt**
opt+shift+caps? opt+caps+shift+cmd? cmd+optR+shift+optL? cmd+opt+shiftR+shiftL?	**opt+shiftL (since shift matches shiftL)**
ctrl+opt?+caps?+shift? ctrl+cmd?+opt?+shift? ctrl+cmd?+opt?+caps? cmd+ctrlR+opt+caps+shift cmd+ctrlL+opt+caps+shift cmd+ctrl+caps+shift+optR? cmd+ctrl+caps+shift+optL? cmd+ctrl+opt+caps+shiftR? cmd+ctrl+opt+caps+shiftL?	**ctrl**
cmd+optL+shiftL	**optL+cmd+shiftL**
cmd+optL+caps	**caps+cmd+optL**
cmd+opt	**cmd+opt**

The use of @modifier as an identifier for a layer is sufficient since it is always unique among the set of layer elements in a keyboard.

row

A row element describes the keys that are present in the row of a keyboard. Row elements are ordered within a layout element with the top visual row being stored first. The row element introduces the keyId, which may be an ISOKey or a specialKey. More formally:

keyId = ISOKey | specialKey

ISOKey = [A-Z][0-9][0-9]

specialKey = [a-z][a-zA-Z0-9]{2,8}

A row element contains one required attribute:

keys This is a string that lists the keyId for each of the keys in a row. Key ranges may be contracted to firstkey-lastkey but only for ISOKey type keyIds. The interpolation between the first and last keys names is entirely numeric. Thus D00-D03 is equivalent to

D00 D01 D02 D03. If the first and last keys do not have the same alphabetic prefix or the last key numeric component is less than or equal to the first key numeric component then it is an error.

specialKey type keyIds may take any value within their syntactic constraint. But the following specialKeys are reserved to allow applications to identify them and give them special handling:

- "bksp", "enter", "space", "tab", "esc", "sym", "num"
- all the reserved modifier names
- specialKeys starting with the letter "x" for future reserved names.

switch

The switch element describes a function key that has been included in the layout. It specifies which layer pressing the key switches you to, and also what the key looks like. It has the following required attributes:

iso The keyId as specified in one of the row elements. This must be a specialKey and not an ISOKey.

layout The modifier attribute of the resulting layout element that describes the new layer the user gets switched to.

keycap A string to be displayed on the key.

Here is a short example of a layouts element and its contents:

```
<layer modifier="none">
        <row keys="D01-D10"/>
        <row keys="C01-C09"/>
        <row keys="shift B01-B07 bksp"/>
        <row keys="sym A01 smilies A02-A03 enter"/>
        <switch iso="sym" layout="sym" keycap="?123"/>
        <switch iso="shift" layout="shift" keycap="&#x21EA;"/>
</layer>

<layer modifier="shift">
        <row keys="D01-D10"/>
        <row keys="C01-C09"/>
        <row keys="shift B01-B07 bksp"/>
        <row keys="sym A01 smilies A02-A03 enter"/>
        <switch iso="sym" layout="sym" keycap="?123"/>
        <switch iso="shift" layout="none" keycap="&#x21EA;"/>
</layer>
```

vkeys

```
<!ELEMENT vkeys ( vkey* )>
<!ATTRIBUTE vkeys type CDATA #REQUIRED>

<!ELEMENT vkey>
<!ATTRIBUTE vkey iso CDATA #REQUIRED>
<!ATTRIBUTE vkey vkey CDATA #REQUIRED>
        <!-- @VALUE -->
<!ATTRIBUTE vkey vkeyModifier CDATA #IMPLIED>
```

<!-- @VALUE -->

On some architectures, applications may directly interact with keys before they are converted to characters. The keys are identified using a virtual key identifier or vkey. The mapping between a physical keyboard key and a vkey is keyboard layout dependent. For example, a French keyboard would identify the D01 key as being an 'a' with a vkey of 'a' as opposed to 'q' on a US English keyboard. While vkeys are layout dependent, they are not modifier dependent. A shifted key always has the same vkey as its unshifted counterpart. In effect, a key is identified by its vkey and the modifiers active at the time the key was pressed.

