Representing Label Generation Rulesets using XML
draft-ietf-lager-specification-13

Abstract

   This document describes a method of representing rules for validating
   identifier labels and alternate representations of those labels using
   Extensible Markup Language (XML). These policies, known as "Label
   Generation Rulesets" (LGRs), are used for the implementation of
   Internationalized Domain Names (IDNs), for example. The rulesets are
   used to implement and share that aspect of policy defining which
   labels and Unicode code points are permitted for registrations, which
   alternative code points are considered variants, and what actions may
   be performed on labels containing those variants.

Status of This Memo

Copyright Notice

carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction

   This document specifies a method of using Extensible Markup Language
   (XML) to describe Label Generation Rulesets (LGRs). LGRs are
   algorithms used to determine whether, and under what conditions, a
   given identifier label is permitted, based on the code points it
   contains and their context. These algorithms comprise a list of
   permissible code points, variant code point mappings, and a set of
   rules that act on the code points and mappings. LGRs form part of an
   administrator's policies. In deploying internationalized domain
   names (IDNs), they have also been known as IDN tables or variant
   tables.

   There are other kinds of policies relating to labels which are not
   normally covered by Label Generation Rulesets and are therefore not
   necessarily representable by the XML format described here. These
   include, but are not limited to policies around trademarks, or
   prohibition of fraudulent or objectionable words.

   Administrators of the zones for top-level domain registries have
   historically published their LGRs using ASCII text or HTML. The
   formatting of these documents has been loosely based on the format
   used for the Language Variant Table described in [RFC3743].
   [RFC4290] also provides a "model table format" that describes a
   similar set of functionality. Common to these formats is that the
   algorithms used to evaluate the data therein are implicit or
   specified elsewhere.

   Through the first decade of IDN deployment, experience has shown that
   LGRs derived from these formats are difficult to consistently
   implement and compare due to their differing formats. A universal
   format, such as one using a structured XML format, will assist by
   improving machine-readability, consistency, reusability and
   maintainability of LGRs.

   When used to represent simple list of permitted code points, the
   format is quite straightforward. At the cost of some complexity in
   the resulting file, it also allows for an implementation of more
   sophisticated handling of conditional variants that reflects the
   known requirements of current zone administrator policies.

   Another feature of this format is that it allows many of the
   algorithms to be made explicit and machine implementable. A
   remaining small set of implicit algorithms is described in this
   document to allow commonality in implementation.

   While the predominant usage of this specification is to represent IDN
   label policy, the format is not limited to IDN usage and may also be

used for describing ASCII domain name label rulesets, or other types
of identifier labels beyond those used for domain names.

2. Design Goals

   The following goals informed the design of this format:

   o The format needs to be implementable in a reasonably
      straightforward manner in software.

   o The format should be able to be automatically checked for
      formatting errors, so that common mistakes can be caught.

   o An LGR needs to be able to express the set of valid code points
      that are allowed for registration under a specific administrator's
      policies.

   o An LGR needs to be able to express computed alternatives to a
      given identifier based on mapping relationships between code
      points, whether one-to-one or many-to-many. These computed
      alternatives are commonly known as "variants".

   o Variant code points should be able to be tagged with explicit
      dispositions or categories that can be used to support registry
      policy (such as whether to allocate the computed variant, or to
      merely block it from usage or registration).

   o Variants and code points must be able to be stipulated based on
      contextual information. For example, some variants may only be
      applicable when they follow a certain code point, or when the code
      point is displayed in a specific presentation form.

   o The data contained within an LGR must be able to be interpreted
      unambiguously, so that independent implementations that utilize
      the contents will arrive at the same results.

   o To the largest extent possible, policy rules should be able to be
      specified in the XML format without relying on hidden, or built-in
      algorithms in implementations.

   o LGRs should be suitable for comparison and re-use, such that one
      could easily compare the contents of two or more to see the
      differences, to merge them, and so on.

   o As many existing IDN tables as practicable should be able to be
      migrated to the LGR format with all applicable interpretation
      logic retained.

These requirements are partly derived from reviewing the existing corpus of published IDN tables, plus the requirements of ICANN's work to implement an LGR for the DNS Root Zone [LGR-PROCEDURE]. In particular, Section B of that document identifies five specific requirements for an LGR methodology.

The syntax and rules in [RFC5892] and [RFC3743] were also reviewed.

It is explicitly not the goal of this format to stipulate what code points should be listed in an LGR by a zone administrator. Which registration policies are used for a particular zone is outside the scope of this memo.

3. Normative Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

4. LGR Format

An LGR is expressed as a well-formed XML Document [XML] that conforms to the schema defined in Appendix D.

As XML is case-sensitive, an LGR must be authored with the correct casing. For example, the XML element names MUST be in lower case as described in this specification, and matching of attribute values, is only performed in a case-sensitive manner.

A document that is not well-formed, non-conforming or violates other constraints specified in this specification MUST be rejected.

4.1. Namespace

The XML Namespace URI is "urn:ietf:params:xml:ns:lgr-1.0".

See Section 11.2 for more information.

4.2. Basic Structure

The basic XML framework of the document is as follows:

```
<?xml version="1.0"?>
<lgr xmlns="urn:ietf:params:xml:ns:lgr-1.0">
    ...
</lgr>
```

The "lgr" element contains up to three sub-elements. First is an
optional "meta" element that contains all meta-data associated with
the LGR, such as its authorship, what it is used for, implementation
notes and references. This is followed by a required "data" element
that contains the substantive code point data. Finally, an optional
"rules" element contains information on contextual and whole-label
evaluation rules, if any, along with "action" elements providing for
the disposition of labels and computed variant labels.

```
<?xml version="1.0"?>
<lgr xmlns="urn:ietf:params:xml:ns:lgr-1.0">
    <meta>
        ...
    </meta>
    <data>
        ...
    </data>
    <rules>
        ...
    </rules>
</lgr>
```

A document MUST contain exactly one "lgr" element. Each "lgr"
element MUST contain zero or one "meta" element, exactly one "data"
element, and zero or one "rules" element; and these three elements
MUST be in that order.

Some elements that are direct or nested child elements of the "rules"
element MUST be placed in a specific relative order to other elements
for the LGR to be valid. An LGR that violates these constraints MUST
be rejected. In other cases, changing the ordering would result in a
valid, but different specification.

In the following descriptions, required, non-repeating elements or
attributes are generally not called out explicitly, in contrast to
"OPTIONAL" ones, or those that "MAY" be repeated. For attributes
that take lists as values, the elements MUST be space-separated.

4.3. Metadata

The "meta" element expresses metadata associated with the LGR, and
the element SHOULD be included so that the associated metadata are
available as part of the LGR and cannot become disassociated. The
following subsections describe elements that may appear within the
"meta" element.

The "meta" element can be used to identify the author or relevant
contact person, explain the intended usage of the LGR, and provide

implementation notes as well as references. Detailed metadata allow the LGR document to become self-documenting, for example if rendered in a human readable format by an appropriate tool.

Providing metadata pertaining to the date and version of the LGR is particularly encouraged to make it easier for interoperating consumers to ensure that they are using the correct LGR.

With the exception of "unicode-version" element, the data contained within is not required by software consuming the LGR in order to calculate valid labels, or to calculate variants. If present, the "unicode-version" element MUST be used by a consumer of the table to identify that it has the correct Unicode property data to perform operations on the table. This ensures that possible differences in code point properties between editions of the Unicode standard do not impact the product of calculations utilizing an LGR.

4.3.1. The version Element

The "version" element is OPTIONAL. It is used to uniquely identify each version of the LGR. No specific format is required, but it is RECOMMENDED that it be the decimal representation of a single positive integer, which is incremented with each revision of the file.

An example of a typical first edition of a document:

    <version>1</version>

The "version" element may have an OPTIONAL "comment" attribute.

    <version comment="draft">1</version>

4.3.2. The date Element

The OPTIONAL "date" element is used to identify the date the LGR was posted. The contents of this element MUST be a valid ISO 8601 "full-date" string as described in [RFC3339].

Example of a date:

    <date>2009-11-01</date>

4.3.3. The language Element

Each OPTIONAL "language" element identifies a language or script for which the LGR is intended. The value of the "language" element MUST be a valid language tag as described in [RFC5646]. The tag may refer

to a script plus undefined language if the LGR is not intended for a specific language.

Example of an LGR for the English language:

    <language>en</language>

If the LGR applies to a script, rather than a specific language, the "und" language tag SHOULD be used followed by the relevant [RFC5646] script subtag. For example, for a Cyrillic script LGR:

    <language>und-Cyrl</language>

If the LGR covers a set of multiple languages or scripts, the "language" element MAY be repeated. However, for cases of a script-specific LGR exhibiting insignificant admixture of code points from other scripts, it is RECOMMENDED to use a single "language" element identifying the predominant script. In the exceptional case of a multi-script LGR where no script is predominant, use Zyyy (Common):

    <language>und-Zyyy</language>

4.3.4. The scope Element

This OPTIONAL element refers to a scope, such as a domain, to which this policy is applied. The "type" attribute specifies the type of scope being defined. A type of "domain" means that the scope is a domain that represents the apex of the DNS zone to which the LGR is applied. For that type, the content of the "scope" element MUST be a a domain name written relative to the root zone, in presentation format with no trailing dot. However, in the unique case of the DNS root zone, it is represented as ".".

    <scope type="domain">example.com</scope>

There may be multiple "scope" tags used, for example to reflect a list of domains to which the LGR is applied.

No other values of the "type" attribute are defined by this specification, however this specification can be used for applications other than domain names. Implementers of LGRs for applications other than domain names SHOULD define the scope extension grammar in an IETF Specification, or use XML Namespaces to distinguish their scoping mechanism distinctly from the base LGR namespace. An explanation of any custom usage of the scope in the "description" element is RECOMMENDED.

```
<scope xmlns="http://example.com/ns/scope/1.0">
    ... content per alternate namespace ...
</scope>
```

4.3.5. The description Element

The "description" element is an OPTIONAL, free-form element that
contains any additional relevant description that is useful for the
user in its interpretation. Typically, this field contains
authorship information, as well as additional context on how the LGR
was formulated and how it applies, such as citations and references
that apply to the LGR as a whole.

This field should not be relied upon for providing instructions on
how to parse or utilize the data contained elsewhere in the
specification. Authors of tables should expect that software
applications that parse and use LGRs will not use the description
field to condition the application of the LGR's data and rules.

The element has an OPTIONAL "type" attribute, which refers to the
internet media type [RFC2045] of the enclosed data. Typical types
would be "text/plain" or "text/html". The attribute SHOULD be a
valid media type. If supplied, it will be assumed that the contents
are of that media type. If the description lacks a type field, it
will be assumed to be plain text ("text/plain").

4.3.6. The validity-start and validity-end Elements

The "validity-start" and "validity-end" elements are OPTIONAL
elements that describe the time period from which the contents of the
LGR become valid (are used in registry policy), and time when the
contents of the LGR cease to be used, respectively.

The dates MUST confirm to the "full-date" format described in section
5.6 of [RFC3339].

```
<validity-start>2014-03-12</validity-start>
```

4.3.7. The unicode-version Element

Whenever an LGR depends on character properties from a given version
of the Unicode standard, the version number used in creating the LGR
MUST be listed in the form x.y.z, where x, y, and z are positive,
decimal integers (see [Unicode-Versions]). If any software
processing the table does not have access to character property data
of the requisite version, it MUST NOT perform any operations relating
to whole-label evaluation relying on Unicode character properties
(Section 6.2.3).

The value of a given Unicode character property may change between
versions of the Unicode Character Database [UAX44], unless such
change has been explicitly disallowed in [Unicode-Stability]. It is
RECOMMENDED to only reference properties defined as stable or
immutable. As an alternative to referencing the property, the
information can be presented explicitly in the LGR.

```
<unicode-version>6.2.0</unicode-version>
```

It is not necessary to include a "unicode-version" element for LGRs
that do not make use of Unicode character properties, however, it is
RECOMMENDED.

4.3.8. The references Element

A Label Generation Ruleset may define a list of references which are
used to associate various individual elements in the LGR to one or
more normative references. A common use for references is to
annotate that code points belong to an externally defined collection
or standard, or to give normative references for rules.

References are specified in an OPTIONAL "references" element,
containing or more "reference" elements, each with a unique "id"
attribute. It is RECOMMENDED that the "id" attribute be a zero-based
integer, however, in addition to digits 0-9, it MAY contain uppercase
letters A-Z, as well as period, hyphen, colon or underscore. The
value of each "reference" element SHOULD be the citation of a
standard, dictionary or other specification in any suitable format.
In addition to an "id" attribute, a "reference" element MAY have a
"comment" attribute for an optional free-form annotation.

```
<references>
  <reference id="0">The Unicode Consortium. The Unicode
    Standard, Version 8.0.0, (Mountain View, CA: The Unicode
    Consortium, 2015. ISBN 978-1-936213-10-8)
    http://www.unicode.org/versions/Unicode8.0.0/</reference>
  <reference id="1">Big-5: Computer Chinese Glyph and Character
    Code Mapping Table, Technical Report C-26, 1984</reference>
  <reference id="2" comment="synchronized with Unicode 6.1">
    ISO/IEC
    10646:2012 3rd edition</reference>
  ...
</references>
...
<data>
  <char cp="0620" ref="0 2" />
  ...
</data>
```

A reference is associated with an element by using its id as part of an optional "ref" attribute (see Section 5.4.1). The "ref" attribute may be used with many kinds of elements in the "data" or "rules" sections of the LGR, most notably those defining code points, variants and rules. However, a "ref" attribute may not occur on certain kinds of elements, including references to named character classes or rules. See description of these elements below.

5. Code Points and Variants

The bulk of a label generation ruleset is a description of which set of code points are eligible for a given label. For rulesets that perform operations that result in potential variants, the code point-level relationships between variants need to also be described.

The code point data is collected within the "data" element. Within this element, a series of "char" and "range" elements describe eligible code points, or ranges of code points, respectively. Collectively, these are known as the repertoire.

Discrete permissible code points or code point sequences (see Section 5.1) are declared with a "char" element. Here is a minimal example declaration for a single code point, with the code point value given in the "cp" attribute:

    <char cp="002D"/>

As described below, a full declaration for a "char" element, whether or not it is used for a single code point, or for a sequence (see Section 5.1), may have optional child elements defining variants. Both the "char" and "range" elements can take a number of optional attributes for conditional inclusion, commenting, cross referencing and character tagging, as described below.

Ranges of permissible code points may be declared with a "range" element, as in this minimal example:

    <range first-cp="0030" last-cp="0039"/>

The range is inclusive of the first and last code points. Any additional attributes defined for a "range" element act as if applied to each code point within. A "range" element has no child elements.

It is always possible to substitute a list of individually specified code points for a range element. The reverse is not necessarily the case. Whenever such a substitution is possible, it makes no difference in processing the data. Tools reading or writing the LGR

format are free to aggregate sequences of consecutive code points of
the same properties into range elements.

Code points MUST be represented according to the standard Unicode
convention but without the prefix "U+": they are expressed in
uppercase hexadecimal, and are zero-padded to a minimum of 4 digits.