For a physical keyboard there is a layout specific default mapping of keys to vkeys. These are listed in a vkeys element which takes a list of vkey element mappings and is identified by a type. There are different vkey mappings required for different platforms. While Windows vkeys are very similar to Mac vkeys, they are not identical and require their own mapping.

The most common model for specifying vkeys is to import a standard mapping, say to the US layout, and then to add a vkeys element to change the mapping appropriately for the specific layout.

In addition to describing physical keyboards, vkeys also get used in virtual keyboards. Here the vkey mapping is local to a layer and therefore a vkeys element may occur within a layout element. In the case where a layout element has no vkeys element then the resulting mapping may either be empty (none of the keys represent keys that have vkey identifiers) or may fallback to the layout wide vkeys mapping. Fallback only occurs if the layout's modifier attribute consists only of standard modifiers as listed as being reserved in the description of the layout/@modifier attribute, and if the modifiers are standard for the platform involved. So for Windows, 'cmd' is a reserved modifier but it is not standard for Windows. Therefore on Windows the vkey mapping for a layout with @modifier="cmd" would be empty.

A vkeys element consists of a list of vkey elements

vkey

A vkey element describes a mapping between a key and a vkey for a particular platform. It takes various attributes:

iso [required] The ISOkey being mapped.

vkey [required] The resultant vkey identifier.

vkeyModifier This attribute may only be used if the parent vkeys element is a child of a layout element. If present it allows an unmodified key from a layer to represent a modified virtual key.

This example shows some of the mappings for a French keyboard layout:

shared/win-vkey.xml

```
<keyboard>
        <vkeys type="windows">
                <vkey iso="D01" vkey="VK_Q"/>
                <vkey iso="D02" vkey="VK_W"/>
                <vkey iso="C01" vkey="VK_A"/>
                <vkey iso="B01" vkey="VK_Z"/>
        </vkeys>
</keyboard>
```

win-fr.xml

```
<keyboard>
```

```
		<import path="shared/win-vkey.xml"/>

		<keyMap>
				<map iso="D01" to="a"/>
				<map iso="D02" to="z"/>
				<map iso="C01" to="q"/>
				<map iso="B01" to="w"/>
		</keyMap>

		<keyMap modifiers="shift">
				<map iso="D01" to="A"/>
				<map iso="D02" to="Z"/>
				<map iso="C01" to="Q"/>
				<map iso="B01" to="W"/>
		</keyMap>

		<vkeys type="windows"/>
				<vkey iso="D01" vkey="VK_A"/>
				<vkey iso="D02" vkey="VK_Z"/>
				<vkey iso="C01" vkey="VK_Q"/>
				<vkey iso="B01" vkey="VK_W"/>
		</vkeys>

</keyboard>
```

In the context of a virtual keyboard there might be a symbol layer with the following layout:

```
<keyboard>
		<keyMap modifiers="sym">
				<map iso="D01" to="1"/>
				<map iso="D02" to="2"/>
						...
				<map iso="D09" to="9"/>
				<map iso="D10" to="0"/>
				<map iso="C01" to="!"/>
				<map iso="C02" to="@"/>
						...
				<map iso="C09" to="("/>
				<map iso="C10" to=")"/>
		</keyMap>
		<layer modifier="sym">
				<row keys="D01-D10"/>
				<row keys="C01-C10"/>
				<row keys="alpha B01-B07 bksp"/>
				<row keys="shift A00-A03 enter"/>
				<switch iso="alpha" layer="none" keycap="ABC"/>
				<switch iso="shift" layer="sym+shift" keycap="=/<"/>
				<vkeys type="windows">
						<vkey iso="D01" vkey="VK_1"/>
								...
						<vkey iso="D10" vkey="VK_0"/>
						<vkey iso="C01" vkey="VK_1" vkeyModifier="shift"/>
								...
						<vkey iso="C10" vkey="VK_0" vkeyModifier="shift"/>
				</vkeys>
		</layer>
</keyboard>
```

Transforms

<!ELEMENT transforms (transform+) >

<!ELEMENT transform EMPTY >
<!ATTLIST transform before CDATA #IMPLIED>
<!ATTLIST transform from CDATA #REQUIRED >
<!ATTLIST transform after CDATA #IMPLIED>
<!ATTLIST transform to CDATA #IMPLIED >
<!--@VALUE-->
<!ATTLIST transform error (true) #IMPLIED>
<!--@VALUE-->

The specification for post keyMap transforms was always designed to grow. The current single *simple* transform provides a powerful mechanism that handles such basic keying behaviours as sequence replacement and dead keys. This transform remains unchanged, in essence, although it does gain some extra expressive power.

transform

A transform element gains some new general attributes and one attribute is extended:

from The from attribute consists of a sequence of elements. Each element matches one character and may consist of a codepoint or a UnicodeSet (both as defined in UTS#35 section 5.3.3).

before This attribute consists of a sequence of elements (codepoint or UnicodeSet) and matches as a zero-width assertion preceding the current position being matched in the text. The attribute must match for the transform to apply. If missing, no before constraint is applied. The attribute value must not be empty.

after This attribute consists of a sequence of elements (codepoint or UnicodeSet) and matches as a zero-width assertion after the @from sequence. The attribute must match for the transform to apply. If missing, no after constraint is applied. The attribute value must not be empty. When the transform is applied, the string matched by the @from attribute is replaced by the string in the @to attribute, with the text matched by the @after attribute left unchanged. After the change, the current position is reset to just after the text output from the @to attribute and just before the text matched by the @after attribute. Warning: some legacy implementations may not be able to make such an adjustment and will place the current position after the @after matched string.

error If set this attribute indicates that the keyboarding application may indicate an error to the user in some way. Processing would stop and rewind any state to before the key was pressed. There is no @to attribute in transform elements for which @error is set. The @error attribute takes the value "fail", or must be absent. Since processing is considered to stop after an error, no further transforms on the same input are applied.

For example:

```
<transform from="\u037A\u037A" error="fail"/>
```

This indicates that it is an error to type two iota subscripts immediately after each other.

In terms of how these different attributes work in processing a sequences of transforms, consider the transform:

```
        <transform before="X" from="Y" after="Z" to="B"/>
```

This would transform the string:

```
        XYZ → XBZ
```

If we mark where the current match position is before and after the transform we see:

```
        X | Y Z → X B | Z
```

And a subsequent transform could transform the Z string, looking back (using @before) to match the B.

There are other keying behaviours that are needed particularly in handling languages and scripts from various parts of the world. The behaviours intended to be covered by the proposed transforms are:

- Reordering combining marks. The order required for underlying storage may differ considerably from the desired typing order. In addition, a keyboard may want to allow for different typing orders.
- Error indication. Sometimes a keyboard layout will want to specify to the application that a particular keying sequence in a context is in error and that the application should indicate that that particular keypress is erroneous.
- Backspace handling. There are various approaches to handling the backspace key. An application may treat it as an undo of the last key input, or it may simply delete the last character in the currently output text, or it may use transform rules to tell it how much to delete.

We consider each transform type in turn and consider attributes to the <transforms> element pertinent to that type.

reorder

<!ELEMENT reorders (reorder+) >

<!ELEMENT reorder EMPTY>
<!ATTLIST reorder before CDATA #IMPLIED>
<!ATTLIST reorder from CDATA #REQUIRED>
<!ATTLIST reorder after CDATA #IMPLIED>
<!ATTLIST reorder order NMTOKENS #IMPLIED>
        <!--@VALUE-->
<!ATTLIST reorder tertiary NMTOKENS #IMPLIED>
        <!--@VALUE-->
<!ATTLIST reorder tertiary_base NMTOKENS #IMPLIED>
        <!--@VALUE-->
<!ATTLIST reorder prebase NMTOKENS #IMPLIED>
        <!--@VALUE-->

The reorder transforms are applied after all transforms except for those with type="final". The transforms of type="final" are run after the reorder transforms.