The rationale for not allowing other encoding formats, including
native Unicode encoding in XML, is explored in [UAX42]. The XML
conventions used in this format, such as element and attribute names,
mirror this document where practical and reasonable to do so. It is
RECOMMENDED to list all "char" elements in ascending order of the
"cp" attribute. Not doing so makes it unnecessarily difficult for
authors and reviewers to check for errors, such as duplications, or
to review and compare against listing of code points in other
documents and specifications.

All "char" elements in the data section MUST have distinct "cp"
attributes. The "range" elements MUST NOT specify code point ranges
that overlap either another range or any single code point "char"
elements. An LGR that defines the same code point more than once by
any combination of "char" or "range" elements MUST be rejected.

5.1. Sequences

A sequence of two or more code points may be specified in an LGR, for
example, when defining the source for n:m variant mappings. Another
use of sequences would be in cases when the exact sequence of code
points is required to occur in order for the constituent elements to
be eligible, such as when some code point is only eligible when
preceded or followed by a certain code point. The following would
define the eligibility of the MIDDLE DOT (U+00B7) only when both
preceded and followed by the LATIN SMALL LETTER L (U+006C):

    <char cp="006C 00B7 006C" comment="Catalan middle dot"/>

All sequences defined this way must be distinct, but sub-sequences
may be defined. Thus, the sequence defined here may coexist with
single code point definitions such as:

    <char cp="006C" />

As an alternative to using sequences to define a required context, a
"char" or "range" element may specify conditional context using an
optional "when" attribute as described below in Section 5.2. Using a
conditional context is more flexible because a context is not limited
to a specific sequence of code points. In addition, using a context

allows the choice of specifying either a prohibited or a required
context.

5.2. Conditional Contexts

   A conditional context is specified by a rule that must be satisfied
   (or alternatively, must not be satisfied) for a code point in a given
   label, often at a particular location in a label.

   To specify a conditional context either a "when" or "not-when"
   attribute may be used. The value of each "when" or "not-when"
   attributes is a whole label or parameterized context rule as
   described below in Section 6.3. The context condition is met when
   the rule specified in the "when" attribute is matched or when the
   rule in the "not-when" attribute fails to match. It is an error to
   reference a rule that is not actually defined in the "rules" element.

   A parameterized context rule (see Section 6.4) defines the context
   immediately surrounding a given code point; unlike a sequence, the
   context is not limited to a specific fixed code point, but for
   example may designate any member of a certain character class or a
   code point that has a certain Unicode character property.

   Given a suitable definition of a parameterized context rule named
   "follows-virama" this example specifies that a ZERO-WIDTH JOINER
   (U+200D) is restricted to immediately follow any of several code
   points classified as virama:

       <char cp="200D" when="follows-virama" />

   For a complete example, see Appendix A.

   In contrast, a whole label rule (see Section 6.3) specifies a
   condition to be met by the entire label, for example that it must
   contain at least one code point from a given script anywhere in the
   label. In the following example, no digit from either range may
   occur in a label that mixes digits from both ranges:

       <data>
          <range first-cp="0660" last-cp="0669" not-when="mixed-digits"
                 tag="arabic-indic-digits" />
          <range first-cp="06F0" last-cp="06F9" not-when="mixed-digits"
                 tag="extended-arabic-indic-digits" />
       </data>

   (See Section 6.3.9 for an example of the "mixed-digits" rule.)

The OPTIONAL "when" or "not-when" attributes are mutually exclusive. They MAY be applied to both "char" and "range" elements in the "data" element, including "char" elements defining sequences of code points, as well as to "var" elements (see Section 5.3.5).

If a label contains one or more code points that fail to satisfy a conditional context, the label is invalid, see Section 7.5. For variants, the conditional context restricts the definition of the variant to the case where the condition is met. Outside the specified context, a variant is not defined.

5.3. Variants

Most LGRs typically only determine simple code point eligibility, and for them, the elements described so far would be the only ones required for their "data" section. Others additionally specify a mapping of code points to other code points, known as "variants". What constitutes a variant code point is a matter of policy, and varies for each implementation. The following examples are intended to demonstrate the syntax; they are not necessarily typical.

5.3.1. Basic Variants

Variant code points are specified using one of more "var" elements as children of a "char" element. The target mapping is specified using the "cp" attribute. Other, optional attributes for the "var" element are described below.

For example, to map LATIN SMALL LETTER V (U+0076) as a variant of LATIN SMALL LETTER U (U+0075):

```
<char cp="0075">
    <var cp="0076"/>
</char>
```

A sequence of multiple code points can be specified as a variant of a single code point. For example, the sequence of LATIN SMALL LETTER O (U+006F) then LATIN SMALL LETTER E (U+0065) might hypothetically be specified as a variant for an LATIN SMALL LETTER O WITH DIAERESIS (U+00F6) as follows:

```
<char cp="00F6">
    <var cp="006F 0065"/>
</char>
```

The source and target of a variant mapping may both be sequences, but not ranges.

If the source of one mapping is a prefix sequence of the source for
another, both variant mappings will be considered at the same
location in the input label when generating permuted variant labels.
If poorly designed, an LGR containing such an instance of a prefix
relation could generate multiple instances of the same variant label
for the same original label, but with potentially different
dispositions. Any duplicate variant labels encountered MUST be
treated as an error (see Section 8.4).

The "var" element specifies variant mappings in only one direction,
even though the variant relation is usually considered symmetric,
that is, if A is a variant of B then B should also be a variant of A.
The format requires that the inverse of the variant be given
explicitly to fully specify symmetric variant relations in the LGR.
This has the beneficial side effect of making the symmetry explicit:

```
<char cp="006F 0065">
    <var cp="00F6"/>
</char>
```

Variant relations are normally not only symmetric, but also
transitive. If A is a variant of B and B is a variant of C, then A
is also a variant of C. As with symmetry, these transitive relations
are only part of the LGR if spelled out explicitly. Implementations
that require an LGR to be symmetric and transitive should verify this
mechanically.

All variant mappings are unique. For a given "char" element all
"var" elements MUST have a unique combination of "cp", "when" and
"not-when" attributes. It is RECOMMENDED to list the "var" elements
in ascending order of their target code point sequence. (For "when"
and "not-when" attributes, see Section 5.3.5).

5.3.2. The type attribute

Variants may be tagged with an OPTIONAL "type" attribute. The value
of the "type" attribute may be any non-empty value not starting with
an underscore and not containing spaces. This value is used to
resolve the disposition of any variant labels created using a given
variant. (See Section 7.2.)

By default, the values of the "type" attribute directly describe the
target policy status (disposition) for a variant label that was
generated using a particular variant, with any variant label being
assigned a disposition corresponding to the most restrictive variant
type. Several conventional disposition values are predefined below
in Section 7. Whenever these values can represent the desired
policy, they SHOULD be used.

```
    <char cp="767C">
        <var cp="53D1" type="allocatable"/>
        <var cp="5F42" type="blocked"/>
        <var cp="9AEA" type="blocked"/>
        <var cp="9AEE" type="blocked"/>
    </char>
```

By default, if a variant label contains any instance of one of the
variants of type "blocked" the label would be blocked, but if it
contained only instances of variants to be allocated it could be
allocated. See the discussion about implied actions in Section 7.6.

The XML format for the LGR makes the relation between the values of
the "type" attribute on variants and the resulting disposition of
variant labels fully explicit. See the discussion in Section 7.2.
Making this relation explicit allows a generalization of the "type"
attribute from directly reflecting dispositions to a more
differentiated intermediate value that is then used in the resolution
of label disposition. Instead of the default action of applying the
most restrictive disposition to the entire label, such a generalized
resolution can be used to achieve additional goals, such as limiting
the set of allocatable variant labels, or to implement other policies
found in existing LGRs (see for example Appendix B).

Because variant mappings MUST be unique, it is not possible to define
the same variant for the same "char" element with different type
attributes (see however Section 5.3.5).

5.3.3. Null Variants

A null variant is a variant string that maps to no code point. This
is used when a particular code point sequence is considered
discretionary in the context of a whole label. To specify a null
variant, use an empty cp attribute. For example, to mark a string
with a ZERO WIDTH NON-JOINER (U+200C) to the same string without the
ZERO WIDTH NON-JOINER:

```
    <char cp="200C">
        <var cp=""/>
    </char>
```

This is useful in expressing the intent that some code points in a
label are to be mapped away when generating a canonical variant of
the label. However, in tables that are designed to have symmetric
variant mappings, this could lead to combinatorial explosion, if not
handled carefully.

The symmetric form of a null variant is expressed as follows:

```
    <char cp="">
        <var cp="200C" type="invalid" />
    </char>
```

A "char" element with an empty "cp" attribute MUST specify at least
one variant mapping. It is strongly RECOMMENDED to use a type of
"invalid" or equivalent when defining variant mappings from null
sequences, so that variant mapping from null sequences are removed in
variant label generation (see Section 5.3.2).

5.3.4. Variants with Reflexive Mapping

At first sight there seems to be no call for adding variant mappings
for which source and target code points are the same, that is for
which the mapping is reflexive, or, in other words, an identity
mapping. Yet such reflexive mappings occur frequently in LGRs that
follow [RFC3743].

Adding a "var" element allows both a type and a reference id to be
specified for it. While the reference id is not used in processing,
the type of the variant can be used to trigger actions. In permuting
the label to generate all possible variants, the type associated with
a reflexive variant mapping is applied to any of the permuted labels
containing the original code point.

In the following example, let's assume the goal is to allocate only
those labels that contain a variant that is considered "preferred" in
some way. As defined in the example, the code point U+3473 exists
both as a variant of U+3447 and as a variant of itself (reflexive
mapping). Assuming an original label of "U+3473 U+3447", the
permuted variant "U+3473 U+3473" would consist of the reflexive
variant of U+3473 followed by a variant of U+3447. Given the variant
mappings as defined here, the types for both of the variant mappings
used to generate that particular permutation would have the value
"preferred":

```
     <char cp="3447" ref="0">
      <var cp="3473" type="preferred" ref="1 3" />
     </char>
     <char cp="3473" ref="0">
       <var cp="3447" type="blocked" ref="1 3" />
       <var cp="3473" type="preferred" ref="0" />
     </char>
```

Having established the variant types in this way, a set of actions
could be defined that return a disposition of "allocatable" or
"activated" for a label consisting exclusively of variants with type

"preferred" for example. (For details on how to define actions based
on variant types see Section 7.2.1.)

In general, using reflexive variant mappings in this manner makes it
possible to calculate disposition values using a uniform approach for
all labels, whether they consist of mapped variant code points,
original code points, or a mixture of both. In particular, the
dispositions for two otherwise identical labels may differ based on
which variant mappings were executed in order to generate each of
them. (For details on how to generate variants and evaluate
dispositions, see Section 8.)

Another useful convention that uses reflexive variants is described
below in Section 7.2.1.

5.3.5. Conditional Variants

Fundamentally, variants are mappings between two sequences of code
points. However, in some instances for a variant relationship to
exist, some context external to the code point sequence must also be
considered. For example, a positional context may determine whether
two code point sequences are variants of each other.

An example of that are Arabic code points which can have different
forms based on position, with some code points sharing forms, thus
making them variants in the positions corresponding to those forms.
Such positional context cannot be solely derived from the code point
by itself, as the code point would be the same for the various forms.

As described in Section 5.2 an OPTIONAL "when" or "not-when"
attribute may be given for any "var" element to specify required or
prohibited contextual conditions under which the variant defined.

Assuming the "rules" element contains suitably defined rules for
"arabic-isolated" and "arabic-final", the following example shows how
to mark ARABIC LETTER ALEF WITH WAVY HAMZA BELOW (U+0673) as a
variant of ARABIC LETTER ALEF WITH HAMZA BELOW (U+0625), but only
when it appears in its isolated or final forms:

```
    <char cp="0625">
        <var cp="0673" when="arabic-isolated"/>
        <var cp="0673" when="arabic-final"/>
    </char>
```

While a "var" element MUST NOT contain multiple conditions (it is
only allowed a single "when" or "not-when" attribute), multiple "var"
elements using the same mapping MAY be specified with different

"when" or "not-when" attributes. The combination of mapping and
conditional context defines a unique variant..

Care must be taken to ensure that for each variant label at most one
of the contextual conditions is met for variants with the same
mapping; otherwise duplicate variant labels would be created for the
same input label. Any such duplicate variant labels MUST be treated
as an error, see Section 8.4.

Two contexts may be complementary, as in the following example, which
shows ARABIC LETTER TEH MARBUTA (U+0629) as a variant of ARABIC
LETTER HEH (U+0647), but with two different types.

```
<char cp="0647" >
  <var cp="0629" not-when="arabic-final" type="blocked" />
  <var cp="0629" when="arabic-final" type="allocatable" />
</char>
```

The intent is that in final position a label that uses U+0629 instead
of U+0647 should be considered essentially the same label and
therefore allocatable to the same entity, while the same substitution
in non-final context leads to labels that are different, but
considered confusable so that either one, but not both should be
delegatable.

For symmetry, the reverse mappings must exist, and must agree in
their "when" or "not-when" attributes. However, symmetry does not
apply to the other attributes. For example, these are potential
reverse mappings for the above:

```
<char cp="0629" >
  <var cp="0647" not-when="arabic-final" type="allocatable" />
  <var cp="0647" when="arabic-final" type="allocatable" />
</char>
```

Here, both variants have the same "type" attribute. While it is
tempting to recognize that in this instance the "when" and "not-when"
attributes are complementary and therefore between them cover every
single possible context, it is strongly RECOMMENDED to use the format
shown in the example that makes the symmetry easily verifiable by
parsers and tools. (The same applies to entries created for
transitivity.)

Arabic is an example of a script for which such conditional variants
have been implemented based on the joining contexts for Arabic code
points. The mechanism defined here supports other forms of
conditional variants that may required by other scripts.

5.4. Annotations

   Two attributes, the "ref" and "comment" attributes, can be used to
   annotate individual elements in the LGR. They are ignored in
   machine-processing or the LGR. The "ref" attribute is intended for
   formal annotations and the "comment" attribute for free form
   annotations. The latter can be applied more widely.

5.4.1. The ref Attribute

   Reference information MAY optionally be specified by a "ref"
   attribute, consisting of a space delimited sequence of reference
   identifiers (see Section 4.3.8).

```
   <char cp="5220" ref="0">
       <var cp="5220" ref="5"/>
       <var cp="522A" ref="2 3"/>
   </char>
```

   This facility is typically used to give source information for code
   points or variant relations. This information is ignored when
   machine-processing an LGR. If applied to a range the "ref" attribute
   applies to every code point in the range. All reference identifiers
   MUST be from the set declared in the "references" element (see
   Section 4.3.8). It is an error to repeat a reference identifier in
   the same "ref" attribute. It is RECOMMENDED that identifiers be
   listed in ascending order.

   In addition to "char", "range" and "var" elements in the data
   section, a "ref" attribute may be present for a number of elements
   types contained in the "rules" element as described below: actions,
   literals ("char" inside a rule), as well as for definitions of rules
   and classes, but not for references to named character classes or
   rules using the "by-ref" attribute defined below. (The use of the
   "by-ref" and "ref" attributes is mutually exclusive.) None of the
   elements in the metadata take a "ref" attribute; to provide
   additional information use the "description" element instead.