This transform has the job of reordering sequences of characters that have been typed, from their typed order to the desired output order. The primary concern in this transform is to sort combining marks into their correct relative order after a base, as described in this section. The reorder transform executes after the simple transform (the transform element that is in CLDR v32), taking its output as its input. While keyboard layout designers are very unlikely to get involved in the core aspects of the algorithm and its configuration, the algorithm *is* data driven and can be configured within any keyboard layout. Keyboard layouts will almost always import a standard definition of this transforms.

The reordering algorithm consists of three parts:

- Create a sort key for each character in the input string. A sort key has 3 parts: (primary, index, tertiary).
    - The **primary weight** is the primary order value.
    - The **secondary weight** is the index, a position in the input string, usually of the character itself, but it may be of a character earlier in the string.
    - The **tertiary weight** is a tertiary order value (defaulting to 0).
    - The **quaternary weight** is the index of the character in the string. This ensures a stable sort for sequences of characters with the same tertiary weight.
- Mark each character as to whether it is a prebase character, one that is typed before the base and logically stored after. Thus it will have a primary order > 0.
- Use the sort key and the prebase mark to identify runs. A run starts with a prefix that contains any prebase characters and a single base character whose primary and tertiary key is 0. The run extends until, but not including, the start of the prefix of the next run.
    - run := prebase* (primary=0 && tertiary=0) ((primary≠0 || tertiary≠0) && !prebase)*
- Sort the character order of each character in the run based on its sort key.

The primary order of a character with a CCC of 0 may well not be 0. In addition, a character may receive a different primary order dependent on context. For example, in the Devanagari sequence ka halant ka, the first ka would have a primary order 0 while the halant ka sequence would give both halant and the second ka a primary order > 0, for example 2. Note that “base” character in this discussion is not a Unicode base character. It is instead a character with primary=0.

In order to get the characters into the correct relative order, it is necessary not only to order combining marks relative to the base character, but also to order some combining marks in a subsequence following another combining mark. For example in Devanagari, a nukta may follow consonant character, but it may also follow a conjunct consisting of a consonant, halant, consonant. Notice that the second consonant is not, in this model, the start of a new run because some characters may need to be reordered to before the first base, for example repha. The repha would get primary < 0, and be sorted before the character with order = 0, which is, in the case of Devanagari, the initial consonant of the orthographic syllable.

The reorder transform consists of a single element type: <reorder> encapsulated in a <reorders> element. Each is a rule that matches against a string of characters with the action of setting the various ordering attributes (primary, tertiary, tertiary_base, prebase) for the matched characters in the string.

from This attribute follows the transform/@from attribute and contains a string of elements. Each element matches one character and may consist of a codepoint or a UnicodeSet (both as defined in UTS#35 section 5.3.3). This attribute is required.

before This attribute follows the transform/@before attribute and contains the element string that must match the string immediately preceding the start of the string that the @from matches.

after This attribute follows the transform/@after attribute and contains the element string that must match the string immediately following the end of the string that the @from matches.

order This attribute gives the primary order for the elements in the matched string in the @from attribute. The value is a simple integer between -128 and +127 inclusive, or a space separated list of such integers. For a single integer, it is applied to all the elements in the matched string. Details of such list type attributes are given after all the attributes are described. If missing, the order value of all the matched characters is 0. We consider the order value for a matched character in the string.

- If the value is 0 and its tertiary value is 0, then the character is the base of a new run.
- If the value is 0 and its tertiary value is non-zero, then it is a normal character in a run, with ordering semantics as described in the @tertiary attribute.
- If the value is negative, then the character is a primary character and will reorder to be before the base of the run.
- If the value is positive, then the character is a primary character and is sorted based on the order value as the primary key following a previous base character.

A character with a zero tertiary value is a primary character and receives a sort key consisting of:

- Primary weight is the order value
- Secondary weight is the index of the character. This may be any value (character index, codepoint index) such that its value is greater than the character before it and less than the character after it.
- Tertiary weight is 0.
- Quaternary weight is the same as the secondary weight.

tertiary This attribute gives the tertiary order value to the characters matched. The value is a simple integer between -128 and +127 inclusive, or a space separated list of such integers. If missing, the value for all the characters matched is 0. We consider the tertiary value for a matched character in the string.

- If the value is 0 then the character is considered to have a primary order as specified in its order value and is a primary character.
- If the value is non zero, then the order value must be zero otherwise it is an error. The character is considered as a tertiary character for the purposes of ordering.

A tertiary character receives its primary order and index from a previous character, which it is intended to sort closely after. The sort key for a tertiary character consists of:

- Primary weight is the primary weight of the primary character
- Secondary weight is the index of the primary character, not the tertiary character
- Tertiary weight is the tertiary value for the character.
- Quaternary weight is the index of the tertiary character.

tertiary_base This attribute is a space separated list of "true" or "false" values corresponding to each character matched. It is illegal for a tertiary character to have a true tertiary_base value. For a primary character it marks that this character may have tertiary characters moved after it. When calculating the secondary weight for a tertiary character, the most recently encountered primary character with a true tertiary_base attribute is used. Primary characters with an @order value of 0 automatically are treated as having tertiary_base true regardless of what is specified for them.

prebase This attribute gives the prebase attribute for each character matched. The value may be "true" or "false" or a space separated list of such values. If missing the value for all the characters matched is false. It is illegal for a tertiary character to have a true prebase value.

If a primary character has a true prebase value then the character is marked as being typed before the base character of a run, even though it is intended to be stored after it. The primary order gives the intended position in the order after the base character, that the prebase character will end up. Thus @primary may not be 0. These characters are part of the run prefix. If such characters are typed then, in order to give the run a base character after which characters can be sorted, an appropriate base character, such as a dotted circle, is inserted into the output run, until a real base character has been typed. A value of "false" indicates that the character is not a prebase.

There is no @error attribute.

For @from attributes with a match string length greater than 1, the sort key information (@order, @tertiary, @tertiary_base, @prebase) may consist of a space separated list of values, one for each element matched. The last value is repeated to fill out any missing values. Such a list may not contain more values than there are elements in the @from attribute:

```
if len(@from) < len(@list) then error
else
    while len(@from) > len(@list)
        append lastitem(@list) to @list
    endwhile
endif
```

For example, consider the word Northern Thai (nod-Lana) word: 'roasted'. This is ideally encoded as U+1A21 (ka) U+1A60 (asat) U+1A45 (wa) U+1A6B (o) U+1A76 (tone-2). Some users may type the upper component of the vowel first, and the tone before or after the lower component. Thus someone might type it as U+1A21 U+1A6B U+1A76 U+1A60 U+1A45. The Unicode normalization would reorder this to U+1A21 U+1A6B U+1A60 U+1A76 U+1A45. Finally there is the sequence with the tone after the lower component: U+1A21 U+1A6B U+1A60 U+1A45 U+1A76. We want these three sequences to end up ordered as the first. To do this, we use the following rules:

```
<reorder from="\u1A60" order="127"/>       <!-- max possible order -->
<reorder from="\u1A6B" order="42"/>
<reorder from="[\u1A75-\u1A7C]" order="55"/>
<reorder before="\u1A6B" from="\u1A60\u1A45" order="10"/>
<reorder before="\u1A6B[\u1A75-\u1A7C]" from="\u1A60\u1A45" order="10"/>
<reorder before="\u1A6B" from="\u1A60[\u1A75-\u1A7C]\u1A45" order="10 55 10"/>
```

The first reorder is the default ordering for the asat which allows for it to be placed anywhere in a sequence, but moves any non-consonants that may immediately follow it, back before it in the sequence. The next two rules give the orders for the top vowel component and tone marks respectively. The next three rules give the asat and wa characters a primary order that places them before the U+1A6B. Notice particularly the final reorder rule where the asat+wa is split by the tone mark. This rule is necessary in case someone types into the middle of previously normalised text.