5.4.2. The comment Attribute

   Any "char", "range" or "variant" element in the data section may
   contain an OPTIONAL "comment" attribute. The contents of a "comment"
   attribute are free-form plain text. Comments are ignored in machine
   processing of the table. Comment attributes MAY also be placed on
   all elements in the "rules" section of the document, such as actions
   and match operators, such as literals ("char"), as well as
   definitions of classes and rules, but not on child elements of the
   "class" element. Finally, in the metadata, only the "version" and

"reference" elements MAY have "comment" attributes (to match the
syntax in [RFC3743]).

5.5. Code Point Tagging

Typically, LGRs are used to explicitly designate allowable code
points, where any label that contains a code point not explicitly
listed in the LGR is considered an ineligible label according to the
ruleset.

For more complex registry rules, there may be a need to discern one
or more subsets of code points. This can be accomplished by applying
an OPTIONAL "tag" attribute to "char" or "range" elements that are
child elements of the "data" element. By collecting code points that
share the same tag value, character classes may be defined (see
Section 6.2.2) which can then be used in whole label evaluation rules
(see Section 6.3.2).

Each "tag" attribute MAY contain multiple values separated by white
space. A tag value is an identifier, which may also include certain
punctuation marks, such as colon. Formally, it MUST correspond to
the XML 1.0 Nmtoken (Name token) production (see [XML] Section 2.3).
It is an error to duplicate a value within the same "tag" attribute.
A "tag" attribute for a "range" element applies to all code points in
the range. Because code point sequences are not proper members of a
set of code points, a "tag" attribute MUST NOT be present in a "char"
element defining a code point sequence.

6. Whole Label and Context Evaluation

6.1. Basic Concepts

The "rules" element contains the specification of both context-based
and whole Whole Label Evaluation (WLE) rules (Section 6.3), the
character classes (Section 6.2) that they depend on and any actions
(Section 7) that assign dispositions to labels based on rules or
variant mappings.

A Whole Label Evaluation rule (WLE) is applied to the whole label.
It is used to validate both original labels and any variant labels
computed from them.

A conditional context rule does not necessarily apply to the whole
label, but may be specific to the context around a single code point
or code point sequence. Certain code points in a label sometimes
need to satisfy context-based rules, for example for the label to be
considered valid, or to satisfy the context for a variant mapping
(see the description of the "when" attribute in Section 6.4).

   For example, if a rule is referenced in the "when" attribute of a
   variant mapping it is used to describe the conditional context under
   which the particular variant mapping is defined to exist.

   Each rule is defined in a "rule" element. A rule may contain the
   following as child elements:

   o literal code points or code point sequences

   o character classes, which define sets of code points to be used for
      context comparisons

   o context operators, which define when character classes and
      literals may appear

   o nested rules, whether defined in place or invoked by reference

   Collectively, these are called match operators and are listed in
   Section 6.3.2. An LGR containing incorrectly defined or nested rules
   or match operators, or rules and match operators with invalid
   attributes or invalid or undefined attribute values MUST be rejected.
   Note that not all of the constraints defined here are validated by
   the schema.

6.2. Character Classes

   Character classes are sets of characters that often share a
   particular property. While they function like sets in every way,
   even supporting the usual set operators, they are called character
   classes here in a nod to the use of that term in regular expression
   syntax. (This also avoids confusion with the term "character set" in
   the sense of character encoding.)

   Character classes can be specified in several ways:

   o by defining the class via matching a tag in the code point data.
      All characters with the same "tag" attribute are part of the same
      class;

   o by referencing a value of one of the Unicode character properties
      defined in the Unicode Character Database;

   o by explicitly listing all the code points in the class; or

   o by defining the class as a set combination of any number of other
      classes.

6.2.1. Declaring and Invoking Named Classes

   A character class has an OPTIONAL "name" attribute, consisting of a
   single, identifier not containing spaces. All names for classes must
   be unique. If the "name" attribute is omitted, the class is
   anonymous and exists only inside the rule or combined class where it
   is defined. A named character class is defined independently and can
   be referenced by name from within any rules or as part of other
   character class definitions.

```
<class name="example" comment="an example class definition">
    0061 4E00
</class>
...
<rule>
    <class by-ref="example" />
</rule>
```

   An empty "class" element with a "by-ref" attribute is a reference to
   an existing named class. The "by-ref" attribute MUST NOT be used in
   the same "class" element with any of these attributes: "name", "from-
   tag", "property" or "ref". The "name" attribute MUST be present, if
   and only if the class is a direct child element of the "rules"
   element. It is an error to reference a named class for which the
   definition has not been seen.

6.2.2. Tag-based Classes

   The "char" or "range" elements that are child elements of the "data"
   element MAY contain a "tag" attribute that consists of one or more
   space separated tag values, for example:

```
<char cp="0061" tag="letter lower"/>
<char cp="4E00" tag="letter"/>
```

   This defines two tags for use with code point U+0061, the tag
   "letter" and the tag "lower". Use

```
<class name="letter" from-tag="letter" />
<class name="lower" from-tag="lower" />
```

   to define two named character classes, "letter" and "lower",
   containing all code points with the respective tags, the first with
   0061 and 4E00 as elements and the latter with 0061, but not 4E00 as
   an element. The "name" attribute may be omitted for an anonymous in-
   place definition of a nested, tag-based class.

Tag values are typically identifiers, with the addition of a few
punctuation symbols, such as colon. Formally they MUST correspond to
the XML 1.0 Nmtoken (Name token) production. While a "tag" attribute
may contain a list of tag values, the "from-tag" attribute MUST
always contain a single tag value.

If the document contains no "char" or "range" elements with a
corresponding tag, the character class represents the empty set.
This is valid, to allow a common "rules" element to be shared across
files. However, it is RECOMMENDED that implementations allow for a
warning to ensure that referring to an undefined tag in this way is
intentional.

6.2.3. Unicode Property-based Classes

A class is defined in terms of Unicode properties by giving the
Unicode property alias and the property value or property value
alias, separated by a colon.

    <class name="virama" property="ccc:9" />

The example above selects all code points for which the Unicode
canonical combining class (ccc) value is 9. This value of the ccc is
assigned to all code points that encode viramas.

Unicode properties values MUST be designated via a composite of the
attribute name and value as defined for the property value in
[UAX42], separated by a colon. Loose matching of property values and
names as described in [UAX44] is not appropriate for an XML schema
and is not supported; it is likewise not supported in the XML
representation [UAX42] of the Unicode Character Database itself.

A property-based class MAY be anonymous, or, when defined as an
immediate child of the "rules" element, it MAY be named to relate a
formal property definition to its usage, such as the use of the value
9 for ccc to designate a virama (or halant) in various scripts.

Unicode properties may, in principle, change between versions of the
Unicode Standard. However, the values assigned for a given version
are fixed. If Unicode Properties are used, a Unicode version MUST be
declared in the "unicode-version" element in the header. (Note: some
Unicode properties are by definition stable across versions and do
not change once assigned (see [Unicode-Stability]).

All implementations processing LGR files SHOULD provide support for
the following minimal set of Unicode properties:

o General Category (gc)

o Script (sc)

o Canonical Combining Class (ccc)

o Bidi Class (bc)

o Arabic Joining Type (jt)

o Indic Syllabic Category (InSC)

o Deprecated (Dep)

The short name for each property is given in parentheses.

If a program that is using an LGR to determine the validity of a
label encounters a property that it does not support, it MUST abort
with an error.

6.2.4. Explicitly Declared Classes

A class of code points may also be declared by listing the code
points that are a member of the class. This is useful when tagging
cannot be used because code points are not listed individually as
part of the eligible set of code points for the given LGR, for
example because they only occur in code point sequences.

To define a class in terms of an explicit list of code points use a
space separated list of hexadecimal code point values:

    <class name="abcd">0061 0062 0063 0064</class>

This defines a class named "abcd" containing the code points for
characters "a", "b", "c" and "d". The ordering of the code points is
not material, but it is RECOMMENDED to list them in ascending order;
not doing so makes it unnecessarily difficult for users to detect
errors, such as duplicates or to compare and review these classes
against other specifications.

In a class definition, ranges of code points are represented by a
hexadecimal start and end value separated by a hyphen. The following
declaration is equivalent to the preceding:

    <class name="abcd">0061-0064</class>

Range and code point declarations can be freely intermixed:

    <class name="abcd">0061 0062-0063 0064</class>

The contents of a class differ from a repertoire in that the latter
MAY contain sequences as elements, while the former MUST NOT.
Instead, they closely resemble character classes as found in regular
expressions.

6.2.5. Combined Classes

Classes may be combined using operators for set complement, union,
intersection, difference (elements of the first class that are not in
the second) and symmetric difference (elements in either class, but
not both). Because classes fundamentally function like sets, the
union of several character classes is itself a class, for example.

Logical Operation	Example
Complement	`<complement><class by-ref="xxx"></complement>`

Union	`<union>`
	    `<class by-ref="class-1"/>`
	    `<class by-ref="class-2"/>`
	    `<class by-ref="class-3"/>`
	`</union>`

Intersection	`<intersection>`
	    `<class by-ref="class-1"/>`
	    `<class by-ref="class-2"/>`
	`</intersection>`

Difference	`<difference>`
	    `<class by-ref="class-1"/>`
	    `<class by-ref="class-2"/>`
	`</difference>`

Symmetric Difference	`<symmetric-difference>`
	    `<class by-ref="class-1"/>`
	    `<class by-ref="class-2"/>`
	`</symmetric-difference>`

Set Operators

The elements from this table may be arbitrarily nested inside each
other, subject to the following restriction: a "complement" element
MUST contain precisely one "class" or one of the operator elements,
while an "intersection", "symmetric-difference" or "difference"
element MUST contain precisely two, and a "union" element MUST
contain two or more of these elements.

An anonymous combined class can be defined directly inside a rule or
any of the match operator elements that allow child elements (see
Section 6.3.2) by using the set combination as the outer element.

```
<rule>
    <union>
        <class by-ref="xxx"/>
        <class by-ref="yyy"/>
    </union>
</rule>
```

The example shows the definition of an anonymous combined class that
represents the union of classes "xxx" and "yyy". There is no need to
wrap this union inside another "class" element, and, in fact, set
combination elements MUST NOT be nested inside a "class" element.

Lastly, to create a named combined class that can be referenced in
other classes or in rules as <class by-ref="xxxyyy"/>, add a "name"
attribute to the set combination element, for example <union
name="xxxyyy" /> and place it at the top level immediately below the
"rules" element (see Section 6.2.1).

```
 <rules>
    <union name="xxxyyy">
        <class by-ref="xxx"/>
        <class by-ref="yyy"/>
    </union>
      ...
  </rules>
```

Because (as for ordinary sets) a combination of classes is itself a
class, no matter by what combinations of set operators a combined
class is created, a reference to it always uses the "class" element
as described in Section 6.2.1. That is, a named class is always
referenced via an empty "class" element using the "by-ref" attribute
containing the name of the class to be referenced.

6.3. Whole Label and Context Rules

Each rule comprises a series of matching operators that must be
satisfied in order to determine whether a label meets a given
condition. Rules may reference other rules or character classes
defined elsewhere in the table.

6.3.1. The rule Element

   A matching rule is defined by a "rule" element, the child elements of
   which are one of the match operators from Section 6.3.2. In
   evaluating a rule, each child element is matched in order. Rule
   elements MAY be nested inside each other and inside certain match
   operators.

   A simple rule to match a label where all characters are members of
   some class called "preferred-codepoint":

```
<rule name="preferred-label">
    <start />
    <class by-ref="preferred-codepoint" count="1+"/>
    <end />
</rule>
```

   Rules are paired with explicit and implied actions, triggering these
   actions when a rule matches a label. For example, a simple explicit
   action for the rule shown above would be:

```
<action disp="allocatable" match="preferred-label" />
```

   The rule in this example would have the effect of setting the policy
   disposition for a label made up entirely of preferred code points to
   "allocatable". Explicit actions are further discussed in Section 7
   and implicit actions in Section 7.5. Another use of rules is in
   defining conditional contexts for code points and variants as
   discussed in Section 5.2 and Section 5.3.5.

   A rule that is an immediate child element of the "rules" element MUST
   be named using a "name" attribute containing a single identifier
   string with no spaces. A named rule may be incorporated into another
   rule by reference as well as referenced by an action, "when" or "not-
   when" attribute. If the "name" attribute is omitted, the rule is
   anonymous and MUST be nested inside another rule or match operator.

6.3.2. The Match Operators

   The child elements of a rule are a series of match operators, which
   are listed here by type and name and with a basic example or two.

```
+-----------+-----------+------------------------------------+
| Type      | Operator  | Examples                           |
+-----------+-----------+------------------------------------+
| logical   | any       | <any />                            |
|           +-----------+------------------------------------+
|           | choice    | <choice>                           |
|           |           |  <rule by-ref="alternative1"/>     |
|           |           |  <rule by-ref="alternative2"/>     |
|           |           | </choice>                          |
+-----------+-----------+------------------------------------+
| positional| start     | <start />                          |
|           +-----------+------------------------------------+
|           | end       | <end />                            |
+-----------+-----------+------------------------------------+
| literal   | char      | <char cp="0061 0062 0063" />       |
+-----------+-----------+------------------------------------+
| set       | class     | <class by-ref="class1" />          |
|           |           | <class>0061 0064-0065</class>      |
+-----------+-----------+------------------------------------+
| group     | rule      | <rule by-ref="rule1" />            |
|           |           | <rule><any /><rule />              |
+-----------+-----------+------------------------------------+
| contextual| anchor    | <anchor />                         |
|           +-----------+------------------------------------+
|           | look-ahead| <look-ahead><any /></look-ahead>   |
|           +-----------+------------------------------------+
|           | look-behind| <look-behind><any /></look-behind> |
+-----------+-----------+------------------------------------+
```

                          Match Operators

   Any element defining an anonymous class can be used as a match
   operator, including any of the set combination operators (see
   Section 6.2.5) as well as references to named classes.

   All match operators shown as empty elements in the Examples column of
   the table above do not support child elements of their own; otherwise
   match operators MAY be nested. In particular, anonymous "rule"
   elements can be used for grouping.

6.3.3. The count Attribute

   The OPTIONAL "count" attribute, when present, specifies the minimally
   required or maximal permitted number of times a match operator is
   used to match input. If the "count" attribute is

   n   the match operator matches the input exactly n times, where n is
         1 or greater.

   n+   the match operator matches the input at least n times, where n
         is 0 or greater.

   n:m   the match operator matches the input at least n times where n is
         0 or greater, but matches the input up to m times in total,
         where m > n. If m = n and n > 0, the match operator matches the
         input exactly n times.

   If there is no "count" attribute, the match operator matches the
   input exactly once.

   In matching, greedy evaluation is used in the sense defined for
   regular expressions: beyond the required number or times, the input
   is matched as many times as possible, but not so often as to prevent
   a match of the remainder of the rule.

   A "count" attribute MUST NOT be applied to any element that contains
   a "name" attribute, but MAY be applied to operators such as "class"
   that declare anonymous classes (including combined classes) or invoke
   any predefined classes by reference. The count attribute MUST NOT be
   applied to any "class" element, or element defining a combined class,
   when it is nested inside a combined class.

   A "count" attribute MUST NOT be applied to match operators of type
   "start", "end", "anchor", "look-ahead" or "look-behind" or to any
   operators, such as "rule" or "choice" that contain a nested instance
   of them. This limitation applies recursively, and irrespective of
   whether a rule element containing these nested instances is declared
   in place or used by reference.