Two <reorder> elements A, B intersect if they could match the same string at the same point.

For example:

(using translit notation, where x{y}z means @before="x" @from="y" @after="z")

A: [abc]{[def]}[ghi] → 3
B: [a]{[de]}[gh] → 5

These intersect, since both would match a|e h.

<reorder> elements are ordered, and executed from first to last. See Importing Ordered Elements for how these are inherited. It is thus important to make sure that the reorder elements are correctly ordered, so that an earlier element does not inadvertently “mask” a later element.

For example, the following elements don’t work properly:

A: b{cd}ef → 3
B: ab{cd}ef → 5

That is, all text that would match B would first match on A, so B is rendered useless. However, the following case is different:

A′: b{cd}ef → 3
B′: ab{c[de]}ef → 5

In this case, B does have an effect, if the input string is “bceef”. Such behavior may be desired, so that specific cases can come before more general ones.

Consider this fragment from the Myanmar script:

```
        <reorder from="\u103C" order="20"/>
        <reorder from="[\u103D\u1082]" order="25"/>
        <reorder from="[\u103E\u1082]\u103A" order="27"/>         <!-- Mon asat -->
        <reorder from="[\u103E\u1060]" order="27"/>
        <reorder from="[\u1031\u1084]" order="30"/>
        <reorder from="[\u102D\u102E\u1033-\u1035\u1071-\u1074\u1085\u109D\uA9E5]"
order="35"/>
```

In addition. In the keyboard layout itself has this transforms element:

```
<reorders type="reorder">

    <reorder from="\u1004\u103A\u1039" order="-1"/>
    <reorder from="\u1031" prebase="1"/>
    <reorder from="\u103C" prebase="1"/>

</reorders>
```

The effect of this that the character \u1031 will be identified as a prebase and will have an order of 30. Likewise a \u103C will be identified as a prebase and will have an order of 20. Notice that a U+1084 will not be identified as a prebase (even if it should be!). The kinzi is described in the layout since it moves something across a run boundary. By separating such movements (prebase or moving to in front of a base) from the shared ordering rules, the shared ordering rules become a self-contained combining order description that can be used in other contexts than keyboarding.

final

Each final transforms element is applied after the reorder transform. It executes in a similar way to the simple transform with the settings ignored — that is, as if there were no settings in the <settings> element.

This is an example from Khmer where split vowels are combined after reordering.

```
        <transforms type="final">
                <transform from="\u17C1\u17B8" to="\u17BE"/>
                <transform from="\u17C1\u17B6" to="\u17C4"/>
        </transforms>
```

Another example allows a keyboard implementation to alert or stop people typing two lower vowels in a Burmese cluster, regardless of whether other diacritics are typed in between:

```
        <transform from="[\u102F\u1030\u1048\u1059][\u102F\u1030\u1048\u1059]"
error="true"/>
```

backspace

```
<!ELEMENT backspaces ( backspace+ )>

<!ELEMENT backspace EMPTY>
<!ATTLIST backspace before CDATA #IMPLIED>
<!ATTLIST backspace from CDATA #REQUIRED >
<!ATTLIST backspace after CDATA #IMPLIED>
<!ATTLIST backspace to CDATA #IMPLIED >
        <!--@VALUE-->
<!ATTLIST backspace error (true) #IMPLIED>
        <!--@VALUE-->
```

The backspace transform is an optional transform that is not applied on input of normal characters, but instead may be used to perform extra backspace modifications to previously committed text.

Keyboarding applications typically, but are not required, to work in one of two modes:

> **text entry** happens while a user is typing new text. A user typically wants the backspace key to undo whatever they last typed, whether or not they typed things in the 'right' order.
>
> **text editing** happens when a user moves the cursor into some previously entered text which may have been entered by someone else. As such, there is no way to know in which order things were typed, but a user will still want appropriate behaviour when they press backspace. This may involve deleting more than one character or replacing a sequence of characters with a different sequence.

In the text entry mode, there is no need for any special description of backspace behaviour. A keyboarding application will typically keep a history of previous output states and just revert to the previous state when backspace is hit.

In text editing mode, different keyboard layouts may behave differently in the same textual context. The backspace transform allows the keyboard layout to specify the effect of pressing backspace in a particular textual context. This is done by specifying a set of backspace rules that match a string before the cursor and replace it with another string. The rules are expressed as backspace elements encapsulated in a backspaces element.

The backspace element has the same **before, from, after, to**, and**errors** attributes of the transform element. The processing model is slightly different, however. If we consider the same transform as in the simple transform example, but as a backspace:

```
<backspace before="X" from="Y" after="Z" to="B"/>
```

This would transform the string:

```
XYZ → XBZ
```

If we mark where the current match position is before and after the transform we see:

```
X Y | Z → X B | Z
```

Whereas a simple or final transform would then run other transforms in the transform list, advancing the processing position until it gets to the end of the string, the backspace transform only matches a single backspace rule and then finishes.

For a simple example, consider deleting a Devanagari ksha:

```
<backspaces type="backspace">
        <backspace from="\u0915\u094D\u0936" to=””/>
</backspaces>
```

Here the **to** attribute is empty since the whole string is being deleted. This is not uncommon in the backspace transforms.

A more complex example comes from a Burmese visually ordered keyboard:

```
<backspaces type="backspace">

    <backspace from="[\u1004\u101B\u105A]\u103A\u1039"/>
    <backspace from="\u1039[\u1000-\u101C\u101E\u1020\u1021\u1050\u1051\u105A-\u105D]"/>
    <backspace from="\u102B\u103A"/>
    <!-- Handle prebases -->
    <backspace from="[\u103A-\u103F\u105E-\u1060\u1082]\u1031" to="\u1031"/>
    <backspace from="[\u103A-\u103B\u105E-\u105F]\u103C" to="\u103C"/>
    <backspace from="\u1039[\u1000-\u101C\u101E\u1020\u1021]\u1031" to="\u1031"/>
    <backspace from="[\u1000-\u102A\u103F-\u1049\u104E]\u1031" to="\uFDDF\u1031"/>
    <backspace from="\u1039[\u1000-\u101C\u101E\u1020\u1021]\u103C" to="\u103C"/>
    <backspace from="[\u1000-\u102A\u103F-\u1049\u104E]\u103C" to="\uFDDF\u103C"/>
    <backspace from="\uFDDF[\u1031\u103C]"/>

</backspaces>
```

The character \uFDDF is special. When a keyboard implementation handles a user pressing a key that inserts a prebase character, it should insert a special filler string before the prebase to ensure that the prebase character does not combine with the previous cluster. See the reorder transform for details. The precise filler string is implementation dependent. Rather than requiring keyboard layout designers to know what the filler string is on any given implementation, \uFDDF is is designed to be replaced by the implementation-dependent filler string.

The first three transforms delete various ligatures with a single keypress. The other transforms handle prebase characters. There are two in this Burmese keyboard. The transforms delete the characters preceding the prebase character up to base which gets replaced with the prebase filler string, which represents a null base. Finally the prebase filler string + prebase is deleted as a unit.

Example Keyboard Layout

TBD: add example of Myanmar keyboard the last character before the cursor.