   However, the "count" attribute MAY be applied to any other instances
   of either an anonymous "rule" element or of a "choice" element,
   including those instances nested inside other match operators. It
   MAY also be applied to the elements "any" and "char", when used as
   match operators.

6.3.4. The name and by-ref Attributes

   Like classes (see Section 6.2.1), rules declared as immediate child
   elements of the "rules" element MUST be named using a unique "name"
   attribute, and all other instances MUST NOT be named. Anonymous
   rules and classes or reference to named rules and classes can be
   nested inside other match operators by reference.

   To reference a named rule or class inside a rule or match operator
   use a "rule" or "class" element with an OPTIONAL "by-ref" attribute
   containing the name of the referenced element. It is an error to
   reference a rule or class for which the complete definition has not

been seen. In other words, it is explicitly not possible to define
recursive rules or class definitions. The "by-ref" attribute MUST
NOT appear in the same element as the "name" attribute, or in an
element that has any child elements.

The example shows several named classes and a named rule referencing
some of them by name.

```
<class name="letter" property="gc:L"/>
<class name="combining-mark" property="gc:M"/>
<class name="digit" property="gc:Nd" />
<rule name="letter-grapheme">
   <class by-ref="letter" count="1+"/>
   <class by-ref="combining-mark" count="0+"/>
</rule>
```

6.3.5. The choice Element

The "choice" element is used to represent a list of two or more
alternatives:

```
<rule name="ldh">
   <choice count="1+">
      <class by-ref="letter"/>
      <class by-ref="digit"/>
      <char cp="002D" comment="literal HYPHEN"/>
   </choice>
</rule>
```

Each child element of a "choice" represents one alternative. The
first matching alternative determines the match for the "choice"
element. To express a choice where an alternative itself consists of
a sequence of elements, the sequence must be wrapped in an anonymous
rule.

6.3.6. Literal Code Point Sequences

A literal code point sequence matches a single code point or a
sequence. It is defined by a "char" element, with the code point or
sequence to be matched given by the "cp" attribute. When used as a
literal, a "char" element MAY contain a "count" in addition to the
"cp" attribute and OPTIONAL "comment" or "ref" attributes. No other
attributes or child elements are permitted.

6.3.7. The any Element

   The "any" element is an empty element that matches any single code
   point. It MAY have a "count" attribute. For an example see
   Section 6.3.9.

   Unlike a literal, the "any" element MUST NOT have a "ref" attribute.

6.3.8. The start and end Elements

   To match the beginning or end of a label, use the "start" or "end"
   element. An empty label would match this rule:

       <rule name="empty-label">
           <start/>
           <end/>
       </rule>

   Conceptually, Whole Label Evaluation Rules evaluate the label as a
   whole, but in practice, many rules do not actually need to be
   specified to match the entire label. For example, to express a
   requirement of not starting a label with a digit, a rule needs to
   describe only the initial part of a label.

   This example uses the previously defined rules, together with start
   and end tag, to define a rule that requires that an entire label is
   well-formed. For this example that means, that it must start with a
   letter and contains no leading digits or combining marks, nor
   combining marks placed on digits.

       <rule name="leading-letter" >
         <start />
         <rule by-ref="letter-grapheme" count="1"/>
         <choice count="0+">
           <rule by-ref="letter-grapheme" count="0+"/>
           <class by-ref="digit" count="0+"/>
         </choice>
         <end />
       </rule>

   Each "start" or "end" element occurs at most once in a rule, except
   if nested inside a "choice" element in such a way that in matching
   each alternative at most one occurrence of each is encountered.
   Otherwise, the result is an error; as is any case where a "start" or
   "end" element is not encountered as first or last element to be
   matched, respectively, in matching a rule. Start and end elements
   are empty elements that do not have a "count" or any other attribute
   other than "comment". It is an error for any match operator

   enclosing a nested "start" or "end" element to have a "count"
   attribute.

6.3.9. Example context rule from IDNA specification

   This is an example of the whole label evaluation rule from [RFC5892]
   forbidding the mixture of the Arabic-Indic and extended Arabic-Indic
   digits in the same label. The example also demonstrates several
   instances of the use of anonymous rules for grouping.

```
    <data>
       <range first-cp="0660" last-cp="0669" not-when="mixed-digits"
             tag="arabic-indic-digits" />
       <range first-cp="06F0" last-cp="06F9" not-when="mixed-digits"
             tag="extended-arabic-indic-digits" />
    </data>
    <rules>
       <rule name="mixed-digits">
          <choice>
            <rule>
               <class from-tag="arabic-indic-digits"/>
               <any count="0+"/>
               <class from-tag="extended-arabic-indic-digits"/>
            </rule>
            <rule>
               <class from-tag="extended-arabic-indic-digits"/>
               <any count="0+"/>
               <class from-tag="arabic-indic-digits"/>
            </rule>
          </choice>
       </rule>
    </rules>
```

   As specified in the example, a label containing a code point from
   either of the two digit ranges is invalid for any label matching the
   "mixed-digits" rule, that is, anytime a code point from the other
   range is also present. Note that invalidating the label is not the
   same as invalidating the definition of the "range" elements; in
   particular, the definition of the tag values does not depend on the
   "when" attribute.

6.4. Parameterized Context or When Rules

   To recap: when a rule is intended to provide a context for evaluating
   the validity of a code point or variant mapping it is invoked by the
   "when" or "not-when" attributes described in Section 5.2. For "char"
   and "range" elements, an action implied by a context rule always has
   a disposition of "invalid" whenever the rule given by the "when"

attribute is not matched (see Section 7.5). Conversely, a "not-when" attribute results in a disposition of "invalid" whenever the rule is matched. When a rule is used in this way, it is called a context or "when" rule.

The example in the previous, section shows a whole label rule used as a context rule, essentially making the whole label the context. The next sections describe several match operators that can be used to provide a more specific specification of a context, allowing a parameterized context rule. See Section Section 7 for an alternative method of defining an invalid disposition for a label not matching a whole label rule.

6.4.1. The anchor Element

Such parameterized context rules are rules that contain a special place holder represented by an "anchor" element. As each When Rule is evaluated, if an "anchor" element if present, it is replaced by a literal corresponding to the "cp" attribute of the element containing the "when" (or "not-when") attribute. The match to the "anchor" element must be at the same position in the label as the code point or variant mapping triggering the When Rule.

For example, the Greek lower numeral sign is invalid if not immediately preceding a character in the Greek script. This is most naturally addressed with a parameterized When Rule using look-ahead:

```
<char cp="0375" when="preceding-greek"/>
...
<class name="greek-script" property="sc:Grek"/>
<rule name="preceding-greek">
    <anchor/>
    <look-ahead>
        <class by-ref="greek-script"/>
    </look-ahead>
</rule>
```

In evaluating this rule, the "anchor" element is treated as if it was replaced by a literal

```
<char cp="0375"/>
```

but only the instance of U+0375 at the given position is evaluated. If a label had two instances of U+0375 with the first one matching the rule and the second not, then evaluating the When Rule MUST succeed for the first and fail for the second instance.

Unlike other rules, rules containing an "anchor" element MUST only be invoked via the "when" or "not-when" attributes on code points or variants; otherwise their "anchor" elements cannot be evaluated. However, it is possible to invoke rules not containing an "anchor" element from a "when" or "not-when" attribute. (See Section 6.4.3)

The anchor element is an empty element, with no attributes permitted, except "comment".

6.4.2. The look-behind and look-ahead Elements

Context rules use the "look-behind" and "look-ahead" elements to define context before and after the code point sequence matched by the "anchor" element. If the "anchor" element is omitted, neither the "look-behind" nor the "look-ahead" element may be present in a rule.

Here is an example of a rule that defines an "initial" context for an Arabic code point:

```
<class name="transparent" property="jt:T"/>
<class name="right-joining" property="jt:R"/>
<class name="left-joining" property="jt:L"/>
<class name="dual-joining" property="jt:D"/>
<class name="non-joining" property="jt:U"/>
<rule name="Arabic-initial">
  <look-behind>
    <choice>
      <start/>
      <rule>
        <class by-ref="transparent" count="0+"/>
        <class by-ref="non-joining"/>
      </rule>
    </choice>
  </look-behind>
  <anchor/>
  <look-ahead>
    <class by-ref="transparent" count="0+" />
    <choice>
      <class by-ref="right-joining" />
      <class by-ref="dual-joining" />
    </choice>
  </look-ahead>
</rule>
```

A "when rule" (or context rule) is a named rule that contains any combination of "look-behind", "anchor" and "look-ahead" elements in that order. Each of these elements occurs at most once, except if

nested inside a "choice" element in such a way that in matching each
alternative at most one occurrence of each is encountered.
Otherwise, the result is undefined. None of these elements takes a
"count" attribute, nor does any enclosing match operator; otherwise,
the result is undefined. If a context rule contains a "look-ahead"
or "look-behind" element, it MUST contain an "anchor" element. If,
because of a choice element, a required anchor is not actually
encountered, the results are undefined.

6.4.3. Omitting the anchor Element

   If the "anchor" element is omitted, the evaluation of the context
   rule is not tied to the position of the code point or sequence
   associated with the "when" attribute.

   According to [RFC5892] Katakana middle dot is invalid in any label
   not containing at least one Japanese character anywhere in the label.
   Because this requirement is independent of the position of the middle
   dot, the rule does not require an "anchor" element.

```
    <char cp="30FB" when="japanese-in-label"/>
    <rule name="japanese-in-label">
        <union>
            <class property="sc:Hani"/>
            <class property="sc:Kata"/>
            <class property="sc:Hira"/>
        </union>
    </rule>
```

   The Katakana middle dot is used only with Han, Katakana or Hiragana.
   The corresponding When Rule requires that at least one code point in
   the label is in one of these scripts, but the position of that code
   point is independent of the location of the middle dot and no anchor
   therefore required. (Note that the Katakana middle dot itself is of
   script Common, that is, "sc:Zyyy").

7. The action Element

   The purpose of an action is to assign a disposition to a label in
   response to being triggered by the label meeting a specified
   condition. Often, the action simply results in blocking or
   invalidating a label that does not match a rule. An example of an
   action invalidating a label because it does not match a rule named
   "leading-letter" is as follows:

```
    <action disp="invalid" not-match="leading-letter"/>
```

If an action is to be triggered on matching a rule, a "match"
attribute is used instead. Actions are evaluated in the order that
they appear in the XML file. Once an action is triggered by a label,
the disposition defined in the "disp" attribute is assigned to the
label and no other actions are evaluated for that label.

The goal of the Label Generation Rules is to identify all labels and
variant labels and to assign them disposition values. These
dispositions are then fed into a further process that ultimately
implements all aspects of policy. To allow this specification to be
used with the widest range of policies, the permissible values for
the "disp" attribute are neither defined nor restricted.
Nevertheless a set of commonly used disposition values is
RECOMMENDED. (See Section 7.3)

7.1. The match and not-match Attributes

An OPTIONAL "match" or "not-match" attribute specifies a rule that
must be matched or not matched as a condition for triggering an
action. Only a single rule may be named as the value of a "match" or
"not-match" attribute. Because rules may be composed of other rules,
this restriction to a single attribute value does not impose any
limitation on the contexts that can trigger an action.

An action MUST NOT contain both a "match" and a "not-match"
attribute, and the value of either attribute MUST be the name of a
previously defined rule, otherwise the document MUST be rejected. An
action without any attributes is triggered by all labels
unconditionally. For a very simple LGR, the following action would
allocate all labels that match the repertoire:

    <action disp="allocatable" />

Since rules are evaluated for all labels, whether they are the
original label or computed by permuting the defined and valid variant
mappings for the label's code points, actions based on matching or
not matching a rule may be triggered for both original and variant
labels, but the rules are not affected by the disposition attributes
of the variant mappings. To trigger any actions based on these
dispositions requires the use of additional optional attributes for
actions described next.

7.2. Actions with Variant Type Triggers

7.2.1. The all-, any- and only-variants Attributes

   An action may contain one of the OPTIONAL attributes "any-variant",
   "all-variants", or "only-variants" defining triggers based on variant
   types. The permitted value for these attributes consists of one or
   more variant type values, separated by spaces. These MAY include
   type values that are not used in any "var" element in the LGR. When
   a variant label is generated, these variant type values are compared
   to the set of type values on the variant mappings used to generate
   the particular variant label (see Section 8).

   Any single match may trigger an action that contains an "any-variant"
   attribute, while for an "all-variants" or "only-variants" attribute,
   the variant type for all variant code points must match one or
   several of the type values specified in the attribute to trigger the
   action. There is no requirement that the entire list of variant type
   values be matched, as long as all variant code points match at least
   one of the values.

   An "only-variants" attribute will trigger the action only if all code
   points of the variant label have variant mappings from the original
   code points. In other words, the label contains no original code
   points other than those with a reflexive mapping (see Section 5.3.4).

```
    <char cp="0078" comment="x" />
       <var cp="0078" type="allocatable" comment="reflexive" />
       <var cp="0079" type="blocked" />
    </char>
    <char cp="0079"  comment="y"/>
       <var cp="0078" type="allocatable" />
    </char>
    ...
    <action disp="blocked" any-variant="blocked" />
    <action disp="allocatable" only-variants="allocatable" />
    <action disp="some-disp" any-variant="allocatable" />
```

   In the example above, the label "xx" would have variant labels "xx",
   "xy", "yx" and "yy". The first action would result in blocking any
   variant label containing "y", because the variant mapping from "x" to
   "y" is of type "blocked", triggering the "any-variant" condition.
   Because in this example "x" has a reflexive variant mapping to itself
   of type "allocatable" the original label "xx" has a reflexive variant
   "xx" that would trigger the "only-variants" condition on the second
   action.

   A label "yy" would have the variants "xy", "yx" and "xx". Because
   the variant mapping from "y" to "x" is of type "allocatable" and a
   mapping from "y" to "y" is not defined, the labels "xy" and "yx"

trigger the "any-variant" condition on the third label. The variant
"xx", being generated using the mapping from "y" to "x" of type
"allocatable", would trigger the "only-variants" condition on the
section action. As there is no reflexive variant "yy", the original
label "yy" cannot trigger any variant type triggers. However, it
could still trigger an action defined as matching or not matching a
rule.

In each action, one variant type trigger may be present by itself or
in conjunction with an attribute matching or not-matching a rule. If
variant triggers and rule-matching triggers are used together, the
label MUST "match" or respectively "not-match" the specified rule,
AND satisfy the conditions on the variant type values given by the
"any-variant", "all-variants", or "only-variants" attribute.