In shared/reorders.xml

```
<!-- Myanmar based on UTN#11 -->
<reorders>
    <reorder
from="[\u1000-\u102C\u103F-\u1049\u104E\u1050-\u1057\u105A-\u105D\u1061-\u1065\u106E-\u1070\u1075-\u1081\u1083\u108E\u109F\uA9E0-\uA9E4\uA9E7-\uA9EF\uA9FA-\uA9FF\uAA60-\uAA6F\uAA71-\uAA76\uAA7A\uAA7E\uAA7F]" order="0"/>
    <reorder from="[\uFE00-\uFE0F]" tertiary="1"/>
    <reorder from="\u1039[\u1000-\u101C\u101E\u1020\u1021\u1050\u1051\u105A-\u105D]" order="5"
tertiary_base="1"/>
    <reorder from="\u103A" order="10"/>
    <reorder from="[\u103B\u105E\u105F]" order="15"/>
    <reorder from="\u103C" order="20"/>
    <reorder from="[\u103D\u1082]" order="25"/>
    <reorder from="[\u103E\u1082]\u103A" order="27"/>      <!-- Mon asat -->
    <reorder from="[\u103E\u1060]" order="27"/>
    <reorder from="[\u1031\u1084]" order="30"/>
    <reorder from="[\u102D\u102E\u1033-\u1035\u1071-\u1074\u1085\u109D\uA9E5]" order="35"/>
    <reorder from="[\u102F\u1030\u1048\u1059]" order="40"/>
    <reorder from="\u1086" order="54"/>
    <reorder from="[\u1032\u1036]" order="55"/>
    <reorder from="\u1037" order="60"/>
    <reorder from="[\u1038\u1087-\u108D\u108F\u109A-\u109C\uAA7B-\uAA7D]" order="65"/>
    <reorder from="[\uA9E6\uAA70]" order="70"/>
</reorders>
```

In a visual-order keyboard that inherits the above.

```
<transforms type="simple">
    <!-- ` deadkey is same as ctrl+alt -->
    <transform from="`\u1041" to="\u100D"/>
    <transform from="`\u1042" to="\u1039\u100C"/>
    <transform from="`\U{1043}" to="\u100B"/>
    <transform from="`\u1044" to="\u104E\u1004\u103A\u1038"/>
    <transform from="`\u1045" to="\u1029"/>
    <transform from="`\u1047" to="\u101B"/>
    <transform from="`\u1040" to="\u101D"/>
    <transform from="`-" to="\u100B"/>
    <transform from="`=" to="\u100D"/>
...
</transforms>

<import path="../../../../shared/reorders.xml"/>
<reorders>
    <!-- only add information or override not replace -->
    <reorder from="\u1031" prebase="1"/>
    <reorder from="\u103C" prebase="1"/>
    <reorder from="[\u1004\u101B\u105A]\u103A\u1039" order="-1"/>
</reorders>
<backspaces>
    <backspace from="[\u1004\u101B\u105A]\u103A\u1039"/>
    <backspace from="\u1039[\u1000-\u101C\u101E\u1020\u1021\u1050\u1051\u105A-\u105D]"/>
    <backspace from="\u102B\u103A"/>
    <!-- Handle prebases -->
    <backspace from="[\u103A-\u103F\u105E-\u1060\u1082]\u1031" to="\u1031"/>
    <backspace from="[\u103A-\u103B\u105E-\u105F]\u103C" to="\u103C"/>
    <backspace from="\u1039[\u1000-\u101C\u101E\u1020\u1021\u1050\u1051\u105A-\u105D]\u1031"
to="\u1031"/>
    <backspace from="[\u1000-\u102A\u103F-\u1049\u104E\u1050-\u1055\u105A-\u105D]\u1031"
to="\u25CC\u1031"/>
    <backspace from="\u1039[\u1000-\u101C\u101E\u1020\u1021\u1050\u1051\u105A-\u105D]\u103C"
to="\u103C"/>
    <backspace from="[\u1000-\u102A\u103F-\u1049\u104E\u1050-\u1055\u105A-\u105D]\u103C"
to="\u25CC\u103C"/>
    <backspace from="\u25CC[\u1031\u103C]"/>
</backspaces>
```