A useful convention combines the "any-variant" trigger with reflexive
variant mappings (Section 5.3.4). This convention is used, for
example, when multiple LGRs are defined within the same registry and
for overlapping repertoire. In some cases, the delegation of a label
from one LGR must prohibit the delegation of another label in some
other LGR. This can be done using a variant of type "blocked" as in
this example from an Armenian LGR, where the Armenian, Latin and
Cyrillic letters all look identical:

```
<char cp="0570" comment="Armenian small letter HO">
  <var cp="0068" type="blocked" comment="Latin small letter H" />
  <var cp="04BB" type="blocked"
      comment="Cyrillic small letter SSHA" />
</char>
```

The issue is that the target code points for these two variants are
both outside the Armenian repertoire. By using a reflexive variant
with the following convention:

```
<char cp="0068" comment="not part of repertoire">
  <var cp="0068" type="out-of-repertoire-var"
      comment="reflexive mapping" />
  <var cp="04BB" type="blocked"  />
  <var cp="0570" type="blocked"  />
</char>
  ...
```

and associating this with an action of the form:

```
<action disp="invalid" any-variant="out-of-repertoire-var" />
```

it is possible to list the symmetric and transitive variant mappings
in the LGR even where they involve out-of-repertoire code points. By

associating the action shown with the special type for these
reflexive mappings any original labels containing one or more of the
out-of-repertoire code points are filtered out -- just as if these
code points had not been listed in the LGR in the first place.
Nevertheless, they do participate in the permutation of variant
labels for n-repertoire labels (Armenian in the example), and these
permuted variants can be used to detect collisions with out-of-
repertoire labels (see Section 8).

7.2.2. Example from RFC 3743 Tables

This section gives an example of using variant type triggers,
combined with variants with reflexive mappings (Section 5.3.4) to
achieve LGRs that implement tables like those defined according to
[RFC3743] where the goal is to allow as variants only labels that
consist entirely of simplified or traditional variants, in addition
to the original label.

Assuming an LGR where all variants have been given suitable "type"
attributes of "blocked", "simplified", "traditional", or "both",
similar to the ones discussed in Appendix B. Given such an LGR, the
following example actions evaluate the disposition for the variant
label:

```
<action disp="blocked" any-variant="blocked" />
<action disp="allocatable" only-variants="simplified both" />
<action disp="allocatable" only-variants="traditional both" />
<action disp="blocked" all-variants="simplified traditional " />
<action disp="allocatable" />
```

The first action matches any variant label for which at least one of
the code point variants is of type "blocked". The second matches any
variant label for which all of the code point variants are of type
"simplified" or "both", in other words an all-simplified label. The
third matches any label for which all variants are of type
"traditional" or "both", that is all traditional. These two actions
are not triggered by any variant labels containing some original code
points, unless each of those code points has a variant defined with a
reflexive mapping (Section 5.3.4).

The final two actions rely on the fact that actions are evaluated in
sequence, and that the first action triggered also defines the final
disposition for a variant label (see Section 7.4). They further rely
on the assumption that the only variants with type "both" are also
reflexive variants.

Given these assumptions, any remaining simplified or traditional
variants must then be part of a mixed label, and so are blocked; all

labels surviving to the last action are original code points only
(that is the original label). The example assumes that an original
label may be a mixed label; if that is not the case, the disposition
for the last action would be set to "blocked".

There are exceptions where the assumption on reflexive mappings made
above does not hold, so this basic scheme needs some refinements to
cover all cases. For a more complete example, see Appendix B.

7.3. Recommended Disposition Values

The precise nature of the policy action taken in response to a
disposition and the name of the corresponding "disp" attributes are
only partially defined here. It is strongly RECOMMENDED to use the
following dispositions only with their conventional sense.

   invalid The resulting string is not a valid label. This disposition
        may be assigned implicitly, see Section 7.5. No variant labels
        should be generated from a variant mapping with this type.

   blocked The resulting string is a valid label, but should be blocked
        from registration. This would typically apply for a derived
        variant that is undesirable due to having no practical use or
        being confusingly similar to some other label.

   allocatable The resulting string should be reserved for use by the
        same operator of the origin string, but not automatically
        allocated for use.

   activated The resulting string should be activated for use. (This
        is the same as a preferred variant in [RFC3743].)

   valid The resultant string is a valid label. (This is the typical
        default action if no dispositions are defined.)

7.4. Precedence

Actions are applied in the order of their appearance in the file.
This defines their relative precedence. The first action triggered
by a label defines the disposition for that label. To define the
order of precedence, list the actions in the desired order. The
conventional order of precedence for the actions defined in
Section 7.3 is "invalid", "blocked", "allocatable", "activated" then
"valid". This default precedence is used for the default actions
defined in Section 7.6.

7.5. Implied Actions

   The context rules on code points ("not-when" or "when" rules) carry
   an implied action with a disposition of "invalid" (not eligible) if a
   "when" context is not satisfied, or respectively a "not-when" context
   is matched. These rules are evaluated at the time the code points
   for a label or its variant labels are checked for validity (see
   Section 8). In other words, they are evaluated before any of the
   whole-label evaluation rules and with higher precedence. The context
   rules for variant mappings are evaluated when variants are generated
   and/or when variant tables are made symmetric and transitive. They
   have an implied action with a disposition of "invalid" which means a
   putative variant mapping does not exist whenever the given context
   matches a "not-when" rule or fails to match a "when" rule specified
   for that mapping. The result of that disposition is that the variant
   mapping is ignored in generating variant labels and the value is
   therefore not accessible to trigger any explicit actions.

   Note that such non-existing variant mapping is different from a
   blocked variant, which is a variant code point mapping that exists
   but results in a label that may not be allocated.

7.6. Default Actions

   If a label does not trigger any of the actions defined explicitly in
   the LGR, the following implicitly defined default actions are
   evaluated. They are shown below in their relative order of
   precedence (see Section 7.4). Default actions have a lower order of
   precedence than explicit actions (see Section 8.3).

   The default actions for variant labels are defined as follows. The
   first set is triggered based on the standard variant type values of
   "invalid", "blocked", "allocatable" and "activated":

       <action disp="invalid" any-variant="invalid"/>
       <action disp="blocked" any-variant="blocked"/>
       <action disp="allocatable" any-variant="allocatable"/>
       <action disp="activated" all-variants="activated"/>

   A final default action sets the disposition to "valid" for any label
   matching the repertoire for which no other action has been triggered.
   This "catch-all" action also matches all remaining variant labels
   from variants that do not have a type value.

       <action disp="valid" comment="Catch-all if other rules not met"/>

   Conceptually, the implicitly defined default actions act just like a
   block of action elements that is added (virtually) beyond the last of

the user-supplied actions. Any label not processed by the user-
supplied actions would thus be processed by the default actions as if
they were present in the LGR. As the last default action is a
"catch-all", all processing is guaranteed to end with a definite
disposition for the label.

8. Processing a Label against an LGR

8.1. Determining Eligibility for a Label

   In order to test a given label for membership in the LGR, a consumer
   of the LGR must iterate through each code point within a given label,
   and test that each instance of a code point is a member of the LGR.
   If any instance of a code point is not a member of the LGR, the label
   shall be deemed as invalid.

   An individual instance of a code point is deemed a member of the LGR
   when it is listed using a "char" element, or is part of a range
   defined with a "range" element, and all necessary conditions in any
   "when" or "not-when" attributes are correctly satisfied for that
   instance.

   Alternatively, an instance of a code point is also deemed a member of
   the LGR when it forms part of a sequence that corresponds to a
   sequence listed using a "char" element for which the "cp" attribute
   defines a sequence, and all necessary conditions in any "when" or
   "not-when" attributes are correctly satisfied for that instance of
   the sequence.

   In determining eligibility, at each position the longest possible
   sequence of code points is evaluated first. If that sequence matches
   a sequence defined in the LGR and satisfies any required context at
   that position, the instances of its constituent code points are
   deemed members of the LGR and evaluation proceeds with the next code
   point following the sequence. If the sequence does not match a
   defined sequence or does not satisfy the required context,
   successively shorter sequences are evaluated, until only a single
   code point remains. The eligibility of that code point is determined
   as described above for an individual code point instance.

   A label must also not trigger any action that results in a
   disposition of "invalid", otherwise it is deemed not eligible. (This
   step may need to be deferred, until variant code point dispositions
   have been determined).

8.1.1. Determining Eligibility using Reflexive Variant Mappings

   For LGRs that contain reflexive variant mappings (defined in
   Section 5.3.4), the final evaluation of eligibility for the label
   must be deferred until variants are generated. In essence, LGRs that
   use this feature treat the original label as the (identity) variant
   of itself. For such LGRs, the ordinary determination of eligibility
   described here is but a first step that generally excludes only a
   subset of invalid labels.

   To further check the validity of a label with reflexive mappings, it
   is not necessary to generate all variant labels. Only a single
   variant needs to be created, where any reflexive variants are applied
   for each code point, and the label disposition is evaluated (as
   described in Section 8.3). A disposition of "invalid" results in the
   label being not eligible. (In the exceptional case where context
   rules are present on reflexive mappings, multiple reflexive variants
   may be defined, but for each original label, at most one of these can
   be valid at each code position. However, see Section 8.4).

8.2. Determining Variants for a Label

   For a given eligible label, the set of variant labels is deemed to
   consist of each possible permutation of original code points and
   substituted code points or sequences defined in "var" elements,
   whereby all "when" and "not-when" attributes are correctly satisfied
   for each "char" or "var" element in the given permutation and all
   applicable whole label evaluation rules are satisfied as follows:

   1. Create each possible permutation of a label, by substituting each
       code point or code point sequence in turn by any defined variant
       mapping (including any reflexive mappings)

   2. Apply variant mappings with "when" or "not-when" attributes only
       if the conditions are satisfied; otherwise they are not defined

   3. Record each of the "type" values on the variant mappings used in
       creating a given variant label in a disposition set; for any
       unmapped code point record the "type" value of any reflexive
       variant (see Section 5.3.4)

   4. Determine the disposition for each variant label per Section 8.3

   5. If the disposition is "invalid", remove the label from the set

   6. If final evaluation of the disposition for the unpermuted label
       per Section 8.3 results in a disposition of "invalid", remove all
       associated variant labels from the set.

The number of potential permutations can be very large. In practice,
implementations would use suitable optimizations to avoid having to
actually create all permutations (see Section 8.5).

In determining the permuted set of variant labels in step (1) above,
all eligible partitions into sequences must be evaluated. A label
"ab" that matches a sequence "ab" defined in the LGR but also matches
the sequence of individual code points "a" and "b" (both defined in
the LGR), must be permuted using any defined variant mappings for
both the sequence "ab" and the code points "a" and "b" individually.

8.3. Determining a Disposition for a Label or Variant Label

For a given label (variant or original), its disposition is
determined by evaluating in order of their appearance all actions for
which the label or variant label satisfies the conditions.

1. For any label that contains code points or sequences not defined
    in the repertoire, or does not satisfy the context rules on all
    of its code points and variants, the disposition is "invalid".

2. For all other labels the disposition is given by the value of the
    "disp" attribute for the first action triggered by the label. An
    action is triggered, if all of the following are true:

    * the label matches the whole label evaluation rule given in the
       "match" attribute for that action;

    * the label does not match the whole label evaluation rule given
       in the "not-match" attribute for that action;

    * any of the recorded variant types for a variant label match
       the types given in the "any-variant" attribute for that
       action;

    * all of the recorded variant types for a variant label match
       the types given in the "all-variants" or "only-variants"
       attribute given for that action;

    * in case of an "only-variants" attribute, the label contains
       only code points that are the target of applied variant
       mappings;

    or

    * the action does not contain any "match", "not-match", "any-
       variant", "all-variants", or "only-variants" attributes:
       catch-all.

3. For any remaining variant label, assign the variant label the
    disposition using the default actions defined in Section 7.6.
    For this step, variant types outside the predefined recommended
    set (see Section 7.3) are ignored.

4. For any remaining label, set the disposition to "valid".

8.4. Duplicate Variant Labels

For a poorly designed LGR, it is possible to generate duplicate
variant labels from the same input label, but with different, and
potentially conflicting dispositions. Implementations MUST treat any
duplicate variant labels encountered as an error, irrespective of
their dispositions.

This situation can arise in two ways. One is described in
Section 5.3.5 and involves defining the same variant mapping with two
context rules that are formally distinct, but nevertheless overlap so
that they are not mutually exclusive for the same label.

The other case involves variants defined for sequences, where one
sequence is a prefix of another (see Section 5.3.1). The following
shows such an example resulting in conflicting reflexive variants:

```
<char cp="0061">
  <var cp="0061" type="allocatable"/>
</char>
<char cp="0062"/>
<char cp="0061 0062">
  <var cp="0061 0062" type="blocked"/>
</char>
```

A label "ab" would generate the variant labels "{a}{b}" and "{ab}"
where the curly braces show the sequence boundaries as they were
applied during variant mapping. The result is a duplicate variant
label "ab", one based on a variant of type "allocatable" plus an
original code point "b" that has no variant, and another one based on
a single variant of type "blocked", thus creating two variant labels
with conflicting dispositions.

In the general case it is difficult to impossible to prove by
mechanical inspection of the LGR that duplicate variant labels will
never occur, so implementations have to be prepared to detect this
error during variant label generation. The condition is easily
avoided by careful design of context rules and special attention to
the relation among code point sequences with variants.

8.5. Checking Labels for Collision

   The obvious method for checking collision between labels is to
   generate the fully permuted set of variants for one of them and see
   whether it contains the other label as a member. As discussed above,
   this can be prohibitive and is not necessary.

   Because of symmetry and transitivity, all variant mappings form
   disjoint sets. In each of these sets, the source and target of each
   mapping are also variants of the sources and targets of all the other
   mappings. However, members of two different sets are never variants
   of each other.

   If two labels have code points at the same position that are members
   of two different of these variant mapping sets, any variant labels of
   one, cannot be variant labels of the other: the sets of their variant
   labels are likewise disjoint. Instead of generating all permutations
   to compare all possible variants, it is enough to find out whether
   code points at the same position belong to the same variant set or
   not.

   For that, it is sufficient to substitute an "index" mapping that
   identifies the set. This index mapping could be, for example, the
   variant mapping for which the target code point (or sequence) comes
   first in some sorting order. This index mapping would, in effect,
   identify the set of variant mappings for that position.

   To check collision then means generating a single variant label from
   the original by substituting the respective "index" value for each
   code point. This results in an "index label". Two labels collide
   whenever the index labels for them are the same.

9. Conversion to and from Other Formats

   Both [RFC3743] and [RFC4290] provide different grammars for IDN
   tables. The formats in those documents are unable to fully support
   the increased requirements of contemporary IDN variant policies.

   This specification is a superset of functionality provided by the
   older IDN table formats, thus any table expressed in those formats
   can be expressed in this new format. Automated conversion can be
   conducted between tables conformant with the grammar specified in
   each document.

   For notes on how to translate an RFC 3743-style table, see
   Appendix B.

10. Media Type

   Well-formed LGRs that comply with this specification SHOULD be
   transmitted with a media type of "application/lgr+xml". This media
   type will signal to an LGR-aware client that the content is designed
   to be interpreted as an LGR.

11. IANA Considerations

   This document requests the following actions from IANA:

11.1. Media Type Registration

   The media type "application/lgr+xml" should be registered to denote
   transmission of label generation rulesets that are compliant with
   this specification, in accordance with [RFC6838].

   Type name: application

   Subtype name: lgr+xml

   Required parameters: N/A

   Optional parameters: charset (as for application/xml per [RFC7303])

   Security considerations: See the security considerations for
   application/xml in [RFC7303] and the specific security considerations
   for Label Generation Rulesets in RFC XXXX (RFC Editor/IANA, please
   replace XXXX with the final number of this document)

   Interoperability considerations: As for application/xml per [RFC7303]

   Published specification: See RFC XXXX (RFC Editor/IANA, please
   replace XXXX with the final number of this document)

   Applications which use this media type: Software using label
   generation rulesets for international identifiers, such as IDNs,
   including registry applications and client validators.

   Additional information:

      Deprecated alias names for this type: N/A

      Magic number(s): N/A

      File extension(s): .lgr

      Macintosh file type code(s): N/A

Person & email address for further information:

    Kim Davies <kim.davies@icann.org>

    Asmus Freytag <asmus@unicode.org>

Intended Usage: COMMON

Restrictions on usage: N/A

Author:

    Kim Davies <kim.davies@icann.org>

    Asmus Freytag <asmus@unicode.org>

Change Controller: IESG

Provisional registration? (standards tree only): No

11.2. URN Registration

This specification uses a URN to describe the XML namespace, in
accordance with [RFC3688].

URI: urn:ietf:params:xml:ns:lgr-1.0

Registrant Contact: See the Authors of this document.

XML: None.

11.3. Disposition Registry

This document establishes a vocabulary of "Label Generation Ruleset
Dispositions" which should be reflected as a new IANA registry. This
registry should be divided into two sub-registries:

o Standard Dispositions - This registry shall list dispositions that
   have been defined in published specifications, i.e. the
   eligibility for such registrations shall be "Specification
   Required" [RFC5226]. The initial set of registrations shall be
   the five dispositions in this document described in Section 7.3.

o Private Dispositions - This registry shall list dispositions that
   have been registered "First Come First Served" [RFC5226] by third
   parties with the IANA. Such dispositions must take the form
   "entity:disposition" where the entity is a domain name that
   uniquely identifies the private user of the namespace. For

example, "example.org:reserved" could be a private extension used
by the example organization to denote a disposition relating to
reserved labels. These extensions are not intended to be
interoperable, but registration is designed to minimize potential
conflicts. It is strongly recommended any new dispositions that
require interoperability and have applicability beyond a single
organization be defined as Standard Dispositions.

Standard dispositions MUST NOT contain the ":" character in order to
distinguish them from private dispositions. All disposition names
shall be in lowercase ASCII.

The IANA registry should provide data on the name of the disposition,
the intended purposes, and the registrant or defining specification
for the disposition.

12. Security Considerations

12.1. LGRs Are Only a Partial Remedy for Problem Space

Substantially unrestricted use of non-ASCII characters in security-
relevant identifiers such as domain name labels may cause user
confusion and invite various types of attacks. In many languages, in
particular those using complex or large scripts, an attacker has an
opportunity to divert or confuse users as a result of different code
points with identical appearance or similar semantics.

Label Generation Rules provide a partial remedy for these risks by
supplying a framework for prohibiting inappropriate code points or
sequences from being registered at all and for permitting "variant"
code points to be grouped together so that labels containing them may
be mutually exclusive or registered only to the same owner.

In addition, by being fully machine processable the format may enable
automated checks for known weaknesses in label generation rules.
However, by itself, the format or this specification do not ensure
that the label generation rules expressed in this format are free of
risk. Additional approaches may be considered depending on the
acceptable trade-off between flexibility and risk for a given
application. One method of managing risk may involve a case-by-case
evaluation of a proposed label in context with already registered
labels, for example, when reviewing labels for their degree of visual
confusability.

12.2. Computational Expense of Complex Tables

   A naive implementation attempting to generate all variant labels for
   a given label could lead to the possibility of exhausting the
   resources on the machine running the LGR processor, potentially
   causing denial-of-service consequences. For many operations, brute
   force generation can be avoided by optimization, and if needed, the
   number of permuted labels can be estimated more cheaply ahead of
   time.

   The implementation of Whole Label Evaluation rules, using certain
   backtracking algorithms, can take exponential time for pathological
   rules or labels and exhaust stack resources. This can be mitigated
   by proper implementation and enforcing the restrictions on
   permissible label length.

13. References

13.1. Normative References

   [RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail
              Extensions (MIME) Part One: Format of Internet Message
              Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996,
              <http://www.rfc-editor.org/info/rfc2045>.

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/
              RFC2119, March 1997,
              <http://www.rfc-editor.org/info/rfc2119>.

   [RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet:
              Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002,
              <http://www.rfc-editor.org/info/rfc3339>.

   [RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying
              Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646,
              September 2009, <http://www.rfc-editor.org/info/rfc5646>.

   [UAX42]   Unicode Consortium, "Unicode Character Database in XML",
              <http://unicode.org/reports/tr42/>.

   [Unicode-Stability]
              Unicode Consortium, "Unicode Encoding Stability Policy,
              Property Value Stability",
              <http://www.unicode.org/policies/
              stability_policy.html#Property_Value>.

   [Unicode-Versions]
            Unicode Consortium, "Unicode Version Numbering",
            <http://unicode.org/versions/#Version_Numbering>.

   [XML]   World Wide Web Consortium, "Extensible Markup Language
            (XML) 1.0", <http://www.w3.org/TR/REC-xml/>.

13.2. Informative References

   [ASIA-TABLE]
            DotAsia Organisation, ".ASIA ZH IDN Language Table".

   [LGR-PROCEDURE]
            Internet Corporation for Assigned Names and Numbers,
            "Procedure to Develop and Maintain the Label Generation
            Rules for the Root Zone in Respect of IDNA Labels",
            <http://www.icann.org/en/resources/idn/
            draft-lgr-procedure-07dec12-en.pdf>.

   [RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688,
            DOI 10.17487/RFC3688, January 2004,
            <http://www.rfc-editor.org/info/rfc3688>.

   [RFC3743] Konishi, K., Huang, K., Qian, H., and Y. Ko, "Joint
            Engineering Team (JET) Guidelines for Internationalized
            Domain Names (IDN) Registration and Administration for
            Chinese, Japanese, and Korean", RFC 3743, DOI 10.17487/
            RFC3743, April 2004,
            <http://www.rfc-editor.org/info/rfc3743>.

   [RFC4290] Klensin, J., "Suggested Practices for Registration of
            Internationalized Domain Names (IDN)", RFC 4290, DOI
            10.17487/RFC4290, December 2005,
            <http://www.rfc-editor.org/info/rfc4290>.

   [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an
            IANA Considerations Section in RFCs", BCP 26, RFC 5226,
            DOI 10.17487/RFC5226, May 2008,
            <http://www.rfc-editor.org/info/rfc5226>.

   [RFC5564] El-Sherbiny, A., Farah, M., Oueichek, I., and A. Al-Zoman,
            "Linguistic Guidelines for the Use of the Arabic Language
            in Internet Domains", RFC 5564, DOI 10.17487/RFC5564,
            February 2010, <http://www.rfc-editor.org/info/rfc5564>.

   [RFC5891] Klensin, J., "Internationalized Domain Names in
              Applications (IDNA): Protocol", RFC 5891, DOI 10.17487/
              RFC5891, August 2010,
              <http://www.rfc-editor.org/info/rfc5891>.

   [RFC5892] Faltstrom, P., Ed., "The Unicode Code Points and
              Internationalized Domain Names for Applications (IDNA)",
              RFC 5892, DOI 10.17487/RFC5892, August 2010,
              <http://www.rfc-editor.org/info/rfc5892>.

   [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type
              Specifications and Registration Procedures", BCP 13, RFC
              6838, DOI 10.17487/RFC6838, January 2013,
              <http://www.rfc-editor.org/info/rfc6838>.

   [RFC7303] Thompson, H. and C. Lilley, "XML Media Types", RFC 7303,
              DOI 10.17487/RFC7303, July 2014,
              <http://www.rfc-editor.org/info/rfc7303>.

   [TDIL-HINDI]
              Technology Development for Indian Languages (TDIL)
              Programme, "Devanagari Script Behaviour for Hindi".

   [UAX44]   Unicode Consortium, "Unicode Character Database",
              <http://unicode.org/reports/tr44/>.

   [WLE-RULES]
              Internet Corporation for Assigned Names and Numbers, "WLE
              Rules", <https://community.icann.org/download/
              attachments/43989034/WLE-Rules.pdf>.

Appendix A. Example Tables

   The following presents a minimal LGR table defining the lower case
   LDH (letter-digit-hyphen) repertoire and containing no rules or
   metadata elements. Many simple LGR tables will look quite similar,
   except that they would contain some metadata.

```
   <?xml version="1.0" encoding="utf-8"?>
   <lgr xmlns="urn:ietf:params:xml:ns:lgr-1.0">
   <data>
       <char cp="002D" comment="HYPHEN (-)" />
       <range first-cp="0030" last-cp="0039"
         comment="DIGIT ZERO - DIGIT NINE" />
       <range first-cp="0061" last-cp="007A"
         comment="LATIN SMALL LETTER A - LATIN SMALL LETTER Z" />
   </data>
   </lgr>
```

In practice, any LGR that includes the hyphen might also contain
rules invalidating any labels beginning, ending, and containing a
hyphen in the third and fourth positions as required by [RFC5891].

```xml
<?xml version="1.0" encoding="utf-8"?>
<lgr xmlns="urn:ietf:params:xml:ns:lgr-1.0">
<data>
    <char cp="002D"
          not-when="hyphen-minus-disallowed" />
    <range first-cp="0030" last-cp="0039" />
    <range first-cp="0061" last-cp="007A" />
</data>
<rules>
    <rule name="hyphen-minus-disallowed"
          comment="RFC5891 restrictions on U+002D">
      <choice>
        <rule comment="no leading hyphen">
          <look-behind>
            <start />
          </look-behind>
          <anchor />
        </rule>
        <rule comment="no trailing hyphen">
          <anchor />
          <look-ahead>
            <end />
          </look-ahead>
        </rule>
        <rule comment="no consecutive hyphens
                in third and fourth positions">
          <look-behind>
            <start />
            <any />
            <any />
            <char cp="002D" comment="hyphen-minus" />
          </look-behind>
          <anchor />
        </rule>
      </choice>
    </rule>
</rules>
</lgr>
```

The following sample LGR shows a more complete collection of the
elements and attributes defined in this specification in a somewhat
typical context.

```xml
<?xml version="1.0" encoding="utf-8"?>
```

```
    <!-- This example uses a large subset of the features of this
         specification. It does not include every set operator,
         match operator element, or action trigger attribute, their
         use being largely parallel to the ones demonstrated. -->

    <lgr xmlns="urn:ietf:params:xml:ns:lgr-1.0">
    <!-- meta element with all optional elements -->
      <meta>
        <version comment="initial version">1</version>
        <date>2010-01-01</date>
        <language>sv</language>
        <scope type="domain">example.com</scope>
        <validity-start>2010-01-01</validity-start>
        <validity-end>2013-12-31</validity-end>
        <description type="text/html">
            <![CDATA[
            This language table was developed with the
            <a href="http://swedish.example/">Swedish
            examples institute</a>.
            ]]>
        </description>
        <unicode-version>6.3.0</unicode-version>
        <references>
          <reference id="0" comment="the most recent" >The
              Unicode Standard 6.2</reference>
          <reference id="1" >RFC 5892</reference>
          <reference id="2" >Big-5: Computer Chinese Glyph
             and Character Code Mapping Table, Technical Report
             C-26, 1984</reference>
        </references>
      </meta>
      <!-- the data section describing the repertoire -->
       <data>
         <!-- single code point "char" element -->
         <char cp="002D" ref="1" comment="HYPHEN" />

         <!-- range elements for contiguous code points,  with tags -->
         <range first-cp="0030" last-cp="0039" ref="1" tag="digit" />
         <range first-cp="0061" last-cp="007A" ref ="1" tag="letter" />

         <!-- code point sequence -->
         <char cp="006C 00B7 006C" comment="Catalan middle dot" />

         <!-- alternatively use a when rule -->
         <char cp="00B7" when="catalan-middle-dot" />

          <!-- code point with context rule -->
         <char cp="200D" when="joiner" ref="2" />
```

```
      <!-- code points with variants -->
      <char cp="4E16" tag="preferred" ref="0">
        <var cp="4E17" type="blocked" ref="2" />
        <var cp="534B" type="allocatable" ref="2" />
      </char>
      <char cp="4E17" ref="0">
        <var cp="4E16" type="allocatable" ref="2" />
        <var cp="534B" type="allocatable" ref="2" />
      </char>
      <char cp="534B" ref="0">
        <var cp="4E16" type="allocatable" ref="2" />
        <var cp="4E17" type="blocked" ref="2" />
      </char>
    </data>

    <!-- Context and whole label rules -->
    <rules>
      <!-- Require the given code point to be between two 006C -->
      <rule name="catalan-middle-dot" ref="0">
          <look-behind>
              <char cp="006C" />
          </look-behind>
          <anchor />
          <look-ahead>
              <char cp="006C" />
          </look-ahead>
      </rule>

      <!-- example of a context rule based on property -->
      <class name="virama" property="ccc:9" />
      <rule name="joiner"  ref="1" >
          <look-behind>
              <class by-ref="virama" />
          </look-behind>
          <anchor />
      </rule>

      <!-- example of using set operators -->

      <!-- Subtract vowels from letters to get
           consonant, demonstrating the different
           set notations and the difference operator -->
       <difference name="consonants">
          <class comment="all letters">0061-007A</class>
          <class comment="all vowels">
                  0061 0065 0069 006F 0075
          </class>
       </difference>
```

```
    <!-- by using the start and end, rule matches whole label -->
     <rule name="three-or-more-consonants">
        <start />
        <!-- reference the class defined by the difference
             and require three or more matches -->
        <class by-ref="consonants" count="3+" />
        <end />
     </rule>

     <!-- rule for negative matching -->
     <rule name="non-preferred"
          comment="matches any non-preferred code point">
        <complement comment="non-preferred" >
           <class from-tag="preferred" />
        </complement>
     </rule>

   <!-- actions triggered by matching rules and/or
        variant types -->
     <action disp="invalid"
            match="three-or-more-consonants" />
     <action disp="blocked" any-variant="blocked" />
     <action disp="allocatable" all-variants="allocatable"
            not-match="non-preferred" />
   </rules>
  </lgr>
```

Appendix B. How to Translate RFC 3743 based Tables into the XML Format

   As a background, the [RFC3743] rules work as follows:

   1. The Original (requested) label is checked to make sure that all
       the code points are a subset of the repertoire.

   2. If it passes the check, the Original label is allocatable.

   3. Generate the all-simplified and all-traditional variant labels
       (union of all the labels generated using all the simplified
       variants of the code points) for allocation.

   To illustrate by example, here is one of the more complicated set of
   variants:

```
        U+4E7E
        U+4E81
        U+5E72
        U+5E79
        U+69A6
        U+6F27
```

   The following shows the relevant section of the Chinese language
   table published by the .ASIA registry [ASIA-TABLE]. Its entries
   read:

    <codepoint>;<simpl-variant(s)>;<trad-variant(s)>;<other-variant(s)>

   These are the lines corresponding to the set of variants listed above

```
   U+4E7E;U+4E7E,U+5E72;U+4E7E;U+4E81,U+5E72,U+6F27,U+5E79,U+69A6
   U+4E81;U+5E72;U+4E7E;U+5E72,U+6F27,U+5E79,U+69A6
   U+5E72;U+5E72;U+5E72,U+4E7E,U+5E79;U+4E7E,U+4E81,U+69A6,U+6F27
   U+5E79;U+5E72;U+5E79;U+69A6,U+4E7E,U+4E81,U+6F27
   U+69A6;U+5E72;U+69A6;U+5E79,U+4E7E,U+4E81,U+6F27
   U+6F27;U+4E7E;U+6F27;U+4E81,U+5E72,U+5E79,U+69A6
```

   The corresponding data section XML format would look like this:

```
    <data>
    <char cp="4E7E">
    <var cp="4E7E" type="both" comment="identity" />
    <var cp="4E81" type="blocked" />
    <var cp="5E72" type="simp" />
    <var cp="5E79" type="blocked" />
    <var cp="69A6" type="blocked" />
    <var cp="6F27" type="blocked" />
    </char>
    <char cp="4E81">
    <var cp="4E7E" type="trad" />
    <var cp="5E72" type="simp" />
    <var cp="5E79" type="blocked" />
    <var cp="69A6" type="blocked" />
    <var cp="6F27" type="blocked" />
    </char>
    <char cp="5E72">
    <var cp="4E7E" type="trad"/>
    <var cp="4E81" type="blocked"/>
    <var cp="5E72" type="both" comment="identity"/>
    <var cp="5E79" type="trad"/>
    <var cp="69A6" type="blocked"/>
    <var cp="6F27" type="blocked"/>
    </char>
```

```
        <char cp="5E79">
        <var cp="4E7E" type="blocked"/>
        <var cp="4E81" type="blocked"/>
        <var cp="5E72" type="simp"/>
        <var cp="5E79" type="trad" comment="identity"/>
        <var cp="69A6" type="blocked"/>
        <var cp="6F27" type="blocked"/>
        </char>
        <char cp="69A6">
        <var cp="4E7E" type="blocked"/>
        <var cp="4E81" type="blocked"/>
        <var cp="5E72" type="simp"/>
        <var cp="5E79" type="blocked"/>
        <var cp="69A6" type="trad" comment="identity"/>
        <var cp="6F27" type="blocked"/>
        </char>
        <char cp="6F27">
        <var cp="4E7E" type="simp"/>
        <var cp="4E81" type="blocked"/>
        <var cp="5E72" type="blocked"/>
        <var cp="5E79" type="blocked"/>
        <var cp="69A6" type="blocked"/>
        <var cp="6F27" type="trad" comment="identity"/>
        </char>
      </data>
```

Here the simplified variants have been given a type of "simp", the
traditional variants one of "trad" and all other ones are given
"blocked".

Because some variant mappings show in more than one column, while the
XML format allows only a single type value, they have been given the
type of "both".

Note that some variant mappings map to themselves (identity), that is
the mapping is reflexive (see Section 5.3.4). In creating the
permutation of all variant labels, these mappings have no effect,
other than adding a value to the variant type list for the variant
label containing them.

In the example so far, all of the entries with type="both" are also
mappings where source and target are identical. That is, they are
reflexive mappings as defined in Section 5.3.4.

Given a label "U+4E7E U+4E81", the following labels would be ruled
allocatable under [RFC3743] based on how that standard is commonly
implemented in domain registries:

```
      Original label:     U+4E7E U+4E81
      Simplified label 1: U+4E7E U+5E72
      Simplified label 2: U+5E72 U+5E72
      Traditional label:  U+4E7E U+4E7E
```

However, if allocatable labels were generated simply by a straight
permutation of all variants with type other than type="blocked" and
without regard to the simplified and traditional variants, we would
end up with an extra allocatable label of "U+5E72 U+4E7E". This
label is comprises both a Simplified Chinese character and a
Traditional Chinese code point and therefore shouldn't be
allocatable.

To more fully resolve the dispositions requires several actions to be
defined as described in Section 7.2.2 which will override the default
actions from Section 7.6. After blocking all labels that contain a
variant with type "blocked", these actions will set to allocatable
labels based on the following variant types: "simp", "trad" and
"both". Note that these variant types do not directly relate to
dispositions for the variant label, but that the actions will resolve
them to the standard dispositions on labels, to with "blocked" and
"allocatable".

To resolve label dispositions requires five actions to be defined (in
the rules section of this document) these actions apply in order and
the first one triggered, defines the disposition for the label. The
actions are:

1. block all variant labels containing at least one blocked variant.

2. allocate all labels that consist entirely of variants that are
    "simp" or "both"

3. also allocate all labels that are entirely "trad" or "both"

4. block all surviving labels containing any one of the dispositions
    "simp" or "trad" or "both" because they are now known to be part
    of an undesirable mixed simplified/traditional label

5. allocate any remaining label; the original label would be such a
    label.

The rules declarations would be represented as:

```
<rules>
  <!--Action elements - order defines precedence-->
  <action disp="blocked"     any-variant="blocked" />
  <action disp="allocatable"  only-variants="simp both" />
  <action disp="allocatable"  only-variants="trad both" />
  <action disp="blocked"     any-variant="simp trad" />
  <action disp="allocatable"  comment="catch-all" />
</rules>
```

Up to now, variants with type "both" have occurred only associated
with reflexive variant mappings. The "action" elements defined above
rely on the assumption that this is always the case. However,
consider the following set of variants:

```
U+62E0;U+636E;U+636E;U+64DA
U+636E;U+636E;U+64DA;U+62E0
U+64DA;U+636E;U+64DA;U+62E0
```

The corresponding XML would be:

```
<char cp="62E0">
<var cp="636E" type="both" comment="both, but not reflexive" />
<var cp="64DA" type="blocked" />
</char>
<char cp="636E">
<var cp="636E" type="simp" comment="reflexive, but not both" />
<var cp="64DA" type="trad" />
<var cp="62E0" type="blocked" />
</char>
<char cp="64DA">
<var cp="636E" type="simp" />
<var cp="64DA" type="trad" comment="reflexive" />
<var cp="62E0" type="blocked" />
</char>
```

To make such variant sets work requires a way to selectively trigger
an action based on whether a variant type is associated with an
identity or reflexive mapping, or is associated with an ordinary
variant mapping. This can be done by adding a prefix "r-" to the
"type" attribute on reflexive variant mappings. For example the
"trad" for code point U+64DA in the preceding figure would become
"r-trad".

With the dispositions prepared in this way, only a slight
modification to the actions is needed to yield the correct set of
allocatable labels:

```
<action disp="blocked" any-variant="blocked" />
<action disp="allocatable" only-variants="simp r-simp both r-both" />
<action disp="allocatable" only-variants="trad r-trad both r-both" />
<action disp="blocked" all-variants="simp trad both" />
<action disp="allocatable" />
```

The first three actions get triggered by the same labels as before.

The fourth action blocks any label that combines an original code point with any mix of ordinary variant mappings; however no labels that are a combination of only original code points (code points having either no variant mappings or a reflexive mapping) would be affected. These are the original labels and they are allocated in the last action.

Using this scheme of assigning types to ordinary and reflexive variants, all RFC 3743-style tables can be converted to XML. By defining a set of actions as outlined above, the LGR will yield the correct set of allocatable variants: all variants consisting completely of variant code points preferred for simplified or traditional, respectively, will be allocated, as will be the original label. All other variant labels will be blocked.

Appendix C. Indic Syllable Structure Example

In LGRs for Indic scripts it may be desirable to restrict valid labels to sequences of valid Indic syllables, or aksharas. This appendix gives a sample set of rules designed to enforce this restriction.

An example of a BNF from for an akshara which has been published in "Devanagari Script Behavior for Hindi" [TDIL-HINDI]. The rules for other languages and scripts used in India are expected to be generally similar.

For Hindi, the BNF has the form:

```
V[m]│{C[N]H}C[N](H│[v][m])
```

Where:

V   (upper case) is any independent vowel

m   is any vowel modifier (Devanagari Anusvara, Visarga, and
     Candrabindu)

C   is any consonant (with inherent vowel)

    N   is Nukta

    H   is a Halant (or Virama)

    v   (lower case) is any dependent vowel sign (matra)

    {}   encloses items which may be repeated one or more times

    [ ] encloses items which may or may not be present

    | separates items, out of which only one can be present

    By using the Unicode character property "InSC" or
    "Indic_Syllabic_Category" which corresponds rather directly to the
    classification of characters in the BNF above, we can translate the
    BNF into a set of WLE rules matching the definition of an akshara.

     <rules>
        <!--Character Class Definitions go here-->
        <class name="halant" property="InSC:Virama" />
        <union name="vowel-modifier">
          <class property="InSC:Visarga" />
          <class property="InSC:Bindu" comment="includes anusvara" />
        </union>
        <!--Whole label evaluation and Context rules go here-->
        <rule name="consonant-with-optional-nukta">
            <class by-ref="InSC:Consonant" />
            <class by-ref="InSC:Nukta" count="0:1"/>
        </rule>
        <rule name="independent-vowel-with-optional-modifier">
            <class by-ref="InSC:Vowel_Independent" />
            <class by-ref="vowel-modifier" count="0:1" />
        </rule>
        <rule name="optional-dependent-vowel-with-opt-modifier" >
          <class by-ref="InSC:Vowel_Dependent" count="0:1" />
          <class by-ref="vowel-modifier" count="0:1" />
        </rule>
        <rule name="consonant-cluster">
          <rule count="0+">
            <rule by-ref="consonant-with-optional-nukta" />
            <class by-ref="halant" />
          </rule>
          <rule by-ref="consonant-with-optional-nukta" />
          <choice>
            <class by-ref="halant" />
            <rule by-ref="optional-dependent-vowel-with-opt-modifier" />
          </choice>
        </rule>

```
      <rule name="akshara">
        <choice>
          <rule by-ref="independent-vowel-with-optional-modifier" />
          <rule by-ref="consonant-cluster" />
        </choice>
      </rule>
      <rule name="WLE-akshara-or-other" comment="series of one or
          more aksharas, possibly alternating with other types of
          code points such as digits">
        <start />
        <choice count="1+">
          <class property="InSC:other"  />
          <rule by-ref="akshara"  />
        </choice>
        <end />
      </rule>
      <!--Action elements go here - order defines precedence-->
      <action disp="invalid" not-match="WLE-akshara-or-other" />
    </rules>
```

   With the rules and classes as defined above, the final action assigns
   a disposition of "invalid" to all labels that are not composed of a
   sequence of well-formed aksharas, optionally interspersed with other
   characters, perhaps digits, for example.

   The relevant Unicode character property could be replicated by
   tagging repertoire values directly in the LGR which would remove the
   dependency on any specific version of the Unicode Standard.

   Generally, dependent vowels may only follow consonant expressions,
   however, for some scripts, like Bengali, the Unicode standard
   supports sequences of dependent vowels or their application on
   independent vowels. This makes the definition of akshara less
   restrictive.

   It is possible to reduce the complexity of these rules by defining
   alternate rules which simply define the permissible pair-wise context
   of adjacent code points by character class--such as the rule that a
   Halant can only follow a (nuktated) consonant. (See the example in
   [WLE-RULES]).

Appendix D. RelaxNG Compact Schema

   <CODE BEGINS>
   #
   # LGR XML Schema 1.0
   #

```
default namespace = "urn:ietf:params:xml:ns:lgr-1.0"

#
# SIMPLE TYPES
#

# RFC 5646 language tag (e.g. "de", "und-Latn", etc.)
language-tag = xsd:token

# The scope to which the LGR applies. For the "domain" scope type it
# should be a fully qualified domain name.
scope-value = xsd:token {
    minLength = "1"
}

## a single code point
code-point = xsd:token {
    pattern = "[0-9A-F]{4,6}"
}

## a space-separated sequence of code points
code-point-sequence = xsd:token {
    pattern = "[0-9A-F]{4,6}( [0-9A-F]{4,6})+"
}

## single code point, or a sequence of code points, or empty string
code-point-literal = code-point | code-point-sequence | ""

## code point or sequence only
non-empty-code-point-literal = code-point | code-point-sequence

## code point sent represented in short form
code-point-set-shorthand = xsd:token {
    pattern = "([0-9A-F]{4,6}|[0-9A-F]{4,6}-[0-9A-F]{4,6})"
             ~ "( ([0-9A-F]{4,6}|[0-9A-F]{4,6}-[0-9A-F]{4,6}))*"
}

## dates are used in information fields in the meta
## section ("YYYY-MM-DD")
date-pattern = xsd:token {
    pattern = "\d{4}-\d\d-\d\d"
}

## variant type
## the variant type MUST be non-emtpy and MUST NOT
## start with a "_", using xsd:NMTOKEN here because
## we need space-separated lists of them
variant-type = xsd:NMTOKEN
```

```
    ## variant type list for action triggers
    ## the list MUST NOT be empty and entries MUST NOT
    ## start with a "_"
    variant-type-list = xsd:NMTOKENS

    ## reference to a rule name (used in "when" and "not-when"
    ## attributes, as well as the "by-ref" attribute of the "rule"
    ## element.)
    rule-ref = xsd:IDREF

    ## a space-separated list of tags. Tags should generally follow
    ## xsd:Name syntax. However, we are using the xsd:NMTOKENS here
    ## because there is no native XSD datatype for space-separated
    ## xsd:Name
    tags = xsd:NMTOKENS

    ## The value space of a "from-tag" attribute. Although it is closer
    ## to xsd:IDREF lexically and semantically, tags are not unique in
    ## the document. As such, we are unable to take advantage of
    ## facilities provided by a validator. xsd:NMTOKEN is used instead of
    ## the stricter xsd:Names here so as to be consistent with the above.
    tag-ref = xsd:NMTOKEN

    ## an identifier type (used by "name" attributes).
    identifier = xsd:ID

    ## used in the class "by-ref" attribute to reference another class of
    ## the same "name" attribute value.
    class-ref = xsd:IDREF

    ## count attribute pattern ("n", "n+" or "n:m")
    count-pattern = xsd:token {
        pattern = "\d+(\+|:\d+)?"
    }

    ## ref attribute pattern
    ## space separated list of id attribute valiues for
    ## reference elements. These reference ids
    ## must be declared in a reference element
    ## before they can be used in a ref attribute
    ref-pattern = xsd:token {
        pattern = "[\-_.:0-9A-Z]+( [\-_.:0-9A-Z]+)*"
    }

    #
    # STRUCTURES
    #
```

```
   ## Representation of a single code point, or a sequence of code
   ## points
   char = element char {
       attribute cp { code-point-literal },
       attribute comment { text }?,
       attribute when { rule-ref }?,
       attribute not-when { rule-ref }?,
       attribute tag { tags }?,
       attribute ref { ref-pattern }?,
         variant*
   }

   ## Representation of a range of code points
   range = element range {
       attribute first-cp { code-point },
       attribute last-cp { code-point },
       attribute comment { text }?,
       attribute when { rule-ref }?,
       attribute not-when { rule-ref }?,
       attribute tag { tags }?,
       attribute ref { ref-pattern }?
   }

   ## Representation of a variant code point or sequence
   variant = element var {
       attribute cp { code-point-literal },
       attribute type { xsd:NMTOKEN }?,
       attribute when { rule-ref }?,
       attribute not-when { rule-ref }?,
       attribute comment { text }?,
       attribute ref { ref-pattern }?
   }


   #
   # Classes
   #


   ## a "class" element that references the name of another "class"
   ## (or set-operator like "union") defined elsewhere.
   ## If used as a matcher (appearing under a "rule" element),
   ## the "count" attribute may be present.
   class-invocation = element class { class-invocation-content }

   class-invocation-content =
       attribute by-ref { class-ref },
       attribute count { count-pattern }?,
```

```
      attribute comment { text }?

   ## defines a new class (set of code points) using Unicode property
   ## or code points of the same tag value or code point literals
   class-declaration = element class { class-declaration-content }

   class-declaration-content =
       # "name" attribute MUST be present if this is a "top-level"
       # class declaration, i.e. appearing directly under the "rules"
       # element. Otherwise, it MUST be absent.
       attribute name { identifier }?,
       # If used as a matcher (appearing in a "rule" element, but not
       # when nested inside a set operator or class), the "count"
       # attribute may be present. Otherwise, it MUST be absent.
       attribute count { count-pattern }?,
       attribute comment { text }?,
       attribute ref { ref-pattern }?,
       (
         # define the class by property (e.g. property="sc:Latn"), OR
         attribute property { xsd:NMTOKEN }
         # define the class by tagged code points, OR
         | attribute from-tag { tag-ref }
         # text node to allow for shorthand notation
         # e.g. "0061 0062-0063"
         | code-point-set-shorthand
       )


   class-invocation-or-declaration = element class {
     class-invocation-content | class-declaration-content
   }

   class-or-set-operator-nested =
     class-invocation-or-declaration | set-operator

   class-or-set-operator-declaration =
     # a "class" element or set operator (effectively defining a class)
     # directly in the "rules" element.
     class-declaration | set-operator


   #
   # Set operators
   #

   complement-operator = element complement {
       attribute name { identifier }?,
       attribute comment { text }?,
```

```
        attribute ref { ref-pattern }?,
        # "count" attribute MUST only be used when this set-operator is
        # used as a matcher (i.e. nested in a <rule> element but not
        # inside a set-operator or class)
        attribute count { count-pattern }?,
        class-or-set-operator-nested
    }

    union-operator = element union {
        attribute name { identifier }?,
        attribute comment { text }?,
        attribute ref { ref-pattern }?,
        # "count" attribute MUST only be used when this set-operator is
        # used as a matcher (i.e. nested in a <rule> element but not
        # inside a set-operator or class)
        attribute count { count-pattern }?,
        class-or-set-operator-nested,
        # needs two or more child elements
        class-or-set-operator-nested+
    }

    intersection-operator = element intersection {
        attribute name { identifier }?,
        attribute comment { text }?,
        attribute ref { ref-pattern }?,
        # "count" attribute MUST only be used when this set-operator is
        # used as a matcher (i.e. nested in a <rule> element but not
        # inside a set-operator or class)
        attribute count { count-pattern }?,
        class-or-set-operator-nested,
        class-or-set-operator-nested
    }

    difference-operator = element difference {
        attribute name { identifier }?,
        attribute comment { text }?,
        attribute ref { ref-pattern }?,
        # "count" attribute MUST only be used when this set-operator is
        # used as a matcher (i.e. nested in a <rule> element but not
        # inside a set-operator or class)
        attribute count { count-pattern }?,
        class-or-set-operator-nested,
        class-or-set-operator-nested
    }

    symmetric-difference-operator = element symmetric-difference {
        attribute name { identifier }?,
        attribute comment { text }?,
```

```
        attribute ref { ref-pattern }?,
        # "count" attribute MUST only be used when this set-operator is
        # used as a matcher (i.e. nested in a <rule> element but not
        # inside a set-operator or class)
        attribute count { count-pattern }?,
        class-or-set-operator-nested,
        class-or-set-operator-nested
    }

    ## operators that transform class(es) into a new class.
    set-operator = complement-operator
                 | union-operator
                 | intersection-operator
                 | difference-operator
                 | symmetric-difference-operator

    #
    # Match operators (matchers)
    #

    any-matcher = element any {
        attribute count { count-pattern }?,
        attribute comment { text }?
    }

    choice-matcher = element choice {
        ## "count attribute MUST only be used when the choice-matcher
        ## contains no nested "start", "end", "anchor", "look-behind"
        ## or "look-ahead" operators and no nested rule-matchers
        ## containing any of these elements
        attribute count { count-pattern }?,
        attribute comment { text }?,
        # two or more match operators
        match-operator-choice,
        match-operator-choice+
    }

    char-matcher =
      # for use as a matcher - like "char" but without a "tag" attribute
      element char {
        attribute cp { non-empty-code-point-literal },
        # If used as a matcher (appearing in a "rule" element), the
        # "count" attribute may be present. Otherwise, it MUST be
        # absent.
        attribute count { count-pattern }?,
        attribute comment { text }?,
        attribute ref { ref-pattern }?
      }
```

```
start-matcher = element start {
    attribute comment { text }?
}

end-matcher = element end {
    attribute comment { text }?
}

anchor-matcher = element anchor {
    attribute comment { text }?
}

look-ahead-matcher = element look-ahead {
    attribute comment { text }?,
    match-operators-non-pos
}
look-behind-matcher = element look-behind {
    attribute comment { text }?,
    match-operators-non-pos
}

## non-positional match operator that can be used as a direct child
## element of the choice matcher.
match-operator-choice = (
  any-matcher | choice-matcher | start-matcher | end-matcher
   | char-matcher | class-or-set-operator-nested | rule-matcher
)

## non-positional match operators do not contain any anchor,
## look-behind or look-ahead elements.
match-operators-non-pos = (
  start-matcher?,
  (any-matcher | choice-matcher | char-matcher
   | class-or-set-operator-nested | rule-matcher)*,
  end-matcher?
)

## positional match operators have an anchor element, which may be
## preceeded by a look-behind element, or followed by a look-ahead
## element, or both.
match-operators-pos =
  look-behind-matcher?, anchor-matcher, look-ahead-matcher?

match-operators = match-operators-non-pos | match-operators-pos


#
# Rules
```

```
    #

    # top-level rule must have "name" attribute
    rule-declaration-top = element rule {
        attribute name { identifier },
        attribute comment { text }?,
        attribute ref { ref-pattern }?,
        match-operators
    }

    ## rule element used as a matcher (either by-ref or contains other
    ## match operators itself)
    rule-matcher =
      element rule {
        ## "count attribute MUST only be used when the rule-matcher
        ## contains no nested "start", "end", "anchor", "look-behind"
        ## or "look-ahead" operators and no nested rule-matchers
        ## containing any of these elements
        attribute count { count-pattern }?,
        attribute comment { text }?,
        attribute ref { ref-pattern }?,
        (attribute by-ref { rule-ref } | match-operators)
      }


    #
    # Actions
    #

    action-declaration = element action {
        attribute comment { text }?,
        attribute ref { ref-pattern }?,
        # dispositions are often named after variant types or vice versa
        attribute disp { variant-type },
        ( attribute match { rule-ref }
          | attribute not-match { rule-ref } )?,
        ( attribute any-variant { variant-type-list }
          | attribute all-variants { variant-type-list }
          | attribute only-variants { variant-type-list } )?
    }

    # DOCUMENT STRUCTURE

    start = lgr
    lgr = element lgr {
        meta-section?,
        data-section,
        rules-section?
```

```
    }

    ## Meta section - information recorded with an label
    ## generation ruleset that generally does not affect machine
    ## processing (except for unicode-version).
    ## However, if any "class-declaration" uses the "property" attribute
    ## a unicode-version MUST be present.
    meta-section = element meta {
        element version {
            attribute comment { text }?,
            text
        }?
        & element date { date-pattern }?
        & element language { language-tag }*
        & element scope {
            # type may by "domain" or an application-defined value
            attribute type { xsd:NCName },
            scope-value
        }*
        & element validity-start { date-pattern }?
        & element validity-end { date-pattern }?
        & element unicode-version {
            xsd:token {
                pattern = "\d+\.\d+\.\d+"
            }
        }?
        & element description {
            # this SHOULD be a valid MIME type
            attribute type { text }?,
            text
        }?
        & element references {
            element reference {
                attribute id {
                    xsd:token {
                        # limit id attribute to uppercase letters,
                        # digits and a few punctuation marks; use of
                        # integers is RECOMMENDED
                        pattern = "[\-_.:0-9A-Z]*"
                        minLength = "1"
                    }
                },
                attribute comment { text }?,
                text
            }*
        }?
    }
```

```
data-section = element data { (char | range)+ }

## Note that action declarations are strictly order dependent.
## class-or-set-operator-declaration and rule-declaration-top
## are weakly order dependent, they must precede first use of the
## identifier via by-ref.
rules-section = element rules {
  ( class-or-set-operator-declaration
    │ rule-declaration-top
    │ action-declaration)*
}

<CODE ENDS>
```

Appendix E. Acknowledgements

   This format builds upon the work on documenting IDN tables by many
   different registry operators. Notably, a comprehensive language
   table for Chinese, Japanese and Korean was developed by the "Joint
   Engineering Team" [RFC3743] that is the basis of many registry
   policies; and a set of guidelines for Arabic script registrations
   [RFC5564] was published by the Arabic-language community.

   Contributions that have shaped this document have been provided by
   Francisco Arias, Julien Bernard, Mark Davis, Martin Duerst, Paul
   Hoffman, Sarmad Hussain, Barry Leiba, Alexander Mayrhofer, Alexey
   Melnikov, Nicholas Ostler, Thomas Roessler, Audric Schiltknecht,
   Steve Sheng, Michel Suignard, Andrew Sullivan, Wil Tan and John
   Yunker.

Appendix F. Change History

   This appendix to be removed prior to final publication.

   draft-davies-idntables-00 Initial draft.

   draft-davies-idntables-01 Add an XML Namespace, and fix other XML
       nits. Add support for sequences of code points. Improve on
       consistently using Unicode nomenclature.

   draft-davies-idntables-02 Add support for validity periods.

   draft-davies-idntables-03 Incorporate requirements from the Label
       Generation Ruleset Procedure for the DNS Root Zone. These
       requirements include a detailed grammar for specifying whole-
       label variants, and the ability to explicitly declare of the
       actions associated with a specific variant. The document also
       consistently applies the term "Label Generation Ruleset", rather

than "IDN table", to reflect the policy term now being used to describe these.

draft-davies-idntables-04 Support reference information per [RFC3743]. Update description in response to feedback. Extend the context rules to "char" elements and allow for inverse matching ("not-when"). Extend the description of label processing and implied actions, and allow for actions that reference disposition attributes on any or all variant mappings used in the generation of a variant label.

draft-davies-idntables-05 Change the name of the "disposition" attribute to "disp". Add comment attribute on version and reference elements. Allow empty "cp" attributes in char elements to support expressing symmetric mapping of null variants. Describe use of variants that map identically. Clarify how actions are triggered, in particular based on variant dispositions, as well as description of default actions. Revise description of processing a label and its variants. Move example table at the head of appendices. Add "only-variants" attribute. Change "name" attribute to "by-ref" attribute for referencing named classes and rules. Change "not" to "complement". Remove "match" attribute on rules as redundant if "start" and "end" are supported. Rename "match" element to "anchor" as better fitting its function and removing confusion with both the "match" attribute on actions as well as the generic term Match Operator. Augmented the examples relevant to [RFC3743].

draft-davies-idntables-06 Extend the discussion of reflexive variants and their use; includes update of the appendix on converting tables in the style of [RFC3743]. Improve description of tagging and clarify that it doesn't apply to sequences. Specify that root zone uses ".". Add an appendix with an Indic Syllable Structure example. Extend count attribute to allow maximal counts.

draft-davies-idntables-07 Change "byref" to "by-ref". Add list of recommended properties. Change "location" to "positional" for collective name of start/end match operators. Use from-tag instead of by-ref for tag-based classes. Made optional or mutually exclusive nature of some attributes more explicit. Allowing "comment" attributes on all child elements of "rules" except "char" and "range" elements used as child elements of "class". Recast the design goals and requirements at the start of the document. Reword aspects of the document to make it clear the format's application is not limited only to domain names.

   draft-davies-idntables-08 Change "domain" to scope with
       type="domain". Reword in several places for clarity. Flesh out
       note on security. Change "disp" to "type" for variants, to mark
       that these attributes do not necessarily correspond one-to-one
       to variant label dispositions. Add example of variant type
       triggers. Remove "long form" of class definition.

   draft-davies-idntables-09 Grammatical updates, clarity improvements.
       Altered some DNS-specific terminology.

   draft-davies-idntables-10 Added convention for out-of-repertoire
       variants, additional examples of when rules in the context of
       symmetry, isolated minor copy editing. Use a URN as the XML
       namespace (provisional). Specify a media type for the file.

   draft-ietf-lager-specification-00 Update to reflect adoption as a
       work item by the IETF LAGER working group.

   draft-ietf-lager-specification-01 Update to reflect decisions in
       first interim meeting of IETF LAGER working group. Correcting a
       number of typos, added section on contextual conditions,
       clarified language on how actions are triggered, and changed
       "block", "allocate" and "activate" to "blocked", "allocatable".
       and "activated". Other minor changes.

   draft-ietf-lager-specification-02 Minor changes.

   draft-ietf-lager-specification-03 Update to fix a typo in the
       schema, and clarify the use of reflexive variants in checking
       label validity. Added security consideration for naive
       implementations of permuted labels and WLE rules. Added
       discussion of error conditions under which duplicate variant
       labels might be created. Other minor changes.

   draft-ietf-lager-specification-04 Updated XML namespace in the
       RelaxNG schema.

   draft-ietf-lager-specification-05 Add IANA Considerations for media
       type registration, URN registration and instantiating a
       dispositions registry. Split references into normative and
       informative. Describe a tighter restriction on permissible
       values for "ref" attributes. Clarify when "count" attributes
       are permitted. Typos fixed. Checked the schema against the
       specification and made corrections as well as replaced the
       "text" datatype with less permissive types for most attributes.

   draft-ietf-lager-specification-06 Change eligibility required for
       disposition registration to be Specification Required, rather

than Standards Track. Restore schema in appendix. Move
registration related procedures to informative references.

draft-ietf-lager-specification-07 Clarify the way the values of the
"property" attribute are constructed from the attribute name and
value in [UAX42]. Minor fixes to examples and wording for a few
of the match operators and the description for reflexive
variants. Clarified default action, added new disposition named
"valid" and changed "catch-all" action to a disposition of
"valid". Removed an ambiguity in the evaluation of labels for
LGRs that define sequences of code points as elements. Added
change controller for media type.

draft-ietf-lager-specification-08 Minor typographical fixes.

draft-ietf-lager-specification-09 Incorporated feedback from LAGER
WG Last Call. Fixed examples in 4.4.1 and 5.2.1. Reviewed
usage of RFC 2119 language, and other editorial suggestions.
Removed "und-Jpan" example as "ja" would suffice in normal
cases. Document approach to expanding "scope" element.

draft-ietf-lager-specification-10 Minor ID-nits related edits.
Integrate feedback on media type registration.

draft-ietf-lager-specification-11 Integrate feedback from AD review.

draft-ietf-lager-specification-12 Integrate additional feedback from
AD review. Use domain names for the prefixes in private
dispositions to reduce potential conflicts. Add clarifying
language on ordering, well-formedness, collision checking and
rules.

draft-ietf-lager-specification-13 Integrate additional feedback on
Security Considerations.

Authors' Addresses

Kim Davies
Internet Corporation for Assigned Names and Numbers
12025 Waterfront Drive
Los Angeles, CA 90094
US

Phone: +1 310 301 5800
Email: kim.davies@icann.org
URI: http://www.icann.org/

Asmus Freytag
ASMUS Inc.

Email: asmus@unicode.org